

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

Приемы объектно-ориентированного проектирования

паттерны проектирования

- паттерное проектирование как возможность повторного применения удачных решений
- принципы применения паттернов проектирования
- классификация паттернов
- различные подходы к выбору паттернов для решения конкретных задач
- пример проектирования редактора документов
- каталог паттернов с детальным описанием
- множество конкретных примеров

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, Massachusetts • Harlow, England • Menlo Park, California
Berkley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

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

БИБЛИОТЕКА ПРОГРАММИСТА

Приемы объектно-ориентированного проектирования

паттерны проектирования

Санкт-Петербург
Москва • Харьков • Минск

2001



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

**Приемы объектно-ориентированного проектирования
Паттерны проектирования**

Серия «Библиотека программиста»

, *Перевел с английского А. Слинкин*

Руководитель проекта

И. Захаров

Научный редактор

Н. Шалаев

Литературный редактор

А. Петроградская

С. Прока

Технический редактор

А. Бахарев

Иллюстрации

Н. Биржаков

Художник

Л. Пискунова

Верстка

ББК 32.973.2-018

УДК 681.3.068

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

П75 Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб: Питер, 2001. — 368 с.: ил. (Серия «Библиотека программиста»)

ISBN 5-272-00355-1

В предлагаемой 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. Авторы излагают принципы использования паттернов проектирования и приводят их каталог. Таким образом, книга одновременно решает две задачи. Во-первых, здесь демонстрируется роль паттернов в создании архитектуры сложных систем. Во-вторых, применяя содержащиеся в справочнике паттерны, проектировщик сможет с легкостью разрабатывать собственные приложения.

Издание предназначено как для профессиональных разработчиков, так и для программистов, осваивающих объектно-ориентированное проектирование.

Original English language Edition Copyright© 1995 by Addison Wesley Longman, Inc.

© Перевод на русский язык, А. Слинкин, 2001

© Серия, оформление, Издательский дом «Питер», 2001

Оригинал-макет подготовлен издательством «ДМК Пресс».

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственность за возможные ошибки, связанные с использованием книги.

ISBN 5-272-00355-1

ISBN 0-201-63361-2 (англ.)

ЗАО «Питер Бук». 196105, Санкт-Петербург, Благодатная ул., д. 67.

Лицензия ИД № 01940 от 05.06.00.

Налоговая льгота - общероссийский классификатор продукции ОК 005-93, том 2; 953000 - книги и брошюры.

Подписано в печать 08.10.00. Формат 70x100/. Усл. п. л. 29,67. Тираж 5000 экз. Заказ№ 1997.

Отпечатано с готовых диапозитивов в ГПП «Печатный двор» Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.

197110, Санкт-Петербург, Чкаловский пр., 15.

Посвящается
Кэрин
- Э. Гамма

Сильвии
- Р. Хелм

Фейт
- Р. Джонсон

Дрю Энн и Мэттью
Джошуа 24:15Ь
- Дж. Влиссидес

Отзывы на книгу «*Design Patterns: Elements of Reusable Object-Oriented Software*»

Одна из самых хорошо написанных и глубоких книг, которые мне доводилось читать... эта работа доказывает необходимость паттернов самым правильным способом: не рассуждениями, а примерами.

Стэн Липпман, *The C++ Report*

...Новая книга Гаммы, Хелма, Джонсона и Влиссидеса окажет важное и продолжительное воздействие на науку проектирования программного обеспечения. Поскольку авторы преподносят свой труд как относящийся только к объектно-ориентированным программам, боюсь, что многие разработчики, не занимающиеся объектной проблематикой, могут не обратить на книгу внимания. Это будет большой ошибкой. На самом деле каждый, кто занимается проектированием программ, найдет здесь много интересного для себя. Все проектировщики применяют паттерны, поэтому более глубокое понимание повторно используемых абстракций лишь пойдет нам на пользу.

Том ДеМарко, *IEEE Software*

Полагаю, что книга чрезвычайно ценна, поскольку описывает богатейший опыт объектно-ориентированного проектирования, изложенный в компактной, удобной для многократного применения форме. Безусловно, я снова и снова буду обращаться к идеям, представленным в издании, а ведь именно в этом и состоит суть повторного использования, не так ли?

Санджив Госсайн. *Journal of Object-Oriented Programming*

Эта книга, столь давно ожидаемая, полностью оправдала все предшествовавшие ей рекламные посулы. Она подобна справочнику архитектора, в котором приведены проверенные временем, испытанные на практике приемы и методы проектирования. Всего авторы отобрали 23 паттерна. Подарите экземпляр этой книги каждому из своих знакомых программистов, желающему усовершенствоваться в своей профессии.

Ларри О'Брайен, *Software Development*

Следует признать, что паттерны могут полностью изменить подходы к инженерному проектированию программ, привнеся в эту область изящество по-настоящему элегантного дизайна. Из всех имеющихся книг на эту тему «Паттерны проектирования», безусловно, лучшая. Ее следует читать, изучать и переводить на другие языки. Она раз и навсегда изменит ваш взгляд на программное обеспечение.

Стив Билов, *Journal of Object-Oriented Programming*

«Паттерны проектирования» - замечательная книга. Потратив на ее чтение сравнительно немного времени, большинство программистов на языке С++ смогут начать применять паттерны в своей работе, что улучшит качество создаваемых ими программ. Эта книга передает в наше распоряжение конкретные инструменты, помогающие более эффективно мыслить и выражать свои идеи. Она может фундаментально изменить ваш взгляд на программирование.

Том Каргилл, *The C++ Report*



Предисловие.....	10
Глава 1. Введение в паттерны проектирования.....	15
1.1. Что такое паттерн проектирования.....	16
1.2. Паттерны проектирования в схеме MVC в языке Smalltalk.....	18
1.3. Описание паттернов проектирования.....	20
1.4. Каталог паттернов проектирования.....	22
1.5. Организация каталога.....	24
1.6. Как решать задачи проектирования с помощью паттернов.....	25
Поиск подходящих объектов.....	25
Определение степени детализации объекта.....	27
Спецификация интерфейсов объекта.....	27
Спецификация реализации объектов.....	29
Механизмы повторного использования.....	32
Сравнение структур времени выполнения и времени компиляции.....	37
Проектирование с учетом будущих изменений.....	38
1.7. Как выбирать паттерн проектирования.....	43
1.8. Как пользоваться паттерном проектирования.....	44
Глава 2. Проектирование редактора документов.....	46
2.1. Задачи проектирования.....	46
2.2. Структура документа.....	48
Рекурсивная композиция.....	49
Глифы.....	51
Паттерн компонент.....	53
2.3. Форматирование.....	53
Инкапсуляция алгоритма форматирования.....	54
Классы Compositor и Composition.....	54
Стратегия.....	56
2.4. Оформление пользовательского интерфейса.....	56
Прозрачное обрамление.....	57
Моноглиф.....	58
Паттерн декоратор.....	60
2.5. Поддержка нескольких стандартов внешнего облика.....	60
Абстрагирование создания объекта.....	61
Фабрики и изготовленные классы.....	61
Паттерн абстрактная фабрика.....	64

2.6. Поддержка нескольких оконных систем.....	64
Можно ли воспользоваться абстрактной фабрикой?.....	64
Инкапсуляция зависимостей от реализации.....	65
Классы Window и WindowImp.....	67
Подклассы WindowImp.....	68
Конфигурирование класса Window с помощью WindowImp.....	70
Паттерн мост.....	70
2.7. Операции пользователя.....	71
Инкапсуляция запроса.....	72
Класс Command и его подклассы.....	73
Отмена операций.....	74
История команд.....	75
Паттерн команда.....	76
2.8. Проверка правописания и расстановка переносов.....	76
Доступ к распределенной информации.....	77
Инкапсуляция доступа и порядка обхода.....	77
Класс Iterator и его подклассы.....	78
Паттерн итератор.....	81
Обход и действия, выполняемые при обходе.....	81
Класс Visitor и его подклассы.....	86
Паттерн посетитель.....	87
2.9. Резюме.....	88
Глава 3. Порождающие паттерны.....	89
Паттерн Abstract Factory.....	93
Паттерн Builder.....	102
Паттерн Factory Method	111
Паттерн Prototype.....	121
Паттерн Singleton	130
Обсуждение порождающих паттернов.....	138
Глава 4. Структурные паттерны.....	140
Паттерн Adapter.....	141
Паттерн Bridge.....	152
Паттерн Composite.....	162
Паттерн Decorator.....	173
Паттерн Facade.....	183
Паттерн Flyweight.....	191
Паттерн Proxy.....	203
Обсуждение структурных паттернов.....	213
Адаптер и мост.....	213
Компоновщик, декоратор и заместитель.....	214
Глава 5. Паттерны поведения.....	216
Паттерн Chain of Responsibility.....	217
Паттерн Command.....	227

Паттерн Interpreter.....	236
Паттерн Iterator.....	249
Паттерн Mediator.....	263
Паттерн Memento.....	272
Паттерн Observer.....	280
Паттерн State.....	291
Паттерн Strategy.....	300
Паттерн Template Method.....	309
Паттерн Visitor.....	314
Обсуждение паттернов поведения.....	328
Инкапсуляция вариаций.....	328
Объекты как аргументы.....	328
Должен ли обмен информацией быть инкапсулированным или распределенным ...	329
Разделение получателей и отправителей.....	330
Резюме.....	332
Глава 6. Заключение.....	333
6.1. Чего ожидать от паттернов проектирования.....	333
Единый словарь проектирования.....	333
Помощь при документировании и изучении.....	334
Дополнение существующих методов.....	334
Цель реорганизации.....	335
6.2. Краткая история.....	336
6.3. Проектировщики паттернов.....	337
Языки паттернов Александра	338
Паттерны в программном обеспечении.....	339
6.4. Приглашение.....	339
6.5. На прощание.....	340
Приложение А. Глоссарий.....	341
Приложение В. Объяснение нотации.....	344
8. 1. Диаграмма классов.....	344
8.2. Диаграмма объектов.....	345
8.3. Диаграмма взаимодействий.....	346
Приложение С. Базовые классы.....	348
C. 1. List.....	348
C.2. Iterator.....	350
C.3. ListIterator.....	350
C.4. Point.....	351
C.5. Rect.....	351
Библиография.....	353
Алфавитный указатель.....	359

Предисловие

Данная книга не является введением в объектно-ориентированное программирование или проектирование. На эти темы есть много других хороших изданий. 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 отлиты в краткую и легко применимую на практике форму.

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 сомнении 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 все понимали, когда начинали писать ее! Помните, что эта книга не из тех, которых, однажды прочитав, ставят на полку. Мы 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 жизни 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 Билемельда (Roger Bielefeld), Грейди Буча (Grady Booch), Тома Каргилла (Tom Cargill), Маршалла Клайна (Marshall Cline), Ральфа Хайра (Ralph Hyde), 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 Addison-Wesley за поддержку и терпение: Кейту Хабибу (Kate Habib), Тиффани Мур (Tiffany Moore), Лайзе Раффаэле (Lisa Raffaele), Прадипу Сива (Pradeepa Siva) и Джону Уэйтю (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 через Internet, ободряли нас и убеждали, что такая работа действительно нужна. Вот далеко не полный перечень наших «незнакомых помощников»: Ион Авотинс (^{Jon} Avotins), Стив Берчук (Steve Berczik), Джуліан Бердич (Julian Berdych), Матиас Болен (Matthias Bohlen), Джон Брант (John Brant), Аллан Кларк (Allan Clarke), Пол Чизхолм (Paul Chisholm), Йене Колдьюи (Jens Coldewey), Дэйв 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 Мак-Комб (Dave McComb), Карл МакКоннелл (Carl McConnell), Кристин Минганс (Christine Mingins), Ханспетер Мессенбек (Hanspeter Mossenbock), Эрик Ньютон (Eric Newton), Марианна Озкан (Marianne Ozkan), Роксан Пайетт (Roxsan Payette), Ларри Подмолик (Larry Podmolik), Джордж Радин (George Radin), Сита Рамакришнан (Sita Ramakrishnan), Русс Рамирес (Russ Ramirez), Александр Ран (Alexander Ran), Дирк Риэле (Dirk Riehle), Брайан Розенбург (Bryan Rosenburg), Аамод Сайн (Aamod Sane), Дури Шмидт (Duri Schmidt), Роберт Зайдль (Robert Seidl), Хин Шу (Xin Shu) и Билл Уолкер (Bill Walker).

Мы не считаем, что набор отобранных нами паттернов полон и неизменен, он просто отражает нынешнее состояние наших мыслей 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 Web-страница <http://st-www.cs.uiuc.edu/users/patterns/DPBook/DPBook.html>, на которой размещается последняя информация и обновления к книге.

Эрих Гамма
Ричард Хелм
Ральф Джонсон
Джон Влиссидес

Маунтин Вью, штат Калифорния
Монреаль, Квебек
Урбана, штат Иллинойс
Готорн, штат Нью-Йорк

Август, 1994

Вступительное слово

Любая хорошо структурированная объектно-ориентированная архитектура изобилует паттернами. На самом деле, для меня одним из критериев качества объектно-ориентированной системы является то, насколько внимательно разработчики отнеслись к типичным взаимодействиям между участвующими в ней объектами. Если таким механизмам на этапе проектирования системы уделялось достаточное внимание, то архитектура получается более компактной, простой и понятной, чем в случае, когда наличие паттернов игнорировалось.

Важность паттернов при создании сложных систем давно осознана в других дисциплинах. Так, Кристофер Александр и его сотрудники, возможно, впервые предложили применять язык паттернов для архитектурного проектирования зданий и городов. Эти идеи, развитые затем другими исследователями, ныне глубоко укоренились в объектно-ориентированном проектировании. В двух словах концепция паттерна проектирования в программировании - это ключ к использованию разработчиками опыта высококвалифицированных коллег.

В данной работе излагаются принципы применения паттернов проектирования и приводится каталог таких паттернов. Тем самым книга решает сразу две задачи. Во-первых, она демонстрирует роль паттернов в проектировании архитектуры сложных систем. Во-вторых, содержит практический справочник удачны паттернов, которые разработчик может применить в собственных приложения.

Мне выпала честь работать вместе с некоторыми авторами книги над проблемами архитектурного дизайна. Я многому научился у этих людей и полагаю, что то же самое можно будет сказать и о вас, после того как вы прочтете эту книгу.

Грейди Буч,
Главный научный сотрудник
Rational Software Corporation

Советы читателю

Книга состоит из двух частей. В главах 1 и 2 рассказывается, что такое паттерны проектирования и как с их помощью можно разрабатывать объектно-ориентированные программы. Практическое применение паттернов проектирования демонстрируется на примерах. Главы 3,4 и 5 - это каталог паттернов проектирования.

Каталог занимает большую часть книги. Три главы отражают деление паттернов на категории: порождающие паттерны, структурные паттерны и паттерны поведения. Каталогом можно пользоваться по-разному: читать его с начала и до конца или переходить от одного паттерна к другому. Удачен и другой путь: тщательно изучить любую главу, чтобы понять, чем отличаются тесно связанные между собой паттерны.

Ссылки между паттернами дают возможность сориентироваться в каталоге. Это также позволит уяснить, как различные паттерны связаны друг с другом, как их можно сочетать между собой и какие паттерны могут хорошо работать совместно. На рис. 1.1 связи между паттернами изображены графически.

Можно читать каталог, ориентируясь на конкретную задачу. В разделе 1.6 вы найдете описание некоторых типичных задач, возникающих при проектировании повторно используемых объектно-ориентированных программ. После этого переходите к изучению паттернов, предназначенных для решения интересующей вас задачи. Некоторые предпочитают сначала ознакомиться со всем каталогом, а потом применить те или иные паттерны.

Если ваш опыт объектно-ориентированного проектирования невелик, начните изучать самые простые и распространенные паттерны:

- а абстрактная фабрика;
- а адаптер;
- а компоновщик;
- а декоратор;
- а фабричный метод;
- а наблюдатель;
- а стратегия;
- а шаблонный метод.

Трудно найти объектно-ориентированную систему, в которой не используются хотя бы некоторые из указанных паттернов, а уж в больших системах встречаются чуть ли не все. Разобравшись с этим подмножеством, вы получите представление как о некоторых определенных паттернах, так и об объектно-ориентированном проектировании в целом.

Принятые в книге обозначения

Для облегчения работы с книгой издательством «ДМК Пресс» приняты следующие соглашения:

- а коды программ, важные операторы, классы, объекты, методы и свойства обозначены в книге специальным шрифтом (Courier), что позволяет легко найти их в тексте;
- а *смыловые акценты, определения, термины*, встретившиеся впервые, обозначены курсивом;
- а **команды, пункты меню, кнопки, клавиши, функции** выделены полужирным шрифтом.

Глава 1. Введение в паттерны проектирования

Проектирование объектно-ориентированных программ - нелегкое дело, а если их нужно использовать повторно, 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 существенные 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 обеспечить «правильный», то есть в достаточной мере гибкий и пригодный для повторного использования дизайн, с первого раза 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 становится понятно, что же такое удачный объектно-ориентированный дизайн. Опытные проектировщики, очевидно, знают какие-то тонкости, ускользающие от новичков. Так что же это?

Прежде всего, опытному разработчику понятно, 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 становится экспертом в своей области. Во многих объектно-ориентированных системах вы встретите повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться 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 совершененно новые сюжеты. Вместо этого они берут за основу уже отработанные в мировой литературе схемы, жанры и образы. Например, трагический герой - Макбет,

Введение в паттерны проектирования

Гамлет и т.д., мотив убийства - деньги, месть, ревность и т.п. 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 их свойства».

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. Некоторые из наиболее распространенных паттернов 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 облегчает доступ к ним со стороны разработчиков новых систем. С помощью паттернов можно улучшить качество документации и сопровождения существующих систем, позволяя 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.

Как уже было сказано, в книгу включены только такие паттерны, которые неоднократно применялись в разных системах. По большей части они никогда ранее не документировались и либо известны самым квалифицированным специалистам по объектно-ориентированному проектированию, либо были частью какой-то удачной системы.

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 параллельности, 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, и, может быть, кто-то их и систематизирует.

1.1. Что такое паттерн проектирования

По словам Кристофера Александра, «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, ничего не изобретая заново» [AIS+77]. Хотя Александр имел в виду паттерны, возникающие при проектировании зданий и городов, но его слова верны и в отношении паттернов объектно-ориентированного проектирования. Наши решения выражаются

Чтотакоепаттернпроектирования

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*. Сославшись на него, мы можем сразу описать проблему проектирования; ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы. Нахождение хороших имен было одной из самых трудных задач при составлении каталога.
2. *Задача*. Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.
3. *Решение*. Описание элементов дизайна, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн - это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (в нашем случае классов и объектов).
4. *Результаты* - это следствия применения паттерна и разного рода компромиссы. 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. Здесь речь идет о выборе языка и 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.

То, что один воспринимает как паттерн, для другого просто строительный блок. В этой книге мы рассматриваем паттерны на определенном уровне абстракции. *Паттерны проектирования* - это не то же самое, что связанные списки или хэш-таблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. Но это и не сложные, предметно-ориентированные решения для целого приложения или подсистемы. Здесь 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 вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. Анализируется, - когда следует применять паттерн, можно ли

Введение в паттерны проектирования

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, строго говоря, паттерны используются в проектировании, 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 объектно-ориентированных языках с динамической типизацией (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из pragматических соображений, поскольку чаще всего работаем с ними и поскольку 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. 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).

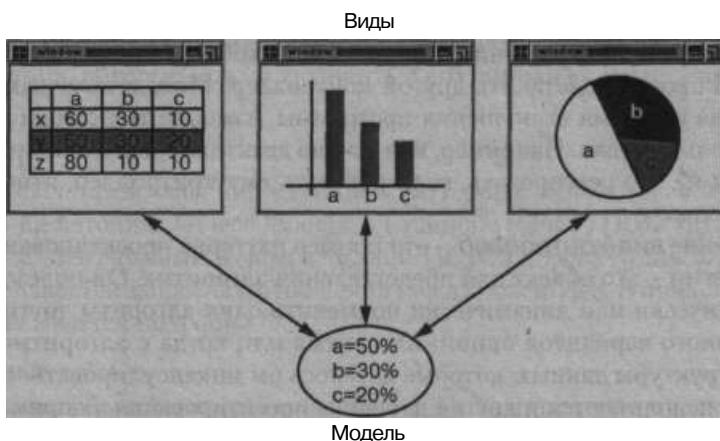
1.2. Паттерны проектирования в схеме MVC в языке Smalltalk

В Smalltalk-80 для построения интерфейсов пользователя применяется тройка классов модель/вид/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 *вид* - экранное представление. *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 вид от модели, устанавливая между ними протокол взаимодействия «подписка/оповещение». Вид должен гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель оповещает все зависящие от нее виды, в результате чего вид обновляет себя. Такой подход позволяет присоединить к одной модели несколько видов, обеспечив тем самым различные представления. Можно создать новый вид, не переписывая модель.

На рисунке ниже показана одна модель и три вида. (Для простоты мы опустили контроллеры.) Модель содержит некоторые данные, которые могут быть представлены в виде электронной таблицы, гистограммы и круговой диаграммы.



Модель оповещает свои виды при каждом изменении значений данных, а виды обращаются к модели для получения новых значений.

На первый взгляд, в этом примере продемонстрирован просто дизайн, отделяющий вид 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 деталях реализации объектов, на которые он оказал воздействие. Этот более общий подход описывается паттерном проектирования наблюдатель.

Еще одно свойство MVC заключается в том, что виды могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составной вид, содержащий вложенные, - по одной кнопке на каждый. Пользовательский интерфейс инспектора объектов может состоять из вложенных видов, используемых также и в отладчике. MVC поддерживает вложенные виды с помощью класса `CompositeView`, являющегося подклассом `View`. Объекты класса `CompositeView` ведут себя так же, как объекты класса `View`, поэтому могут использоваться всюду, где и виды. Но еще они могут содержать вложенные виды и управлять ими.

Здесь можно было бы считать, что этот дизайн позволяет обращаться с составным видом, 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 вида на действия пользователя. При этом визуальное представление остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать всплывающие 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.

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

Отношение вид-контроллер – это пример паттерна проектирования стратегия. Стратегия – это объект для представления алгоритма. Он полезен, когда вы хотите статически или динамически подменить один алгоритм другим, если существует много вариантов одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать.

В МУС используются и другие паттерны проектирования, например фабричный метод, позволяющий задать для вида класс контроллера по умолчанию, и декоратор для добавления к виду возможности прокрутки. Но основные отношения в схеме МУС описываются паттернами наблюдатель, компоновщик и стратегия.

1.3. Описание паттернов проектирования

Как мы будем описывать паттерны проектирования? Графических обозначений 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 позволяют увидеть применение паттерна.

При описании паттернов проектирования мы будем придерживаться единого принципа. 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 четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая изложена в разделе 1.5.

Назначение

Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.

Известен также под именем

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 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 В эта нотация описывается подробно.

Участники

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

Отношения

Взаимодействие участников для выполнения своих функций.

Результаты

Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?

Реализация

Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?

Пример кода

Фрагмент кода, иллюстрирующий вероятную реализацию на языках C++ или Smalltalk.

Известные применения

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

Родственные паттерны

Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

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

1.4. Каталог паттернов проектирования

Каталог 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) (93)

Предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

Adapter (адаптер) (141)

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

Bridge (мост) (152)

Отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.

Builder (строитель) (103)

Отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений.

Chain of Responsibility (цепочка обязанностей) (217)

Можно избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока какой-то объект его не обработает.

Command (команда) (226)

Инкапсулирует запрос в виде объекта, позволяя тем самым параметризовать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.

Composite (компоновщик) (162)

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) (173)

Динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.

Facade (фасад) (183)

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

Factory Method (фабричный метод) (111)

Определяет интерфейс для создания объектов, при этом выбранный класс инстанцируется подклассами.

Flyweight (приспособленец) (191)

Использует разделение для эффективной поддержки большого числа мелких объектов.

Interpreter (интерпретатор) (236)

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

Iterator (итератор) (173)

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

Mediator (посредник) (263)

Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

Memento (хранитель) (272)

Позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

Observer (наблюдатель) (281)

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

Prototype (прототип) (121)

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

Proxy (заместитель) (203)

Подменяет другой объект для контроля доступа к нему.

Singleton (одиночка) (130)

Гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.

State (состояние) (291)

Позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта.

Strategy (стратегия) (300)

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

Template Method (шаблонный метод) (309)

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

Visitor (посетитель) (314)

Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

1.5. Организация каталога

Паттерны проектирования различаются степенью детализации и уровнем абстракции и должны быть каким-то образом организованы. В данном разделе описывается классификация, позволяющая ссылаться на семейства взаимосвязанных паттернов. Она поможет быстрее освоить паттерны, описываемые в каталоге, и укажет направление поиска новых.

Мы будем классифицировать паттерны по двум критериям (табл. 1.1). Первый - *цель* - отражает назначение паттерна. В связи с этим выделяются порождающие паттерны, структурные паттерны 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 между собой.

Таблица 1.1. Пространство паттернов проектирования

Цель Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Второй критерий - *уровень* - говорит о том, 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 сфокусированы лишь на отношениях между классами. Обратите внимание: большинство паттернов действуют 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, совместно функционируют и выполняют задачу, которая ни одномуциальному 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.

1.6. Как решать задачи проектирования с помощью паттернов

Паттерны проектирования позволяют разными способами решать многие задачи, с которыми постоянно 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*.

Посылка запроса - это *единственный* способ заставить объект выполнить операцию. А выполнение операции - *единственный* способ изменить внутреннее состояние объекта. Имея в виду два эти ограничения, говорят, что внутреннее состояние объекта *инкапсулировано*: к нему нельзя получить непосредственный доступ, то есть представление объекта закрыто от внешней программы.

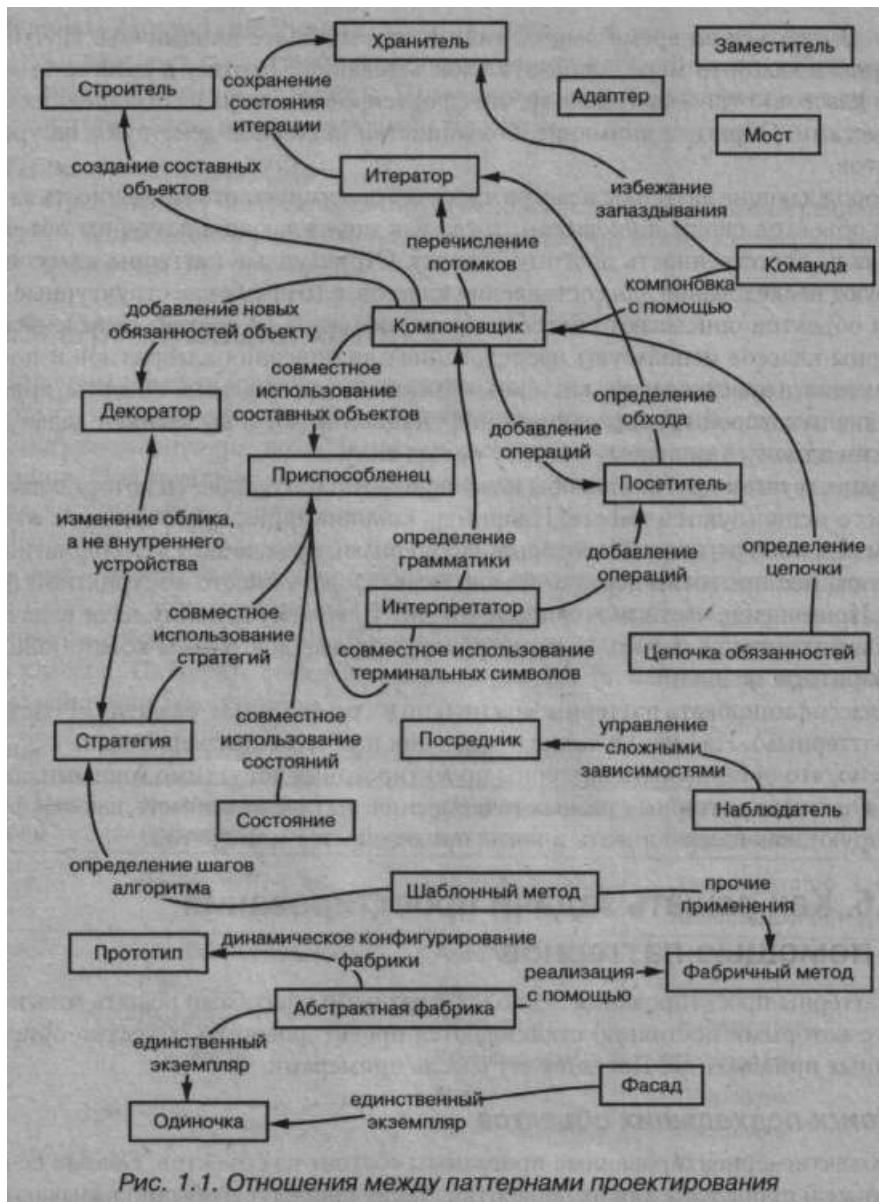


Рис. 1.1. Отношения между паттернами проектирования

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

Методики объектно-ориентированного проектирования отражают разные подходы. Вы можете 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 на свой дизайн. Согласие по поводу того, какой подход самый лучший, никогда не будет достигнуто.

Многие объекты возникают в проекте из построенной в ходе анализа модели. Но нередко появляются и классы, у которых нет прототипов в реальном мире. Это могут быть классы как низкого уровня, например массивы, так и высокого. Паттерн компоновщик 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 описывает способ реализации взаимозаменяемых семейств алгоритмов. Паттерн состояние позволяет представить состояние некоторой сущности в виде объекта. Эти объекты редко появляются во время анализа и даже на ранних стадиях проектирования. Работа с ними начинается позже, при попытках сделать дизайн более гибким и пригодным для повторного использования.

Определение степени детализации объекта

Размеры и число объектов могут сильно варьироваться. С их помощью может быть представлено все, начиная с уровня аппаратуры и до законченных приложений. Как же решить, что должен представлять собой объект?

Здесь и потребуются паттерны проектирования. Паттерн фасад показывает, как представить в виде объекта целые подсистемы, а паттерн приспособленец - как поддержать большое число объектов при высокой степени детализации. Другие паттерны указывают путь к разложению объекта на меньшие подобъекты. Абстрактная срабрика и строитель описывают объекты, единственной целью которых является создание других объектов, а посетитель и команда - объекты, отвечающие за реализацию запроса к другому объекту или группе.

Спецификация интерфейсов объекта

При объявлении объектом любой операции 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 соответствует интерфейсу объекта, может быть ему послан.

Type — это 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. Часть интерфейса объекта может быть охарактеризована одним типом, а часть — другим. Два объекта одного и того же типа должны 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 *супертипов* для первого. Часто говорят также, что подтип *наследует* интерфейс своего супертипа.

В объектно-ориентированных системах интерфейсы фундаментальны. Об объектах известно только то, что они сообщают о себе через свои 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 посыпается запрос, то операция, которую он будет выполнять, зависит как от запроса, так и от объекта-адресата. Разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на такие запросы разные операции. Ассоциация 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, допустимо написать программу, которая ожидает 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 точно такой же интерфейс. Такая взаимозаменяемость называется *полиморфизмом* и является важнейшей особенностью объектно-ориентированных систем. Она позволяет клиенту не делать почти никаких предположений об объектах, кроме того, что они поддерживают определенный интерфейс. Полиморфизм упрощает определение клиентов, позволяет отделить объекты друг от друга и дает объектам возможность изменять взаимоотношения во время выполнения.

Паттерны проектирования позволяют определять интерфейсы, задавая их основные элементы и то, какие данные можно передавать через интерфейс. Паттерн может также «сказать», что не должно проходить через интерфейс. Хорошим примером в этом отношении является хранитель. 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 хранители, а другой привилегированный, которым может пользоваться только сам объект для сохранения и извлечения информации о состоянии их хранителя.

Паттерны проектирования специфицируют также отношения между интерфейсами. В частности, нередко они содержат требование, что некоторые классы должны иметь схожие интерфейсы, а иногда налагают ограничения на интерфейсы классов. Так, декоратор и заместитель требуют, чтобы интерфейсы объектов этих паттернов были идентичны интерфейсам декорируемых и замещаемых объектов соответственно. Интерфейс объекта, принадлежащего паттерну посетитель, должен отражать все классы объектов, с которыми он будет работать.

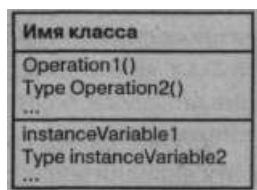
Спецификация реализации объектов

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

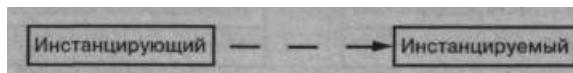
В нашей нотации, основанной на ОМТ (см. приложение В), класс изображается в виде прямоугольника, внутри которого жирным шрифтом написано имя класса. Ниже обычным шрифтом перечислены операции. Любые данные, которые определены для класса, следуют после операций. Имя класса, операции и данные разделяются горизонтальными линиями.

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

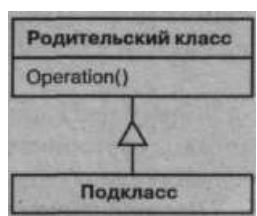
Объекты создаются с помощью *инстанцирования* класса. Говорят, что объект является *экземпляром* класса. В процессе инстанцирования выделяется память для *переменных экземпляра* (внутренних данных объекта), и с этими данными ассоциируются операции. С помощью инстанцирования одного класса можно создать много разных объектов-экземпляров.



Пунктирная линия со стрелкой обозначает класс, который инстанцирует объекты другого класса. Стрелка направлена в сторону класса инстанцированного объекта.



Новые классы можно определить в терминах существующих с помощью *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 подклассе и его предках. Отношение «является подклассом» обозначается вертикальной линией с треугольником.

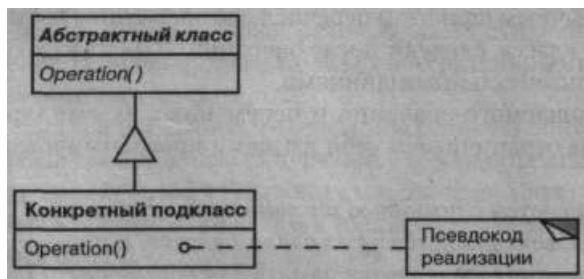


Класс называется *абстрактным*, если его единственное назначение - определить общий интерфейс для всех своих подклассов. Абстрактный класс делегирует

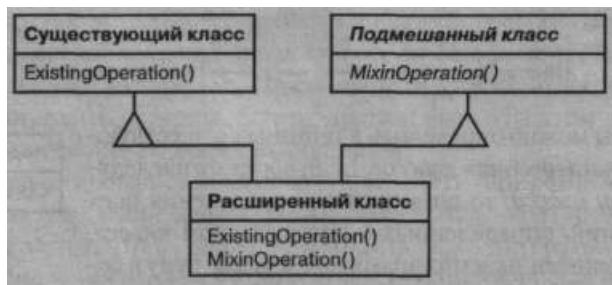
реализацию всех или части своих операций подклассам, поэтому у него не может быть экземпляров. Операции, объявленные, но не реализованные в абстрактном классе, называются *абстрактными*. Класс, не являющийся абстрактным, называется *конкретным*.

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. Тем самым можно без труда определять семейства объектов со схожей функциональностью.

Имена абстрактных классов оформлены 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, соединенным пунктирной линией с операцией, которую он реализует.



Подмешанным (mixin class) называется класс, назначение которого - предоставить дополнительный интерфейс или функциональность другим классам. Он родственен абстрактным классам в том смысле, что не предполагает непосредственного инстанцирования. Для работы с подмешанными классами необходимо множественное наследование.



Наследование класса и наследование интерфейса

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

Класс объекта определяет, как объект реализован, то есть внутреннее состояние и реализацию операций объекта. Напротив, тип относится только к интерфейсу

объекта - множеству запросов, на которые объект отвечает. У объекта может быть много типов, и объекты разных классов могут иметь один и тот же тип.

Разумеется, между классом и типом есть тесная связь. Поскольку класс определяет, какие операции может выполнять объект, то заодно он 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 используются для специфирования, типа и реализации объекта. В программах на языке Smalltalk типы переменных не объявляются, поэтому компилятор не проверяет, что тип объекта, присваиваемого переменной, является подтипов 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.

Две эти 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 исключительно виртуальные функции. Истинное наследование интерфейса можно аппроксимировать в C++ с помощью открытого наследования абстрактному классу. Истинное наследование реализации или класса аппроксимируется с помощью 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 предпочитают считать, 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, должны иметь общий тип, но их реализация обычно различна. В паттерне компоновщик отдельный объект (компонент) определяет общий интерфейс, но реализацию часто определяет составной объект (композиция). Паттерны команда, наблюдатель, состояние и стратегия часто реализуются абстрактными классами с исключительно виртуальными функциями.

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 в терминах уже имеющегося. Новую реализацию вы можете получить посредством наследования большей части необходимого кода из ранее написанных классов.

Однако 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 операций, определенных 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:

- а клиенту не нужно иметь информации о конкретных типах объектов, которыми он пользуется, при условии, что все они имеют ожидаемый клиентом интерфейс;
- а 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. Это одна из наших ключевых идей.

Конечно, где-то в системе вам придется инстанцировать конкретные классы, то есть определить конкретную реализацию. Как раз это и позволяют сделать порождающие паттерны: абстрактная фабрика, строитель, фабричный метод, прототип и одиночка. Абстрагируя процесс создания объекта, эти паттерны предоставляют вам разные способы прозрачно ассоциировать интерфейс с его реализацией в момент инстанцирования. Использование порождающих паттернов гарантирует, что система написана в терминах 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 вы сможете сделать это проще.

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 в терминах другого. Повторное использование за счет порождения подкласса называют еще *прозрачным ящиком* (*white-box reuse*). Такой термин подчеркивает, что внутреннее устройство родительских классов видимо подклассам.

Композиция объектов - это альтернатива наследованию класса. В этом случае новую, более сложную функциональность мы получаем путем объединения или композиции объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ повторного использования называют *черным ящиком* (*black-box 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, поскольку не исключено, что они вызывают замещенные.

Но у наследования класса есть и минусы. Во-первых, нельзя изменить унаследованную от родителя реализацию во время выполнения программы, поскольку само наследование фиксировано на этапе компиляции. Во-вторых, 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 настолько тесно связаны, 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. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик. Поскольку доступ к объектам осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку при

реализации объекта кодируются прежде всего его интерфейсы, то зависимость от реализации резко снижается.

Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию 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), и 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, чтобы добиться повторного использования, вообще не следовало бы создавать новые компоненты. Хорошо бы, чтобы можно было получить всю нужную функциональность, просто собирая вместе уже существующие компоненты. 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.

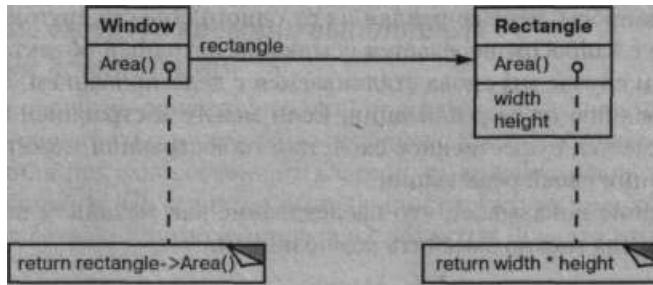
Тем не менее, наш опыт показывает, что проектировщики злоупотребляют наследованием. Нередко дизайн мог бы стать лучше и проще, если бы автор больше полагался на композицию объектов.

Делегирование

С помощью *делегирования* композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование [Lie86, JZ91]. При делегировании в процесс обработки запроса вовлечено *два* объекта: получатель поручает выполнение операций другому объекту - *полномоченному*. Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную-член (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, специфичные для прямоугольников. Другими словами, окно не является прямоугольником, а *содержит* его. Теперь класс *Window* может явно перенаправлять запросы своему члену *Rectangle*, а не наследовать его операции.

На диаграмме ниже изображен класс *Window*, который делегирует операцию *Area()* над своей внутренней областью 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 прямоугольник.

Главное достоинство делегирования в том, что оно упрощает композицию поведений во время выполнения. При этом способ комбинирования поведений можно изменять. Внутреннюю область окна разрешается сделать круговой 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. Заключается он в том, что динамическую, в высокой степени параметризованную программу最难 понять, нежели статическую. Есть, конечно, и некоторая потеря машинной производительности, но неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения, а не усложнения дизайна. Нелегко сформулировать правила, ясно говорящие, 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, стратегия, посетитель. В первом получатель 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 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

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 отделяет абстракцию от ее реализации. Если между абстракцией и конкретной реализацией имеется существенное сходство, то абстракция может просто делегировать операции своей реализации.

Делегирование 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. Неспецифицированные типы передаются в виде параметров 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 дают в наше распоряжение 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, мы могли бы сделать сравнение:

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

Но между тремя данными 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. С помощью параметризованных типов допустимо изменять типы, используемые классом. Но ни наследование, ни параметризованные типы не подлежат модификации во время выполнения. Выбор того или иного подхода зависит от проекта и ограничений на реализацию.

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.

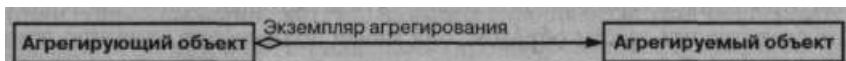
Сравнение структур времени выполнения и времени компиляции

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.

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 диаграммах осведомленность будет обозначаться сплошной линией со стрелкой. Линия со стрелкой и ромбиком вначале 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-членов, которые являются экземплярами, но чаще их определяют как указатели или ссылки. Осведомленность также реализуется с помощью указателей и ссылок.

Различие между осведомленностью и агрегированием 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 делятся лишь во время исполнения некоторой операции. Отношения осведомленности, кроме того, 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. Поведение системы во время 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 и декоратор полезны для построения сложных структур времени выполнения. Наблюдатель порождает структуры времени выполнения, которые часто трудно понять, не зная паттерна. Паттерн цепочка обязанностей также приводит к таким схемам взаимодействия, в которых наследование неочевидно. В общем можно утверждать, что разобраться в структурах времени выполнения невозможно, если не понимаешь специфики паттернов.

Проектирование с учетом будущих изменений

Системы необходимо проектировать 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:

а *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, фабричный метод, прототип;

а *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, команда;

а *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. Даже на «родной» платформе такую программу трудно поддерживать.

Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

Паттерны проектирования: абстрактная фабрика, мост;

- а *зависимость от представления или реализации объекта.* Если клиент «знает», как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений.

Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель;

- а *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, вероятность изменения которых высока, следует изолировать.

Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель;

- а *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 слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель;

- а *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 также ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов.

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, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. 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, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия;

а *неудобства при изменении классов*. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (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, декоратор, посетитель.

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

Прикладные программы

Если вы проектируете прикладную программу, 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 связанныности увеличивает вероятность того, что некоторый класс объектов сможет совместно работать с другими. Например, устранив 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 позволяют упростить сопровождение приложения, если использовать их для ограничения платформенных зависимостей и разбиения системы на отдельные слои. Они способствуют и наращиванию функций системы, показывая, как расширять иерархии классов и когда 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 предопределенных классов. Такие библиотеки называются *инструментальными*. Инструментальная библиотека - это набор взаимосвязанных, повторно используемых классов, спроектированный с целью предоставления полезных функций 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, благодаря которым в приложениях проще решать поставленные задачи, позволяют разработчику не изобретать заново повсеместно применяемые функции. Таким образом, в инструментальных библиотеках упор сделан на повторном использовании кода. Это объектно-ориентированные эквиваленты библиотек подпрограмм.

Можно утверждать, что проектирование инструментальной библиотеки сложнее, чем проектирование 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 предложений и зависимостей, способных ограничить гибкость библиотеки, 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: рисования, сочинении музыки или САПР [VL90, Joh92]. Другим каркасом рекомендуется пользоваться при создании компиляторов для разных языков программирования и целевых машин [JML92]. Третий упростит построение приложений для финансового моделирования [BE93]. Каркас можно подстроить под конкретное приложение 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, основные функции тех и других, методы взаимодействия объектов и классов и потоки управления. Данные параметры проектирования задаются каркасом, а прикладные проектировщики или разработчики могут сконцентрироваться на специфике приложения. В каркасе аккумулированы 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 меняет направление связей между приложением и программным обеспечением, лежащим в его основе, на противоположное. При использовании инструментальной библиотеки (или, если хотите, обычной библиотеки подпрограмм) вы пишете тело приложения и вызываете из него код, который планируете использовать повторно. При работе с каркасом вы, наоборот, повторно используете тело и пишете код, который оно вызывает. Вам приходится кодировать операции с предопределенными именами и параметрами вызова, но зато число принимаемых вами проектных решений сокращается.

В результате приложение создается быстрее. Более того, 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 актуальны 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 отработанных каркасах обычно можно обнаружить несколько разных паттернов проектирования. Паттерны помогают адаптировать архитектуру каркаса ко многим приложениям без повторного проектирования.

Дополнительное преимущество появляется потому, что вместе с каркасом документируются те паттерны, 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 всех типов программного обеспечения, но для каркасов этот аспект важен вдвое. Для освоения работы с каркасами надо потратить немало усилий, и только после этого они начнут приносить реальную пользу. Паттерны могут существенно упростить задачу, явно выделив 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, существует три основных различия:

а *паттерны проектирования более абстрактны, чем каркасы*. В код могут быть включены целые каркасы, но только экземпляры 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, компромиссы и последствия выбранного дизайна;

а *как архитектурные элементы, паттерны проектирования мельче, чем каркасы*. Типичный каркас содержит несколько паттернов. Обратное утверждение неверно;

а *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 разрешается использовать 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 в той же мере, что и каркасы.

Значение каркасов возрастает. Именно с их помощью объектно-ориентированные системы можно использовать повторно в максимальной степени. Крупные объектно-ориентированные приложения составляются из слоев взаимодействующих друг с другом каркасов. Дизайн и код приложения в значительной мере определяются теми каркасами, которые применялись при его создании.

1.7. Как выбирать паттерн проектирования

Если в распоряжение проектировщика предоставлен каталог из более чем 20 паттернов, трудно решать, какой паттерн лучше всего подходит для решения конкретной задачи проектирования. Ниже представлены разные подходы к выбору подходящего паттерна:

- а *подумайте, как паттерны решают проблемы проектирования.* В разделе 1.6 обсуждается то, как с помощью паттернов можно 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;
- а *пролистайте разделы каталога, описывающие назначение паттернов.* В разделе 1.4, перечислены назначения всех представленных паттернов. Ознакомьтесь с целью каждого паттерна, когда будете искать тот, что в наибольшей степени относится к вашей проблеме. Чтобы сузить поиск, воспользуйтесь схемой в таблице 1.1;
- а *изучите взаимосвязи паттернов.* На рис. 1.1 графически изображены соотношения между различными паттернами проектирования. Данная информация поможет вам найти нужный паттерн или группы паттернов;
- а *проанализируйте паттерны со сходными целями.* Каталог 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;
- а *разберитесь в причинах, вызывающих перепроектирование.* Взгляните на перечень причин, приведенный выше. Быть может, в нем упомянута ваша проблема? Затем обратитесь к изучению паттернов, помогающих устраниć эту причину;

а посмотрите, что в вашем дизайне должно быть изменяющимся. Такой подход противоположен исследованию причин, вызвавших необходимость перепроектирования. Вместо этого подумайте, что могло бы заставить изменить дизайн, а также о том, что бы вы хотели изменять без перепроектирования. Акцент здесь делается на *инкапсуляции сущностей, подверженных изменениям*, а это предмет многих паттернов. В таблице 1.2 перечислены те аспекты дизайна, которые разные паттерны позволяют варьировать независимо, устранив тем самым необходимость в перепроектировании.

1.8. Как пользоваться паттерном проектирования

Как пользоваться паттерном проектирования, который вы выбрали для изучения и работы? Вот перечень шагов, которые помогут 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 понимаете упоминаемые в паттерне классы и объекты и то, как они взаимодействуют друг с другом.
3. *Посмотрите на раздел «Пример кода», где приведен конкретный пример использования 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 употреблять их непосредственно в коде. Тем не менее бывает полезно включить имя участника как имя в программе. Это помогает сделать паттерн более очевидным при 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, которыми будут представлены данные объекты и ссылки на другие объекты. Выявите имеющиеся в вашем 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 отвечают за отношения, определенные в паттерне.* Советы о том, как это лучше сделать, вы найдете в разделе «Реализация». Поможет и «Пример кода».

Как пользоваться паттерном проектирования

Все вышесказанное - обычные рекомендации. Со временем вы выработаете собственный подход к работе с паттернами проектирования.

Таблица 1.2. Изменяемые паттернами элементы дизайна

Назначение	Паттерн проектирования	Аспекты, которые можно изменять
Порождающие паттерны	Абстрактная фабрика	Семейства порождаемых объектов
	Одиночка	Единственный экземпляр класса
	Прототип	Класс, из которого инстанцируется объект
	Строитель	Способ создания составного объекта
	Фабричный метод	Инстанцируемый подкласс объекта
Структурные паттерны	Адаптер	Интерфейс к объекту
	Декоратор	Обязанности объекта без порождения подкласса
	Заместитель	Способ доступа к объекту, его местоположение
	Компоновщик	Структура и состав объекта
	Мост	Реализация объекта
	Приспособленец	Накладные расходы на хранение объектов
	Фасад	Интерфейс к подсистеме
Паттерны поведения	Интерпретатор	Грамматика и интерпретация языка
	Итератор	Способ обхода элементов агрегата
	Команда	Время и способ выполнения запроса
	Наблюдатель	Множество объектов, зависящих от другого объекта; способ, которым зависимые объекты поддерживают себя в актуальном состоянии
	Посетитель	Операции, которые можно применить к объекту или объектам, не меняя класса
	Посредник	Объекты, взаимодействующие между собой, и способ их коопераций
	Состояние	Состояние объекта
	Стратегия	Алгоритм
	Хранитель	Закрытая информация, хранящаяся вне объекта, и время ее сохранения
	Цепочка обязанностей	Объект, выполняющий запрос
	Шаблонный метод	Шаги алгоритма

Никакое обсуждение того, как пользоваться паттернами проектирования, нельзя считать полным, если не сказать о том, как не надо их применять. Нередко за гибкость и простоту изменения, которые дают паттерны, приходится платить усложнением дизайна и ухудшением производительности. Паттерн проектирования стоит применять, только когда дополнительная гибкость действительно необходима. Для оценки достоинств и недостатков паттерна большую помощь могут оказать разделы каталога «Результаты».



Глава 2. Проектирование редактора документов

В данной главе рассматривается применение паттернов на примере проектирования визуального редактора документов Lexi¹, построенного по принципу «что видишь, то и получаешь» (WYSIWYG). Мы увидим, как с помощью паттернов можно решать проблемы проектирования, характерные для Lexi и аналогичных приложений. Здесь описывается опыт работы с восемью паттернами.

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, отформатированные разными способами. 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:

- а *структура документа*. Выбор внутреннего представления документа отражается практически на всех аспектах дизайна. Для редактирования, форматирования, отображения и анализа текста необходимо уметь обходить это представление. Способ организации информации играет решающую роль при дальнейшем проектировании;
- а *форматирование*. Как в Lexi организованы текст и графика в виде строк и колонок? Какие объекты отвечают за реализацию стратегий форматирования? Взаимодействие данных стратегий с внутренним представлением документа;
- а *создание привлекательного интерфейса пользователя*. В состав пользовательского интерфейса Lexi входят полосы прокрутки, рамки и оттененные выпадающие меню. Вполне вероятно, что количество и состав элементов интерфейса будут изменяться по мере его развития. Поэтому важно иметь возможность легко добавлять и удалять элементы оформления, не затрагивая приложение;
- а *поддержка стандартов внешнего облика программы*. Lexi должен без серьезной модификации адаптироваться к стандартам внешнего облика программ, например, таким как Motif или Presentation Manager (PM);

¹ Дизайн Lexi основан на программе Doc - текстового редактора, разработанного Кальдером [CL92].

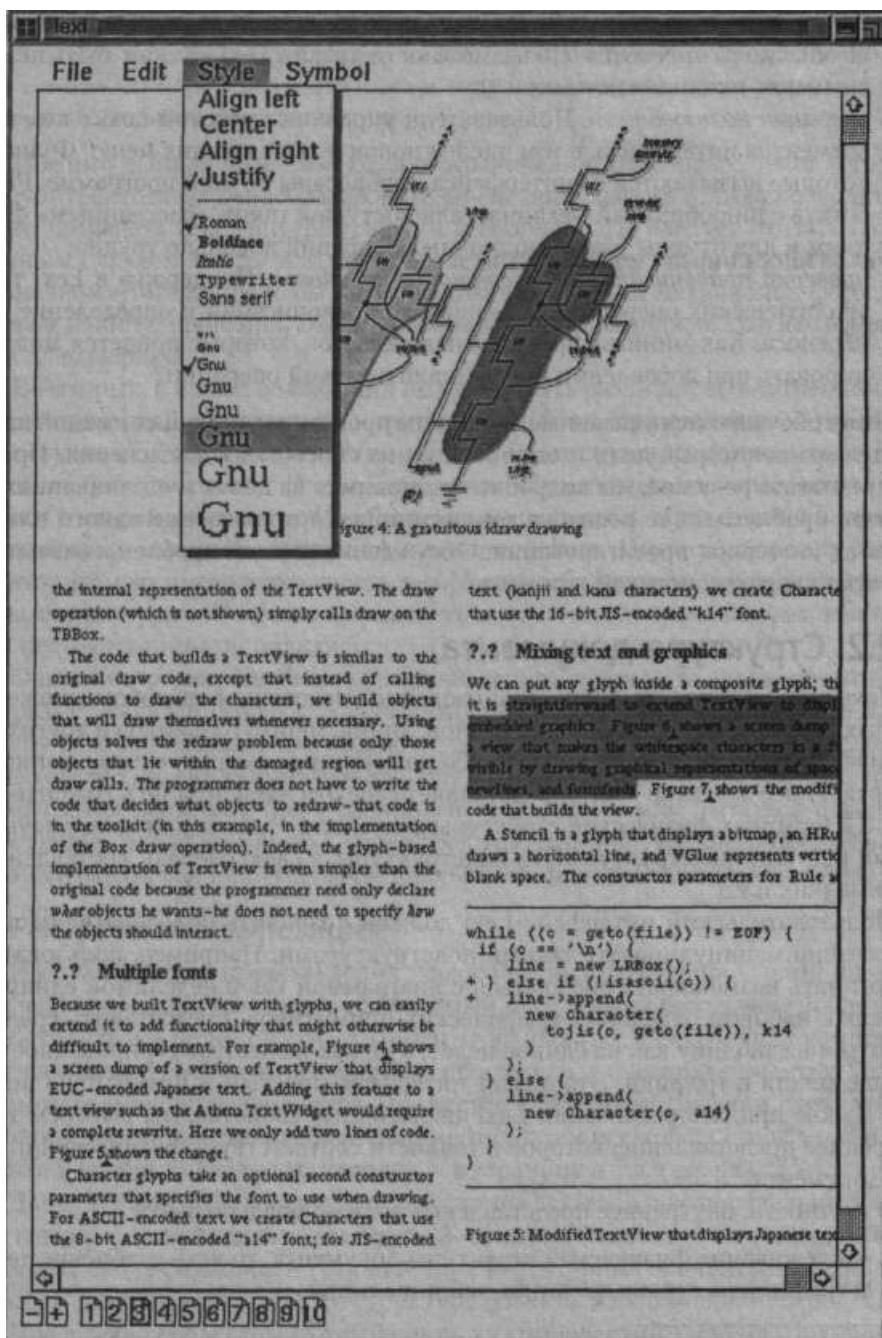


Рис. 2.1. Пользовательский интерфейс Lex/

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TextBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw—that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare what objects he wants—he does not need to specify how the objects should interact.

7.2 Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded “a14” font; for JIS-encoded

text (kanji and kana characters) we create Characters that use the 16-bit JIS-encoded “k14” font.

7.2 Mixing text and graphics

We can put any glyph inside a composite glyph; this is straightforward to extend TextView to display embedded graphics. Figure 6 shows a screen dump of a view that makes the whitespace characters in a string visible by drawing graphical representations of space, newlines, and softbreaks. Figure 7 shows the modified code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu draws a horizontal line, and VGlue represents vertical blank space. The constructor parameters for Rule are

```

while ((c = getc(file)) != EOF) {
    if (c == '\n') {
        line = new LRBox();
    } else if (!isascii(c)) {
        line->append(
            new Character(
                tojis(c, getc(file)), k14
            )
        );
    } else {
        line->append(
            new Character(c, a14)
        );
    }
}

```

Figure 5: Modified TextView that displays Japanese text.

- а *поддержка оконных систем*. В оконных системах стандарты внешнего облька обычно различаются. По возможности дизайн Lexi должен быть независимым от оконной системы;
- а *операции пользователя*. Пользователи управляют работой Lexi с помощью элементов интерфейса, в том числе кнопок и выпадающих меню. Функции которые вызываются из интерфейса, разбросаны по всей программе. Разработать единообразный механизм для доступа к таким «рассеянным» функциям и для отмены уже выполненных операций довольно трудно;
- а *проверка правописания и расстановка переносов*. Поддержка в Lexi таких аналитических операций, как проверка правописания и определение места переноса. Как минимизировать число классов, которые придется модифицировать при добавлении новой аналитической операции?

Ниже обсуждаются указанные проблемы проектирования. Для каждой из них определены 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. Все они несут в себе полную информацию о содержании документа. И все же автор 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 непосредственно манипулировать такими подструктурами. Например, пользователю следует дать возможность обращаться с диаграммой как с неделимой единицей, а не как с набором отдельных графических примитивов, предоставить средства ссылаться на таблицу как на единое целое, а не как на неструктурированное хранилище текста и графики. Это делает интерфейс простым и интуитивно понятным. Чтобы придать реализации Lexi аналогичные свойства, мы выберем такое внутреннее представление, которое в точности соответствует физической структуре документа.

В частности, внутреннее представление должно поддерживать:

- а отслеживание физической структуры документа, то есть разбиение текста и графики на строки, колонки, таблицы и т.д.;

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

а генерирование визуального представления документа;
а отображение позиций экрана на элементы внутреннего представления. Это позволит определить, что имел в виду пользователь, когда указал на что-то в визуальном представлении.

Помимо данных целей имеются и ограничения. Во-первых, текст и графику следует трактовать единообразно. Интерфейс приложения должен позволять свободно помещать текст внутрь графики и наоборот. Не следует считать графику частным случаем текста или текст - частным случаем графики, поскольку это в конечном итоге привело бы к появлению избыточных механизмов форматирования и манипулирования. Одного набора механизмов должно хватить и для текста, и для графики.

Во-вторых, в нашей реализации не может быть различий во внутреннем представлении отдельного элемента и группы элементов. При одинаковой работе Lexi с простыми и сложными элементами можно будет создавать документы со структурой любой сложности. Например, десятым элементом на пересечении пятой строки и второй колонки мог бы быть как один символ, так и сложно устроенная диаграмма со многими внутренними компонентами. Но, коль скоро мы уверены, что этот элемент имеет возможность изображать себя на экране и сообщать свои размеры, его внутренняя сложность не имеет никакого отношения к тому, как и в каком месте страницы он появляется.

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 вид анализа зависит от анализируемого объекта. Так, вряд ли имеет смысл проверять орфографию многоугольника или пытаться переносить его с одной строки на другую. При проектировании внутреннего представлении надо учитывать эти и другие потенциально конфликтующие ограничения.

Рекурсивная композиция

На практике для представления иерархически структурированной информации часто применяется прием, называемый *рекурсивной композицией*. Он позволяет строить все более сложные элементы из простых. Рекурсивная композиция дает нам способ 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. (см. рис. 2.2).

Данную физическую структуру можно представить, введя 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 графический элемент документа, мы обеспечиваем гибкость на самых низких уровнях дизайна Lexi. С точки зрения отображения, форматирования и вкладывания друг в друга единообразно трактуются текст и графика. Мы сможем расширить Lexi для поддержки новых наборов

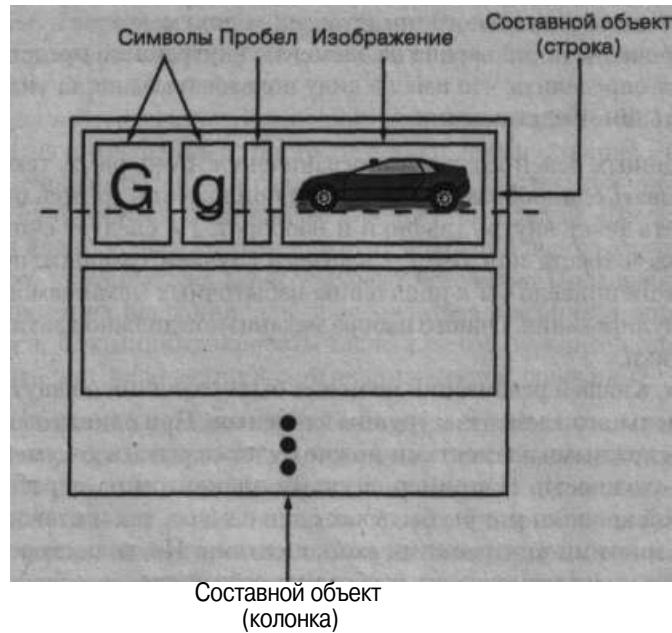


Рис. 2.2. Рекурсивная композиция текста и графики

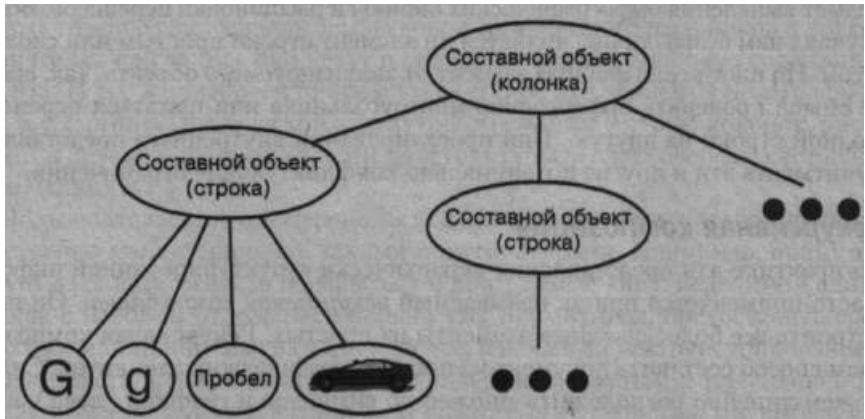


Рис. 2.3. Структура объекта для рекурсивной композиции текста и графики

символов, не затрагивая никаких других функций. Объектная структура Lexi точно отражает физическую структуру документа.

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 документа¹. 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 таблице 2.1 более подробно представлен базовый интерфейс этого класса в нотации C++².

Таблица 2. 1. Базовый интерфейс класса *Glyph*

Обязанность	Операции
внешнее представление	virtual void Draw(Window*)
обнаружение точки воздействия	virtual void Bounds(Rect&)
структура	virtual bool intersects(const Point&) virtual void Insert(Glyph*, int) virtual void Remove(Glyph*) virtual Glyph* Child(int) virtual Glyph* Parent()

У глифов есть три основные функции. Они имеют информацию о своих предках и потомках, а также о том, как нарисовать себя на экране и сколько места они занимают.

Подклассы класса *Glyph* переопределяют операцию *Draw*, выполняющую перерисовку себя в окне. При вызове *Draw* ей передается ссылка на объект *Window*. В классе *Window* определены графические операции для прорисовки в окне на экране текста и основных геометрических фигур. Например, в подклассе *Rectangle* операция *Draw* могла определяться так:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect (_x0,
```

где *_x0*, *_y0*, *_x1* и *_y1* - это данные-члены класса *Rectangle*, определяющие два противоположных угла прямоугольника, а *DrawRect* - операция из класса *Window*, рисующая на экране прямоугольник.

Впервые термин «глиф» в этом контексте употребил Пол Кальдер [CL90]. В большинстве современных редакторов документов отдельные символы не представляются объектами, скорее всего, изображений эффективности. Кальдер продемонстрировал практическую пригодность этого подхода в своей диссертации [Ca193]. Наши глифы проще предложенных им, поскольку мы для простоты ограничились строгими иерархиями. Глифы Кальдера могут использоваться совместно для уменьшения потребления памяти и, стало быть, образуют направленные ациклические графы. Для достижения того же эффекта мы можем воспользоваться паттерном Приспособленец, но оставим это в качестве упражнения читателю. Представленный здесь интерфейс намеренно сделан минимальным, чтобы не загромождать обсуждение техническими деталями. Полный интерфейс должен был включать операции для работы с графическими атрибутами: цветами, шрифтами и преобразованиями координат, а также операции для более развитого управления потомками.

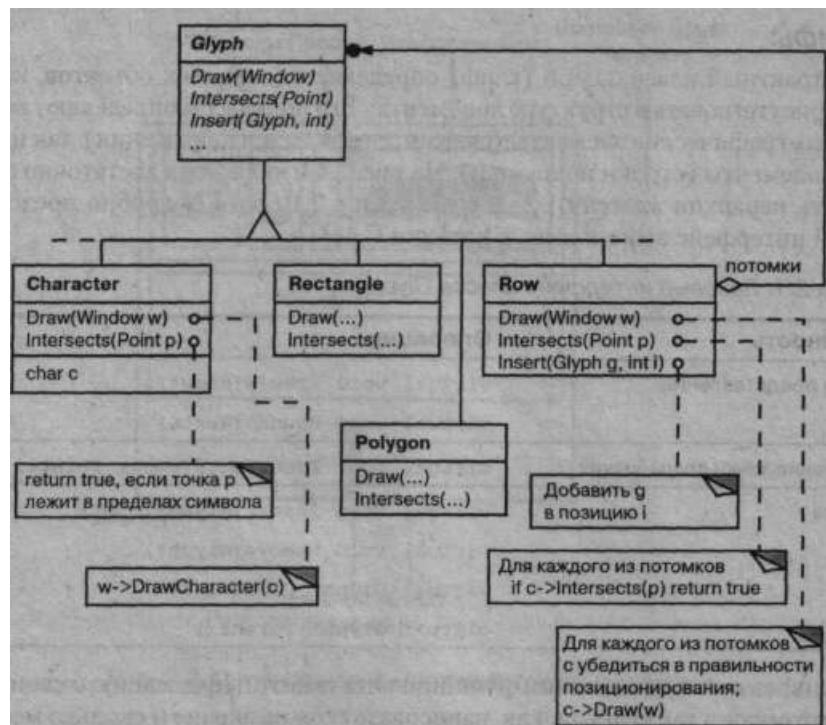


Рис. 2.4. Частичная иерархия класса `Glyph`

Глифу-родителю часто бывает нужно «знать», сколько места на экране занимает глиф-потомок, чтобы расположить его и остальные глифы в строке без перекрытий (как показано на рис. 2.2). Операция `Bounds` возвращает прямоугольную область, занимаемую глифом, точнее, противоположные углы наименьшего прямоугольника, содержащего глиф. В подклассах класса `Glyph` эта операция переопределена в соответствии с природой конкретного элемента.

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 оказалась под курсором 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, потомки класса `Row` – это 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.

Возможно, целочисленный индекс – не лучший способ описания потомков глифа. Это зависит от структуры данных, используемой внутри глифа. Если потомки хранятся в связанным списке, то более эффективно было бы передавать указатель на элемент списка. Мы еще увидим более удачное решение проблемы индексации в разделе 2.8, когда будем обсуждать анализ документа.

Операция `Child` возвращает потомка с заданным индексом (если таковой существует). Глифы типа `Row`, у которых действительно есть потомки, 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 обходят 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 `Parent` просто возвращает эту ссылку.

Паттерн компоновщик

Рекурсивная композиция пригодна не только для документов. Мы можем воспользоваться ей для представления любых потенциально сложных иерархических структур. Паттерн компоновщик инкапсулирует сущность рекурсивной композиции объектно-ориентированным способом. Сейчас самое время обратиться к разделу об этом паттерне и изучить его, имея в виду, только что рассмотренный сценарий.

2.3. Форматирование

Мы решили, 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. Описание внутренней структуры не дает возможности добраться до определенной подструктуры. Ответственность за это лежит в основном на `Lexi`. Редактор разбивает текст на строки, строки - на колонки и т.д., учитывая при этом пожелания пользователя. Так, пользователь может изменить ширину полей, размер отступа и положение точек табуляции, установить одиночный или двойной межстрочный интервал, а также задать много других параметров форматирования.¹ В процессе форматирования это учитывается.

Кстати говоря, мы сузим значение термина «форматирование», понимая под этим лишь разбиение на строки. Будем считать слова «форматирование» и «разбиение на строки взаимозаменяемыми». Обсуждаемая техника в равной мере применима и к разбиению строк на колонки, и к разбиению колонок на страницы.

Таблица 2.2. Базовый интерфейс класса `Compositor`

Обязанность	Операции
что форматировать	<code>void SetComposition(Composition*)</code>
когда форматировать	<code>virtual void Compose()</code>

У пользователя найдется что сказать и по поводу *логической* структуры документа: предложений, абзацев, разделов, глав и т.д. *Физическая* структура в общем-то менее интересна. Большинству людей не важно, где в абзаце произошел разрыв строки, если в целом все отформатировано правильно. То же самое относится и к формированию колонок и страниц. Таким образом, пользователи задают только высокоуровневые ограничения на физическую структуру, а `Lexi` выполняет их.

Инкапсуляция алгоритма форматирования

С учетом всех ограничений и деталей процесс форматирования с трудом поддается автоматизации. К этой проблеме есть много подходов, и у разных алгоритмов форматирования имеются свои сильные и слабые стороны. Поскольку Lexi - это WYSIWYG-редактор, важно соблюдать компромисс 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 удается установить на этапе компиляции. Например, можно предположить, что пользователь готов смириться с замедленной реакцией в обмен на лучшее качество форматирования. При таком предположении нужно было бы применять совершенно другой алгоритм форматирования. Есть также компромисс между временем и памятью, скорее, имеющий отношение к реализации: время форматирования можно уменьшить, если хранить в памяти больше информации.

Поскольку алгоритмы форматирования обычно оказываются весьма сложными, желательно, чтобы они были достаточно замкнутыми, а еще лучше - полностью независимыми от структуры документа. Оптимальный вариант - добавление нового вида глифа 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 легко можно было заменить, по крайней мере, на этапе компиляции, если уж не во время выполнения. Допустимо изолировать алгоритм и одновременно сделать его легко замещаемым путем инкапсулирования в объекте. Точнее, мы определим отдельную иерархию классов для объектов, инкапсулирующих алгоритмы форматирования. Для корневого класса иерархии определим 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 *Composer* и *Composition*

Мы определим класс *Composer* (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 *Composer* (специализированный для конкретного алгоритма разбиения на строки) и в нужные моменты предписывает композитору форматировать глифы, по мере того как пользователь изменяет документ. На рис. 2.5 изображены отношения между классами *Composition* и *Composer*.

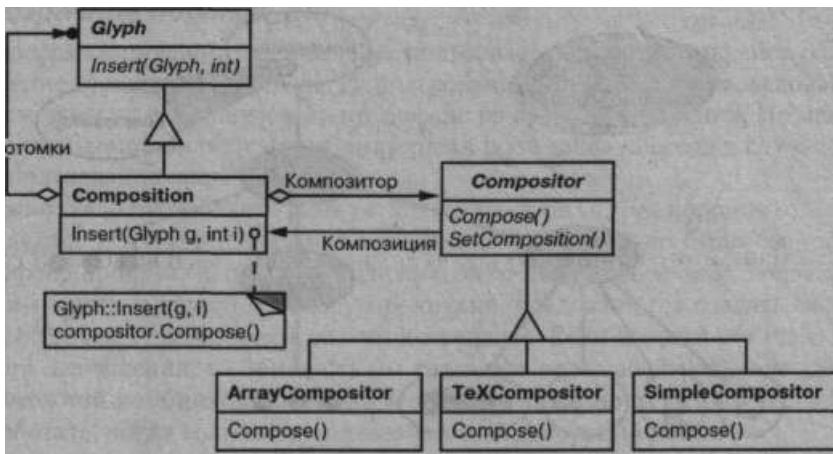


Рис. 2.5. Отношения классов *Composition* и *Compositor*

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, заштрихованы на рисунке серым цветом.

Каждый подкласс класса *Compositor* может реализовывать свой собственный алгоритм форматирования. Например, класс *SimpleCompositor* мог бы осуществлять быстрый проход, не обращая внимания на такую экзотику, как «цвет» документа. Под «хорошим цветом» понимается равномерное распределение текста и пустого пространства. Класс *TeXCompositor* мог бы реализовывать полный алгоритм *TjX*[Knu84], учитывающий наряду со многими другими вещами и цвет, но за счет увеличения времени форматирования.

Наличие классов *Compositor* и *Composition* обеспечивает четкое отделение кода, поддерживающего физическую структуру документа, от кода различных алгоритмов форматирования. Мы можем добавить новые подклассы к классу *Compositor*, не трогая классов глифов, и наоборот. Фактически допустимо подменить алгоритм разбиения на строки во время выполнения, добавив одну-единственную операцию *SetCompositor* к базовому интерфейсу класса *Composition*.

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

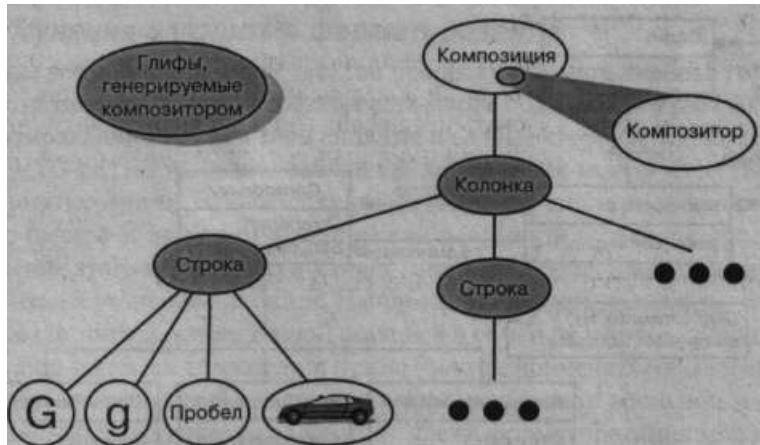


Рис. 2.6. Объектная структура, отражающая алгоритм разбиения на строки, выбираемый композитором

Стратегия

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

Ключ к применению паттерна стратегия – спроектировать интерфейсы стратегии и контекста, достаточно общие для поддержки широкого диапазона алгоритмов. Не должно возникать необходимости изменять интерфейс стратегии или контекста для поддержки нового алгоритма. В нашем примере поддержка доступа к потомкам, их вставки и удаления, предоставляемая базовыми интерфейсами класса Glyph, достаточно общая, чтобы подклассы класса Compositor могли изменять физическую структуру документа независимо от того, с помощью каких алгоритмов это делается. Аналогично интерфейс класса Compositor дает композициям все, что им необходимо для инициализации форматирования.

2.4. Оформление пользовательского интерфейса

Рассмотрим два усовершенствования пользовательского интерфейса Lexi. Первое добавляет рамку вокруг области редактирования текста, чтобы четко обозначить страницу текста, второе – полосы прокрутки, позволяющие пользователю просматривать разные части страницы. Чтобы упростить добавление и удаление таких элементов оформления (особенно во время выполнения), мы не должны использовать наследование. Максимальной гибкости можно достичь, если другим объектам пользовательского интерфейса даже не будет известно о том, какие еще есть элементы оформления. Это позволит добавлять и удалять декорации, не изменяя других классов.



Прозрачное обрамление

В программировании оформление пользовательского интерфейса означает расширение существующего кода. Использование 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 элементов оформления. Это решение быстро перестает работать, когда количество разнообразных декораций растет.

С помощью композиции объектов можно найти куда более приемлемый и гибкий механизм расширения. Но из каких объектов составлять 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. Следующий шаг - решить, что является агрегатом, а что - компонентом. Допустимо считать, что рамка содержит глиф, и это имеет смысл, 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 наличии 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 настоятельные причины поступить именно таким образом: клиентов не должно волновать, есть у глифов рамки или нет. Все глифы должны трактоваться единообразно. Когда клиент сообщает простому глифу без рамки о необходимости нарисовать себя, тот делает это, не добавляя никаких элементов оформления. Если же этот глиф заключен в рамку, то клиент не должен обрабатывать рамку как-то специально; он просто предписывает составному глифу выполнить отображение 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 обрамления* (*transparent enclosure*), где комбинируются понятия о композиции с одним потомком (однокомпонентные), и о совместимых интерфейсах. В общем случае клиенту неизвестно, имеет ли он дело с компонентом или его *обрамлением* (то есть родителем), особенно если обрамление просто делегирует все операции своему единственному компоненту. Но обрамление может также и расширять поведение компонента, выполняя дополнительные действия либо до, либо после делегирования (а возможно, и до, и после). Обрамление может также добавить компоненту состояние. Как именно, будет показано ниже.

Моноглиф

Концепцию прозрачного обрамления можно применить ко всем глифам, оформляющим другие глифы. Чтобы конкретизировать эту идею, определим подкласс класса `Glyph`, называемый `MonoGlyph`. Он будет выступать в роли абстрактного 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);
}
```

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, вызывая свою собственную закрытую операцию `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

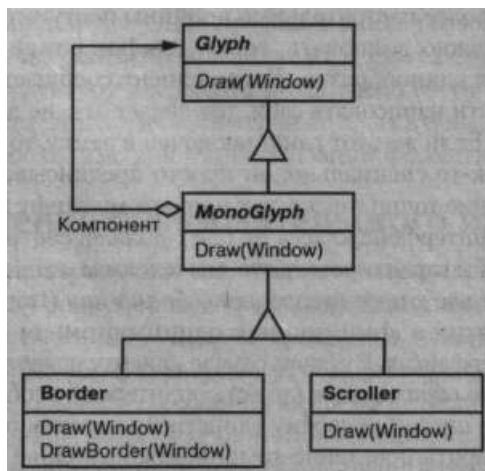


Рис. 2.7. Отношения класса `MonoGlyph` с другими классами

положения двух полос прокрутки, добавляющихся в качестве элементов оформления. Когда `Scroller` отображает свой компонент, графическая система обретает 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 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.

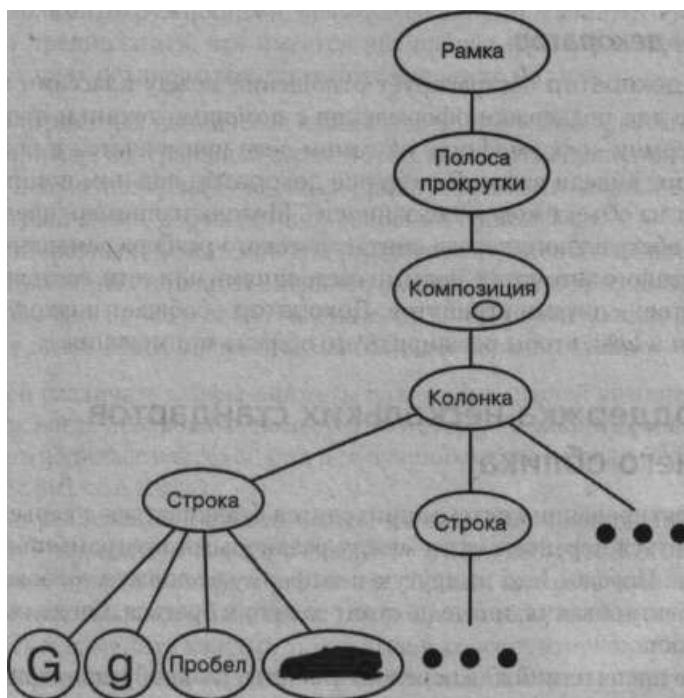


Рис. 2.8. Объектная структура после добавления элементов оформления

Обратите внимание, что мы могли изменить порядок композиции на обратный, сначала добавив рамку, а потом погрузив результат в `Scroller`. В таком случае рамка прокручивалась бы вместе с текстом. Может быть, это то, что вам нужно, а может, и нет. Важно лишь, что прозрачное обрамление легко позволяет экспериментировать с разными вариантами, освобождая клиента от знания деталей кода, добавляющего декорации.

Отметим, что рамка допускает композицию только с одним глифом, не более того. 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 дает ничего нового, 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 избавлены от необходимости дублировать уже имеющуюся функциональность.

Паттерн декоратор

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

2.5. Поддержка нескольких стандартов внешнего облика

При проектировании системы приходится сталкиваться с серьезной проблемой: как добиться переносимости между различными программно-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 внешнего облика, 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, написанные для Motif, выглядят не совсем так, как аналогичные приложения на других платформах, и наоборот. Программа, работающая более чем на одной платформе, должна всюду соответствовать принятой стилистике пользовательского интерфейса.

Наша проектная цель - сделать так, чтобы Lexi поддерживал разные стандарты внешнего облика и чтобы легко можно было добавить поддержку нового стандарта, как только таковой появится (а это неизбежно произойдет). Хотелось бы также, чтобы наш дизайн решал и другую задачу: изменение внешнего облика 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 содержат видимые - скажем, кнопку (Button) или символ (Character) - и правильно располагают их на экране. В руководствах по стилистическому оформлению 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.

Мы будем предполагать, что имеется два набора классов глифов-виджетов, с помощью которых реализуются стандарты внешнего облика:

- а 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;
- а 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 инстанцировать подкласс класса `Glyph` для нужного стиля кнопки (`MotifButton`, `PMBUTTON`, `MacButton` и т.д.).

Ясно, что в реализации Lexi это нельзя сделать непосредственно, например, вызвав конструктор, если речь идет о языке C++. При этом была бы жестко закодирована кнопка одного конкретного стиля, 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 внешнего облика вызывает существенные неудобства при сопровождении. Стоит что-нибудь пропустить - и в приложении, ориентированном на платформу Mac, появится меню в стиле Motif.

Lexi необходим какой-то способ определить нужный стандарт внешнего облика 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*. Далее на примере мы продемонстрируем, что имеется в виду.

Фабрики и изготовленные классы

В Motif для создания экземпляра глифа полосы прокрутки обычно было достаточно написать следующий код на C++:

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

Но такого кода надо избегать, если мы хотим минимизировать зависимость Lexi от стандарта внешнего облика. Предположим, однако, что sb инициализируется так:

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

где guiFactory - CreateScrollBar объект класса MotifFactory. Операция возвращает новый экземпляр подходящего подкласса ScrollBar, который соответствует желательному варианту внешнего облика, в нашем случае - Motif. С точки зрения клиентов результат тот же самый, что и при прямом обращении к конструктору MotifScrollBar. Но есть и существенное отличие: нигде в коде⁴ больше не упоминается имя Motif. Объект guiFactory абстрагирует процесс создания не только полос прокрутки для Motif, но и любых других. Более того, guiFactory не ограничен изготовлением только полос прокрутки. Его можно применять для производства любых виджетов, включая кнопки, поля ввода, меню и т.д.

Все это стало возможным, поскольку MotifFactory является подклассом GUIFactory - абстрактного класса, который определяет общий интерфейс для создания глифов-виджетов. В нем есть такие операции, как CreateScrollBar и CreateButton, для инстанцирования различных видов виджетов. Подклассы GUIFactory реализуют эти операции, возвращая глифы вроде MotifScrollBar и PMButton, которые имеют нужный внешний облик и поведение. На рис. 2.9 показана иерархия классов для объектов guiFactory.

Мы говорим, что фабрики *изготавливают* объекты. Продукты, изготовленные фабриками, 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.

Экземпляр класса GUIFactory может быть создан любым способом. Переменная guiFactory может быть глобальной или статическим членом хорошо известного класса или даже локальной, если весь пользовательский интерфейс создается внутри одного класса или функции. Существует специальный паттерн проектирования одиничка, 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, какой внешний облик нужен.

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

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

Если же пользователю разрешается задавать внешний облик с помощью строки-параметра при запуске, то код создания фабрики мог бы выглядеть так:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
// получаем это от пользователя или среды при запуске
```

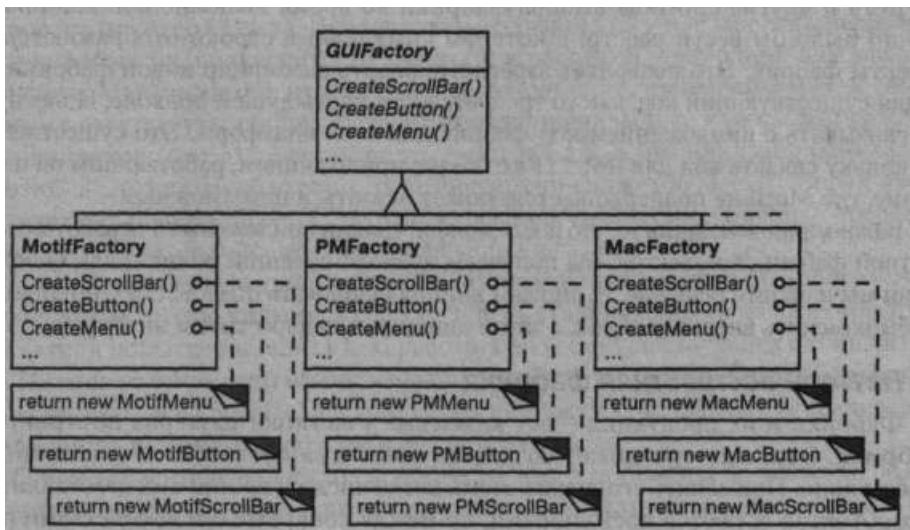
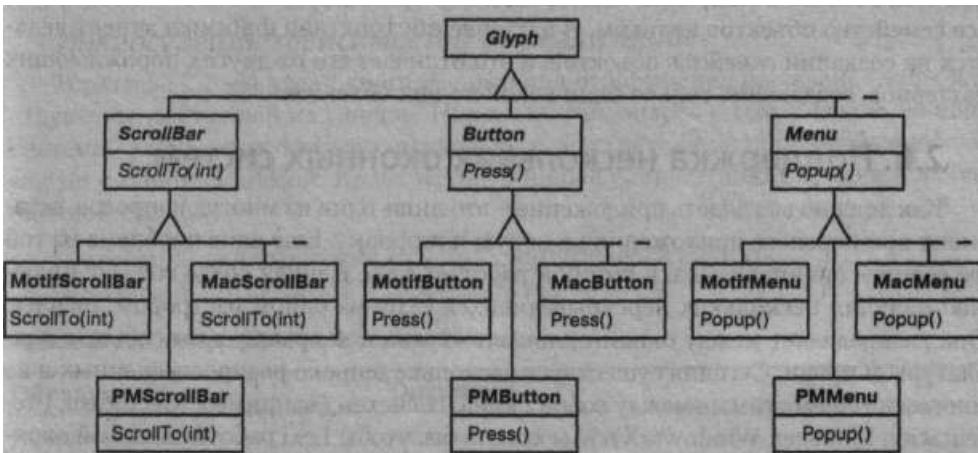
Рис. 2.9. Иерархия классов *GUIFactory*

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

```

if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;

} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;

} else {
    guiFactory = new DefaultGUIFactory;
}
    
```

Есть и другие способы выбора фабрики во время выполнения. Например, можно было бы вести реестр, в котором символьные строки отображаются на объекты фабрик. Это позволяет зарегистрировать экземпляр новой фабрики, не меняя существующий код, как то требуется при предыдущем подходе. И нет нужды связывать с приложением код фабрик для всех платформ. Это существенно, поскольку связать код для Motif Factory с приложением, работающим на платформе, где Motif не поддерживается, может оказаться невозможным.

Важно, впрочем, лишь то, что после конфигурации приложения для работы с конкретной фабрикой объектов, мы получаем нужный внешний облик. Если впоследствии мы изменим решение, то сможем инициализировать guiFactory по-другому, чтобы изменить внешний облик, а затем динамически перестроим интерфейс.

Паттерн абстрактная фабрика

Фабрики и их продукция - вот ключевые участники паттерна абстрактная фабрика. Этот паттерн может создавать семейства объектов, не инстанцируя классы явно. Применять его лучше всего, когда число и общий вид изготавливаемых объектов остаются постоянными, но между конкретными семействами продуктов имеются различия. Выбор того или иного семейства осуществляется путем инстанцирования конкретной фабрики, после чего она используется для создания всех объектов. Подставив вместо одной фабрики другую, мы можем 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, вставших при переносе приложения на новую 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 внешнего облика.

Можно ли воспользоваться абстрактной фабрикой?

На первый взгляд представляется, что перед нами еще одна возможность применить паттерн абстрактная фабрика. Но ограничения, связанные с переносом на другие оконные системы, существенно отличаются от тех, что накладывают независимость от внешнего облика.

Применяя паттерн абстрактная фабрика, мы предполагали, что удастся определить конкретный класс глифов-виджетов для каждого стандарта внешнего облика. Это означало, что можно будет произвести конкретный класс для данного

стандарта (например, `Mouse ScrollBar` и `Mac ScrollBar`) от абстрактного класса (допустим, `ScrollBar`). Предположим, однако, что у нас уже есть несколько иерархий классов, полученных от разных поставщиков, - по одной для каждого стандарта. Маловероятно, что данные иерархии будут совместимы между собой. Поэтому отсутствуют общие абстрактные изготавливаемые классы 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 мы решили эту проблему, разработав собственные абстрактные и конкретные изготавливаемые классы. Теперь сталкиваемся с аналогичной трудностью при попытке заставить Lexi работать во всех существующих оконных средах. 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 на внешний облик, 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 можно закрыть любую конкретную реализацию оконной системы.

Инкапсуляция зависимостей от реализации

В разделе 2.2 мы ввели 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 понятие окна в любой оконной системе:

- а операции для отрисовки базовых геометрических фигур;
- а возможность свернуть и развернуть окно;

Таблица 2.3. Интерфейс класса `Window`

Обязанность	Операции
управление окнами	<code>virtual void Redraw()</code> <code>virtual void Raise()</code> <code>virtual void Lower()</code> <code>virtual void Iconify()</code> <code>virtual void Deiconify()</code>
графика	<code>virtual void DrawLine(...)</code> <code>virtual void DrawRect(...)</code> <code>virtual void DrawPolygon(...)</code> <code>virtual void DrawText(...)</code>

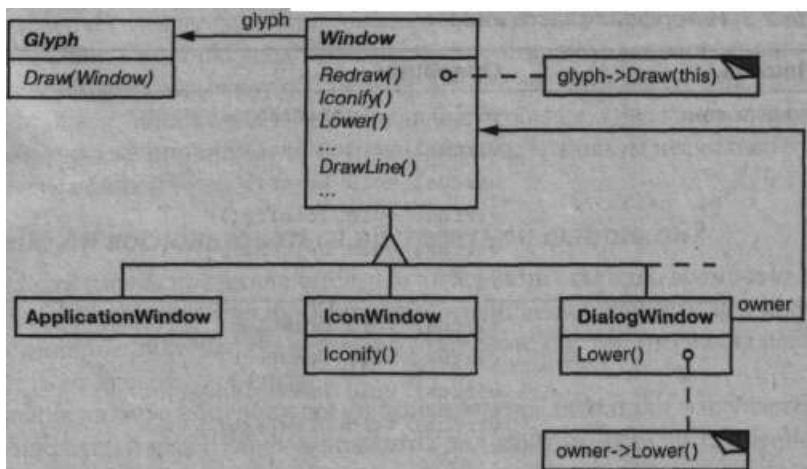
- а возможность изменить собственные размеры;
- а возможность при необходимости отобразить свое содержимое, например при развертывании или открытии ранее перекрытой части окна.

Класс Window должен покрывать функциональность окон, которая имеется в различных оконных системах. Рассмотрим два крайних подхода:

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

Ни то, ни другое решение «в чистом виде» не годятся, 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 перечислены некоторые операции из интерфейса класса Window.

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 платформенно-зависимой 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?

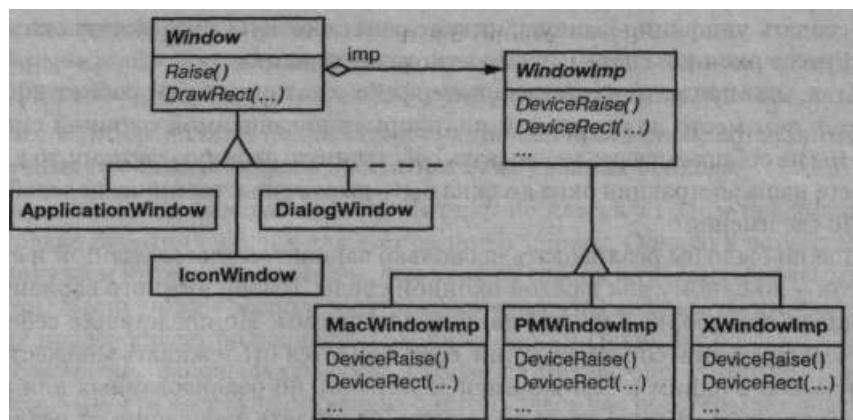
Можно было бы реализовать несколько вариантов класса `Window` и его подклассов - по одному для каждой оконной среды. Выбор нужного варианта производится при сборке Lexi для данной платформы. Но представьте себе, с чем вы столкнетесь при сопровождении, если придется отслеживать множество разных классов с одним и тем же именем `Window`, но реализованных для разных оконных систем. Вместо этого мы могли бы создать зависящие от реализации подклассы каждого класса в иерархии `Window`, но закончилось бы это тем же самым комбинаторным ростом числа классов, о котором уже говорилось при попытке добавить элементы оформления. Кроме того, оба решения недостаточно гибки, 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 выглядят привлекательно, но что еще можно сделать? То же самое, что мы сделали для форматирования и декорирования, - *инкапсулировать изменяющуюся сущность*. В этом случае изменчивой частью является реализация оконной системы. Если инкапсулировать функциональность оконной системы в объекте, то удастся реализовать свой класс `Window` и его подклассы в терминах интерфейса этого объекта. Более того, если такой интерфейс сможет поддержать все интересующие нас оконные системы, то не придется изменять ни `Window`, ни его подклассы при переходе на другую систему. Мы сконфигурируем оконные объекты в соответствии с требованиями нужной оконной системы, просто передав 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 системно-зависимый код. Чтобы заставить Lexi работать в конкретной оконной системе, каждый оконный объект будем конфигурировать экземпляром того подкласса `WindowImp`, который предназначен для этой системы. На диаграмме 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 терминах операции *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-член класса *Window*, в которой хранится указатель на объект *WindowImp*, использованный при конфигурировании *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 определено именно так, поскольку XDrawRectangle (функция X Windows для рисования прямоугольников) определяет прямоугольник с помощью левого нижнего угла, ширины и высоты. DeviceRect должна вычислить эти значения по переданным ей параметрам. Сначала находится левый нижний угол (поскольку (xO, yO) может обозначать любой из четырех углов прямоугольника), а затем вычисляется длина и ширина.

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

```
void PMWindowImp::DeviceRect(
    Coord xO, Coord yO, Coord xl, Coord yl

    Coord left = min(xO, xl);
    Coord right = max(xO, xl);
    Coord bottom = min(yO, yl);
    Coord top = max(yO, yl);

    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(Jips) == false)

        // сообщить об ошибке

    } else {
        GpiStrokePath(_hps, 1L, OL);
    }
}
```

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 деталях оконной системы. Заодно мы получаем возможность добавлять поддержку для новых оконных систем, не изменяя классы из иерархии Window.

Конфигурирование класса *Window* с помощью *WindowImp*

Важнейший вопрос, который мы еще не рассмотрели, - как сконфигурировать окно с помощью подходящего подкласса *WindowImp*. Другими словами, когда инициализируется переменная *_imp* и как узнать, какая оконная система (следовательно, и подкласс *WindowImp*) используется? Ведь окну необходим объект *WindowImp*.

Тут есть несколько возможностей, но мы остановимся на той, где используется паттерн абстрактная фабрика. Можно определить абстрактный фабричный класс *WindowSystemFactory*, предоставляющий интерфейс для создания различных видов системно-зависимых объектов:

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

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

Далее разумно 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; }
    // ...
};
```

Чтобы инициализировать член *_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*. Она, аналогично переменной *guiFactory*, определяет внешний облик. И инициализировать *WindowSystemFactory* можно точно так же.

Паттерн 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 ближе к идеи пересечения или объединения функциональности - в зависимости от требований к целевой оконной системе.

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 нам 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 является собой пример паттерна мост. Идея его создания заключалась в том, чтобы предоставить возможность совместной работы отдельным иерархиям классов, даже в случае их раздельного эволюционирования. Критерии разработки, которыми мы руководствовались, заставили нас создать две различные иерархии классов: одну, поддерживающую логическое понятие окон, и другую для хранения промежуточных вариантов окон. Паттерн мост позволяет нам сохранять и совершенствовать наши логические абстракции управления окнами без необходимости привлечения программно-зависимого кода и наоборот.

2.7. Операции пользователя

Часть функциональности Lexi доступна через WYSIWYG-представление документа. Вы вводите и удаляете текст, перемещаете точку вставки и выбираете участки текста, просто указывая и щелкая мышью или нажимая клавиши. Другая часть функциональности доступна через выпадающие меню, кнопки и клавиши-ускорители. К этой категории относятся такие операции:

- а создание нового документа;
- а открытие, сохранение и печать существующего документа;
- а вырезание выбранной части документа и вставка ее в другое место;
- а изменение шрифта и стиля выбранного текста;
- а изменение форматирования текста, например, установка режима выравнивания;
- а завершение 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 с помощью кнопки или выбора из меню). Кроме того, в будущем может понадобиться изменить интерфейс.

Далее. Операции реализованы в разных классах. Нам как разработчикам хотелось бы получать доступ к функциональности классов, не создавая зависимостей между классами, отвечающими за реализацию и за пользовательский интерфейс. В противном случае получится сильно связанный код, который будет трудно понять, модернизировать и сопровождать.

Ситуация осложняется еще и тем, что Lexi должен поддерживать операции отмены и повтора¹ большинства, но *не всех* функций. 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.

Ясно, что 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 вложенных глифов. От других глифов, имеющих потомков, его отличает 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 клиента². Для выполнения запроса может потребоваться вызвать одну операцию одного объекта или много операций разных объектов. Возможны и промежуточные варианты.

Мы могли бы определить подкласс класса `MenuItem` для каждой операции пользователя, а затем жестко закодировать каждый подкласс для выполнения соответствующего запроса. Но вряд ли это правильно: нам не нужен подкласс `MenuItem` для каждого запроса, точно так же, как не нужен отдельный подкласс для каждой строки в выпадающем меню. Помимо всего прочего, такой подход тесно привязывает запрос к конкретному элементу пользовательского интерфейса, поэтому выполнить тот же запрос через другой интерфейс будет нелегко.

Предположим, что на последнюю страницу документа вы можете перейти, выбрав соответствующий пункт из меню или щелкнув по значку с изображением страницы в нижней части окна Lexi (для коротких документов это удобно). Если мы ассоциируем запрос с подклассом `MenuItem` с помощью наследования, то должны сделать то же самое и для значка страницы, и для любого другого виджета, который способен послать подобный запрос. В результате число классов будет примерно равно произведению числа типов виджетов на число запросов.

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

² Концептуально клиентом является пользователь Lexi, но на самом деле это просто какой-то другой объект (например, диспетчер событий), который управляет обработкой ввода пользователя.

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. Параметризовать MenuItem можно вызываемой функцией, но это решение неполно по трем причинам:

- а в нем не учитывается проблема отмены/повтора;
- а с функцией трудно ассоциировать состояние. Например, функция, изменяющая шрифт, должна «знать», какой именно это шрифт;
- а функции с трудом поддаются расширению, а использовать их части тоже затруднительно.

Поэтому лучше параметризовать пункты меню *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 возможность реализовать отмену и повтор. Вот 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 запросов. Некоторые подклассы могут частично или полностью делегировать работу другим объектам, а остальные выполняют запрос сами (см. рис. 2.11). Однако для запрашивающего объект Command — это всего лишь объект Command, все они рассматриваются одинаково.

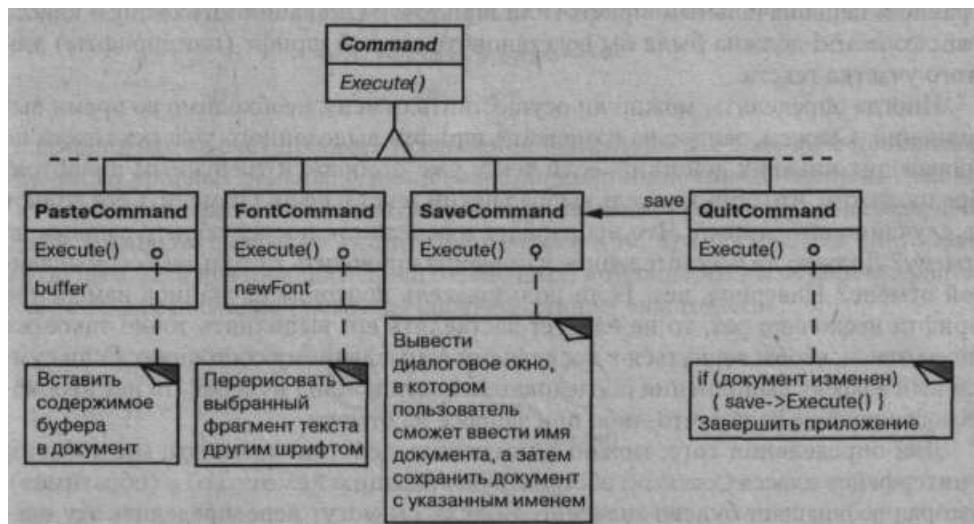


Рис. 2.11. Часть иерархии класса Command

Теперь в классе MenuItem может храниться объект, инкапсулирующий запрос (рис. 2.12). Каждому объекту, представляющему пункт меню, мы передаем 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, тем самым предлагая ему выполнить запрос. Заметим, что кнопки и другие виджеты могут пользоваться объектами Command точно так же, как и пункты меню.

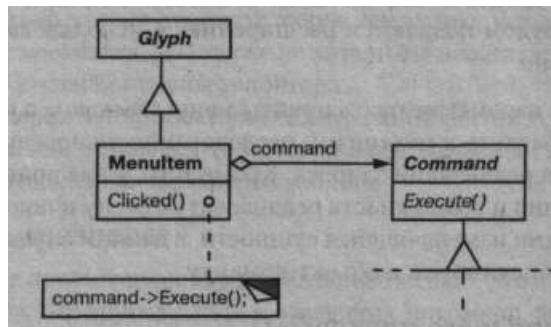


Рис. 2.12. Отношение между классами MenuItem и Command

Отмена операций

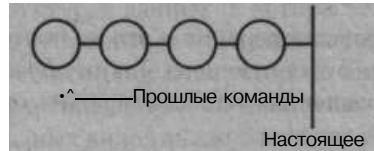
Для того чтобы отменить или повторить команду, нужна операция Unexecute в интерфейсе класса Command. Ее выполнение отменяет все, что было сделано предыдущей командой Execute. При этом используется информация, сохраненная операцией Execute. Так, при команде FontCommand операция Execute была бы должна сохранить координаты участка текста, шрифт которого изменялся, а равно и первоначальный шрифт (или шрифты). Операция Unexecute класса FontCommand должна была бы восстановить старый шрифт (или шрифты) для этого участка текста.

Иногда определять, можно ли осуществить отмену, необходимо 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.

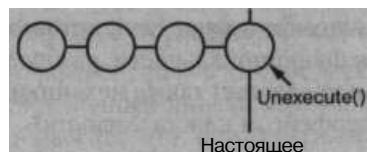
История команд

Последний шаг в направлении поддержки отмены и повтора с произвольным числом уровней - определение *истории команд*, то есть списка ранее выполненных или отмененных команд. Концептуально история команд выглядит так:



Каждый кружок представляет один объект Command. В данном случае пользователь выполнил четыре команды. Линия с пометкой «настоящее» показывает самую последнюю выполненную (или отмененную) команду.

Для того чтобы отменить последнюю команду, мы просто вызываем операцию Unexecute для самой последней команды:

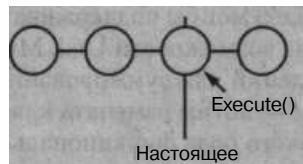


После отмены команды сдвигаем линию «настоящее» на одну команду влево. Если пользователь выполнит еще одну отмену, то произойдет откат на один шаг (см. рис. ниже).

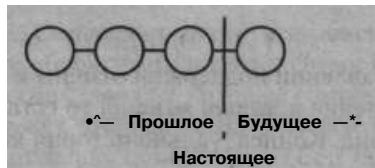


Видно, что за счет простого повторения процедуры мы получаем произвольное число уровней отмены, ограниченное лишь длиной списка истории команд.

Чтобы повторить только что отмененную команду, произведем 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 мы сдвигаем линию «настоящее» так, чтобы следующий повтор вызвал операцию Execute для следующей команды «в области 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 команда

Команды Lexi - это пример применения паттерна команда, описывающего, как инкапсулировать запрос. Этот паттерн предписывает единообразный интерфейс для выдачи запросов, с помощью которого можно сконфигурировать клиенты 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, надстроенные над базовым интерфейсом класса Command.

2.5. Проверка правописания и расстановка переносов

Наша последняя задача связана с анализом текста, точнее, 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, есть много способов реализовать поиск орфографических ошибок и вычисление точек переноса. Поэтому и здесь планировалась поддержка нескольких алгоритмов. Пользователь сможет выбрать тот алгоритм, который его больше устраивает по соотношению потребляемой памяти, скорости и качеству. Добавление новых алгоритмов тоже должно реализовываться просто.

Также мы хотим 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 - это лишь два вида анализа текста, которые Lexi мог бы поддерживать. Со временем мы собираемся 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 две стороны: доступ к анализируемой информации, которая разбросана по разным глифам в структуре документа и собственно выполнение анализа. Рассмотрим их по отдельности.

Доступ к распределенной информации

Для многих видов анализа необходимо рассматривать текст посимвольно. Но анализируемый текст рассеян по иерархии структур, состоящих из объектов-глифов. Чтобы исследовать текст, представленный в таком виде, нужен механизм доступа, знающий о структурах данных, в которых хранится текст. Для некоторых глифов потомки могут храниться в связанных списках, для других - в массивах, а для 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 реверсивного поиска нужно проходить по тексту в обратном, а не в прямом направлении. А при вычислении алгебраических выражений необходим внутренний порядок обхода.

Итак, наш механизм доступа должен уметь приспосабливаться к разным структурам данных и поддерживать разные способы обхода.

Инкапсуляция доступа и порядка обхода

Пока что в нашем интерфейсе глифов для доступа к потомкам со стороны клиентов используется целочисленный индекс. Хотя для тех классов глифов, которые содержат потомков в массиве, это, может быть, и разумно, но совершенно неэффективно для глифов, пользующихся связанным списком. Роль абстракции глифов в том, чтобы скрыть структуру данных, в которой содержатся потомки. Тогда мы сможем изменить данную структуру, не затрагивая другие классы.

Поэтому только глифу разрешено 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 возможности множественного доступа и обхода прямо в классы глифов и предоставить способ выбирать между ними, возможно, передавая константу, являющуюся элементом 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 данного подхода мы могли бы добавить в интерфейс класса `Glyph` следующие абстрактные операции:

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

Операции `First`, `Next` и `IsDone` управляют обходом. `First` производит инициализацию. В качестве параметра передается вид обхода в виде перечисляемой константы типа `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 (во внутреннем порядке). Next переходит к следующему глифу в порядке обхода, а IsDone сообщает, закончился ли обход. Get Current заменяет операцию Child - осуществляет доступ к текущему в данном обходе глифу. Старая операция Insert переписывается, теперь она вставляет глиф в текущую позицию.

При анализе можно было бы использовать следующий код на C++ для обхода структуры глифов с корнем в g в прямом порядке:

```
Glyph* g;

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

    // выполнить анализ
}
```

Обратите внимание, что мы исключили целочисленный индекс из интерфейса глифов. Не осталось ничего, что предполагало бы какой-то предпочтительный контейнер. Мы также уберегли клиенты от необходимости самостоятельно реализовывать типичные виды доступа.

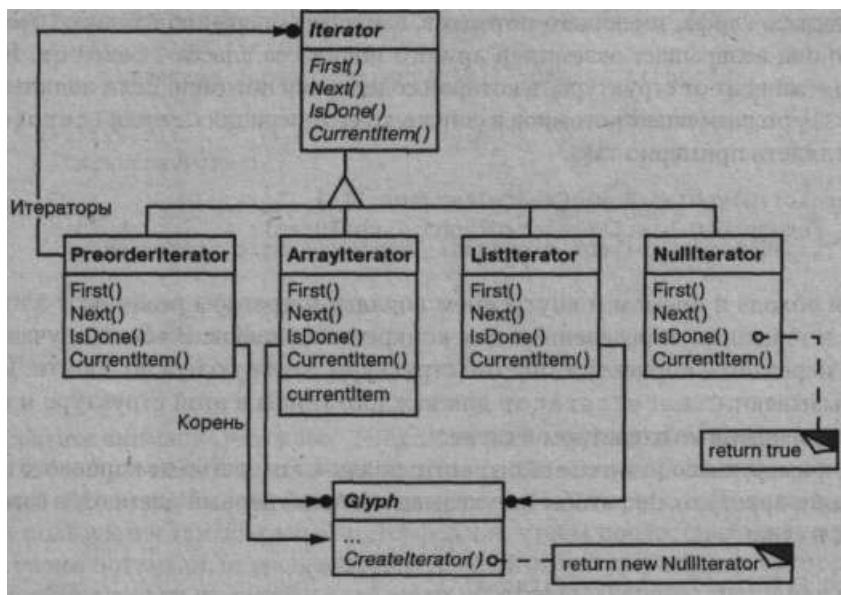
Но этот подход еще не идеален. Во-первых, здесь не поддерживаются новые виды обхода, не расширяется множество значений перечисления и не добавляются новые операции. Предположим, что нам нужен вариант прямого обхода, при котором автоматически пропускаются нетекстовые глифы. Тогда пришлось бы изменить перечисление `Traversal`, включив в него значение `TEXTUAL_PREORDER`.

Но нежелательно менять уже имеющиеся объявления. Помещение всего механизма обхода в иерархию класса `Glyph` затрудняет модификацию и расширение без изменения многих других классов. Механизм также трудно использовать повторно для обхода других видов структур. И еще нельзя иметь более одного активного обхода над данной структурой.

Как уже не раз бывало, наилучшее решение - инкапсулировать изменяющуюся сущность в класс. В данном случае это механизмы доступа и обхода. Допустимо 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 не придется изменять интерфейсы глифов или трогать реализации существующих глифов.

Класс Iterator и его подклассы

Мы применим абстрактный класс `Iterator` для определения общего интерфейса доступа и обхода. Конкретные подклассы вроде `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 обходов 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

Рис. 2. 13. Класс *Iterator* и его подклассы

Iterator и несколько его подклассов. Обратите внимание, что мы добавили в интерфейс класса *Glyph* абстрактную операцию *CreateIterator* для поддержки итераторов.

Интерфейс итератора предоставляет операции *First*, *Next* и *IsDone* для управления обходом. В классе *ListIterator* реализация операции *First* указывает на первый элемент списка, а *Next* перемещает итератор на следующий элемент. Операция *IsDone* возвращает признак, говорящий о том, перешел ли указатель за последний элемент списка. Операция *Current Item* разыменовывает итератор для возврата глифа, на который он указывает. Класс *ArrayIterator* делает то же самое с массивами глифов.

Теперь мы можем получить доступ 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 потомком
}
    
```

CreateIterator по умолчанию возвращает экземпляр *NullIterator*. *NullIterator* - это вырожденный итератор для глифов, у которых нет потомков, то есть листовых глифов. Операция *IsDone* для *NullIterator* всегда возвращает истину.

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);
}
```

Для обхода в прямом и внутреннем порядке итераторы реализуют алгоритм обхода в терминах, определенных для 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` должна будет просто вызвать операцию `CurrentItem` для итератора на вершине стека:

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

Операция `Next` обращается к итератору с вершины стека с тем, чтобы элемент, на который он указывает, создал свой итератор, спускаясь тем самым по структуре глифов как можно ниже (это ведь прямой порядок, не так ли?). `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`, то обход текущего поддерева (или листа) закончен. В таком случае `Next` снимает итератор с вершины стека и повторяет всю последовательность действий, пока не найдет следующее не полностью обойденное дерево, если таковое существует. Если же необойденных деревьев больше нет, то мы закончили обход:

```
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, что класс **Iterator** позволяет вводить новые виды обходов, не изменяя классы глифов, - мы просто порождаем новый подкласс и добавляем новый обход так, как проделали это для **PreorderIterator**. Подклассы класса **Glyph** пользуются тем же самым интерфейсом, чтобы предоставить клиентам доступ к своим потомкам, не раскрывая 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, ничего не мешает параметризовать класс типа **PreorderIterator** типом объекта структуры. В C++ мы воспользовались бы для этого шаблонами. Тогда описанный механизм итераторов можно было бы применить для обхода других структур.

Паттерн итератор

Паттерн итератор абстрагирует описанную технику 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 экранирует клиентов от деталей внутренней структуры объектов, которые они обходят. Паттерн итератор - это еще один пример того, как инкапсуляция изменяющейся сущности помогает достичь гибкости и повторной используемости. Но все равно проблема итерации оказывается 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 аккумулировать 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. Кому еще можно поручить анализ? Мы знаем, что разновидностей анализа достаточно много, и в каждом случае в те или иные моменты обхода будут выполняться различные действия. В зависимости от вида анализа некоторые глифы могут оказаться более важными, чем другие. При проверке правописания и расстановке переносов мы хотели бы рассматривать только символьные глифы и пропускать графические - линии, растровые изображения и т.д. Если мы занимаемся разделением цветов, то желательно было бы принимать во внимание только видимые, но никак не невидимые глифы. Таким образом, разные глифы должны просматриваться разными видами анализа.

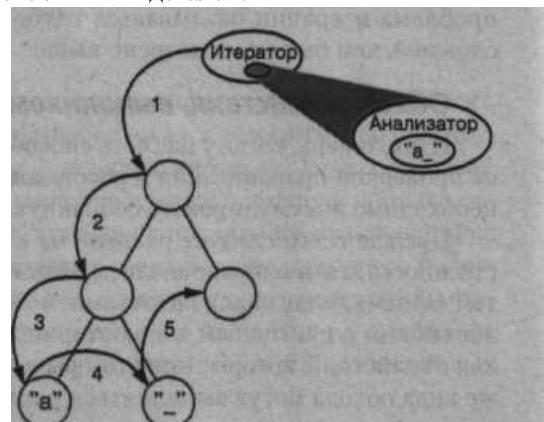
Поэтому данный вид анализа должен уметь различать глифы по их типу. Очевидное решение - встроить аналитические возможности в сами классы глифов. Тогда для каждого вида анализа мы можем добавить одну или несколько абстрактных операций в класс *Glyph* и реализовать их в подклассах в соответствии с той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

Однако неприятная особенность такого подхода состоит в том, что придется изменять каждый класс глифов 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.

Но, 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 затемнят смысл этого интерфейса. Будет трудно понять, что основная цель глифа - определить и структурировать объекты, имеющие внешнее представление и форму; в интерфейсе появится много лишних деталей.

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, чтобы класс SpellingChecker включал такой псевдокод:

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

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

    } else if (r = dynamic_cast<Row*>(glyph)) {
        // анализировать потомки г

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

Такой код опирается на специфические возможности 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, необходимость в котором хотелось бы устраниить.

Как уйти от данного грубого подхода? 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 Glyph¹:

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

¹ Мы могли бы воспользоваться перегрузкой функций, чтобы присвоить этим функциям-членам 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.

```

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)) {
        // добавить букву к _currentWord

    } else {
        // встретилась не-буква

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

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

```

Обратите внимание, что 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, находящегося

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 может продолжать проверку:

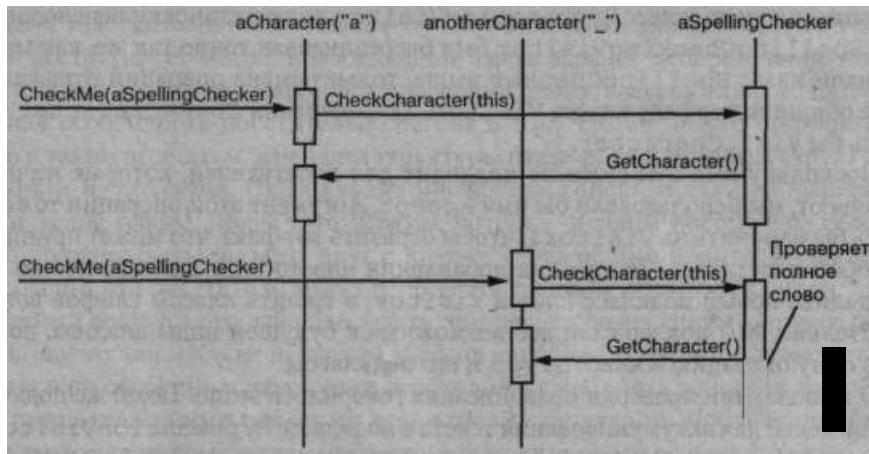
```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);

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

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



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

¹ Функция `IsMisspelled` реализует алгоритм проверки орфографии, детали которого мы здесь не приводим, 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).

всякий раз, как вводится 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 (SpellingCheckerk) 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 было 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*.

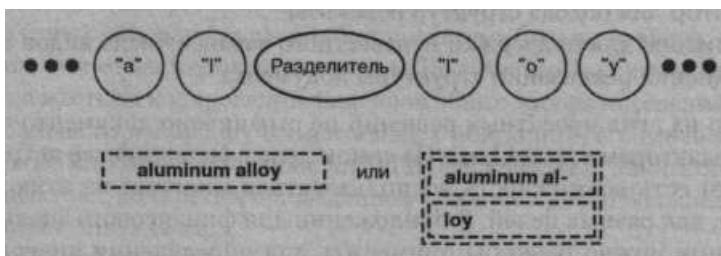
Поскольку имя *CheckMe* не подходит для посетителей, которые ничего не проверяют, мы использовали бы имя *Accept*. Аргумент этой операции тоже пришлось бы изменить на *Visitor&*, чтобы отразить тот факт, что может приниматься любой посетитель. Теперь для добавления нового вида анализа нужно лишь определить новый подкласс класса *Visitor*, а трогать классы глифов вовсе не обязательно. Мы поддержали все возможные в будущем виды анализа, добавив лишь одну операцию в класс *Glyph* и его подклассы.

О выполнении проверки правописания говорилось выше. Такой же подход будет применен для аккумулирования 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

¹«Посетить» - это лишь немногим более общее слово, чем «проанализировать». Оно просто предвосхищает ту терминологию, которой мы будем пользоваться при обсуждении следующего паттерна.

(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 не отображается вообще. Разделяющий глиф запрашивает у своего родителя (объекта **Row**), является ли он последним потомком, и делает это всякий раз, когда от него требуют отобразить себя или вычислить свои размеры. Стратегия форматирования трактует разделяющие глифы точно так же, как пропуски, считая их «кандидатами» на завершающий символ строки. На диаграмме ниже показано, как может выглядеть встроенный разделятель.



Паттерн посетитель

Вышеописанная процедура - пример применения паттерна посетитель. Его главными участниками являются класс **Visitor** и его подклассы. Паттерн посетитель абстрагирует метод, позволяющий иметь заранее неопределенное 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. Сюда входят множества, списки и даже направленные ациклические графы. Более того, классы, которые обходит посетитель, не обязательно должны быть связаны друг с другом через общий родительский класс. А это значит, что посетители могут пересекать границы иерархий классов.

Важный вопрос, который надо задать себе перед применением паттерна посетитель, звучит так: «Какие иерархии классов наиболее часто будут изменяться?» Этот паттерн особенно удобен, если необходимо выполнять действия над объектами, принадлежащими классу со стабильной структурой. Добавление нового вида посетителя не требует изменять структуру класса, что особенно важно, когда класс большой. Но при каждом добавлении нового подкласса вы будете вынуждены обновить все интерфейсы посетителя с целью включить 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 **Visit Foo**. Однако при наших проектных условиях 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:

- а компоновщик для представления физической структуры документа;
- а стратегия для возможности использования различных алгоритмов формирования;
- а декоратор для оформления пользовательского интерфейса;
- а абстрактная фабрика для поддержки нескольких стандартов внешнего облика;
- а мост для поддержки нескольких оконных систем;
- а команда для реализации отмены и повтора операций пользователя;
- а итератор для обхода структур объектов;
- а посетитель 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 разных 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, но 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 инстанцирования. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, - это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, *что* создается, *кто* это создает, *как* и *когда*. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

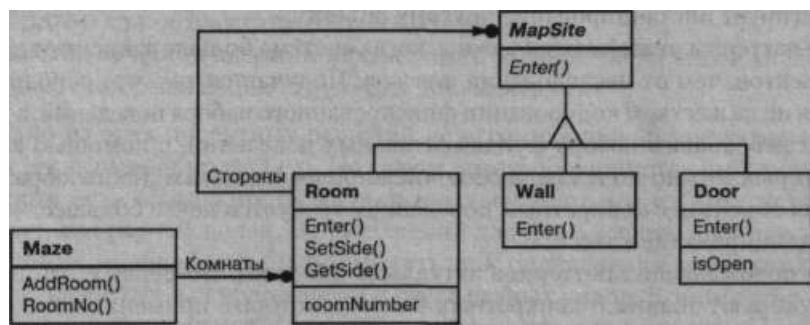
Иногда допустимо выбирать между тем или иным порождающим паттерном. Например, есть случаи, когда с пользой для дела можно использовать как прототип, так и абстрактную фабрику. В других ситуациях порождающие паттерны дополняют друг друга. Так, применяя строитель, можно использовать другие паттерны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой.

Поскольку порождающие паттерны тесно связаны друг с другом, мы изучим разу все пять, чтобы лучше были видны их сходства и различия. Изучение будет нестись 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 у игрока нет лишь один локальный вид помещения. В других случаях в лабиринтах могут встречаться задачки, которые игрок должен решить, и опасности, которые ему дстоит преодолеть. В подобных играх может отображаться карта того участка лабиринта, который уже был исследован.

Мы опустим детали того, что может встречаться в лабиринте и сколько игроков принимают участие в забаве, а сосредоточимся лишь на принципах создания лабиринта. Лабиринт мы определим 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 нашей структуры.

На диаграмме 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 терминологии языка C++:

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

В программах на языке Smalltalk для представления направлений воспользуемся соответствующими символами.

Класс `MapSite` - это общий абстрактный класс для всех компонентов лабиринта. Определим в нем только одну операцию `Enter`. 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. Если же дверь 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 игрок определяется, какой объект класса `MapSite` находится к востоку от вас, и для него вызывается операция `Enter`. Определенные в подклассах операции `Enter` «выясняют», изменили ли вы направление или расшибли нос. В реальной игре `Enter` могла бы принимать в качестве аргумента объект, представляющий 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. Номерами идентифицируются все комнаты в лабиринте:

```
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;  
};
```

Но нам необходимо знать не только об отдельных частях лабиринта. Определим еще класс Maze для представления набора комнат. В этом классе есть операция RoomNo для нахождения комнаты по ее номеру:

```
class Maze {  
public:  
    Maze();  
  
    void AddRoom(Room*);
```

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

RoomNo могла бы реализовывать свою задачу с помощью линейного списка, хэш-таблицы или даже простого массива. Но пока нас не интересуют такие детали. Займемся тем, как описать компоненты объекта, представляющего лабиринт.

Определим также класс MazeGame, который создает лабиринт. Самый простой способ сделать это - строить лабиринт с помощью последовательности операций, добавляющих к нему компоненты, которые потом соединяются. Например, следующая функция-член 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;
```

Довольно сложная функция, если принять во внимание, что она всего лишь создает 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-члена, а в ее негибкости. В функции жестко «зашита» структура лабиринта. Чтобы изменить структуру, придется изменить саму функцию, либо заместив ее (то есть полностью переписав заново), либо непосредственно модифицировав 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, что мы хотим 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?

Самое серьезное препятствие лежит в жестко зашитой в код информации о том, какие классы инстанцируются. С помощью порождающих паттернов можно различными способами избавиться от явных ссылок на конкретные классы из кода, выполняющего их инстанцирование:

- а если CreateMaze вызывает виртуальные функции вместо конструкторов для создания комнат, стен и дверей, то инстанцируемые классы можно подменить, создав подкласс MazeGame и переопределив в нем виртуальные функции. Такой подход применяется в паттерне фабричный метод;
- а когда функции CreateMaze в качестве параметра передается объект, используемый для создания комнат, стен и дверей, то их классы можно изменить, передав другой параметр. Это пример паттерна абстрактная фабрика;
- а 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;
- а если CreateMaze параметризована прототипами комнаты, двери и стены, которые она затем копирует и добавляет к лабиринту, то состав лабиринта можно варъировать, заменяя одни объекты-прототипы другими. Это паттерн прототип.

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

Паттерн Abstract Factory

Название и классификация паттерна

Абстрактная фабрика - паттерн, порождающий объекты.

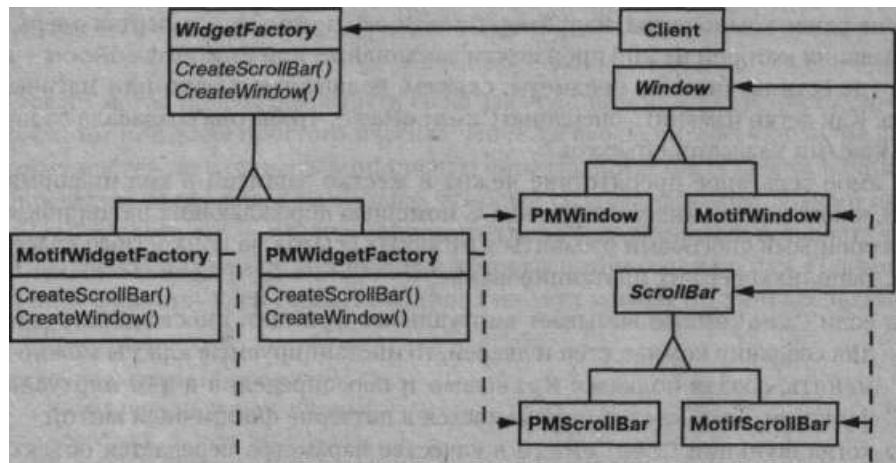
Назначение

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

Известен также под именем

Kit (инструментарий).

Мотивация



Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты внешнего облика, например Motif и Presentation Manager. Внешний облик 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 закодирован внешний облик виджетов. Если инстанцирование классов для конкретного внешнего облика разбросано по всему приложению, то изменить облик впоследствии будет нелегко.

Мы можем решить эту проблему, определив 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 внешним обликом. 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 используют. Стало быть, клиенты остаются независимыми от выбранного стандарта внешнего облика.

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

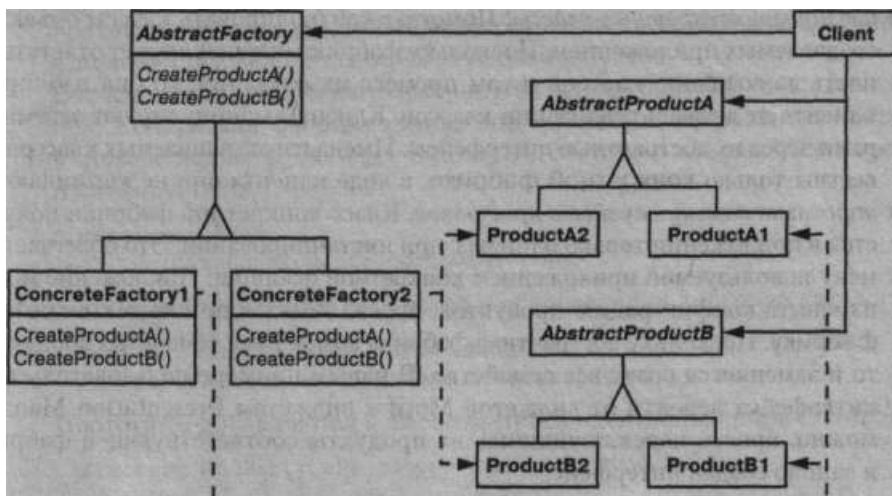
Класс Widget Factory также устанавливает зависимости между конкретными классами виджетов. Полоса прокрутки для Motif должна использоваться с кнопкой и текстовым полем Motif, и это ограничение поддерживается автоматически, как следствие использования класса MotifWidgetFactory.

Применимость

Используйте паттерн абстрактная фабрика, когда:

- Q система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- а входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;
- а система должна конфигурироваться одним из семейств составляющих ее объектов;
- а вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура



Участники

- a AbstractFactory (WidgetFactory)** - абстрактная фабрика:
 - объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- a ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)** - конкретная фабрика:
 - реализует операции, создающие конкретные объекты-продукты;
- a AbstractProduct (Window, ScrollBar)** - абстрактный продукт:
 - объявляет интерфейс для типа объекта-продукта;

- а **ConcreteProduct** (*Mot ifWindow*, *Mot ifScrollBar*) – конкретный продукт:
 - определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
 - реализует интерфейс *Abstract Product*;
- а **Client** - клиент:
 - пользуется исключительно интерфейсами, которые объявлены в классах *AbstractFactory* и *AbstractProduct*.

Отношения

- а 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;
- а *AbstractFactory* передоверяет создание объектов-продуктов своему подклассу *ConcreteFactory*.

Результаты

- Паттерн абстрактная фабрика обладает следующими плюсами и минусами:
- а *изолирует конкретные классы*. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;
 - а *упрощает замену семейств продуктов*. Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. 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 пользовательского интерфейса перейти от виджетов *Motif* к виджетам *Presentation Manager* можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс;
 - а *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;
 - а *поддержать новый вид продуктов трудно*. Расширение абстрактной фабрики для изготовления новых видов продуктов - непростая задача. 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. Решение этой проблемы мы обсудим в разделе «Реализация».

Реализация

Вот некоторые полезные приемы реализации паттерна абстрактная фабрика:

а *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;

а *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). Конкретная фабрика специфицирует 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 семейство продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна прототип. 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 подлежащие клонированию 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 идентификации их символом:

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

В языках, где сами классы являются настоящими объектами (например, Smalltalk и Objective C), возможны некие вариации подхода на базе прототипов. 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 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 подлежащих клонированию прототипов partCatalog хранит классы продуктов. Метод make: выглядит теперь следующим образом:

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

а *определение расширяемых фабрик*. Класс AbstractFactory обычно определяет разные операции для каждого вида 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 не такой 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, указывающим 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, 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 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 динамическое 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 Maze Factory может создавать компоненты лабиринтов. Он строит комнаты, стены и двери между комнатами. Им разумно воспользоваться из программы, которая считывает 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* MakeMaze0 const  
    { return new Maze; }  
    virtual Wall* MakeWalK) const  
    { return new Wall; }  
    virtual Room* MakeRoom(int n) const  
    { return new Room(n); }  
    virtual Door* MakeDoor(Room* rl, Room* r2) const  
    { return new Door(rl, r2); }  
};
```

Напомним, что функция-член 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.

Вот версия CreateMaze, в которой нет подобного недостатка, поскольку она принимает MazeFactory в качестве параметра:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {  
    Maze* aMaze = factory.MakeMaze();  
    Room* rl = factory.MakeRoom(1);  
    Room* r2 = factory.MakeRoom(2);  
    Door* aDoor = factory.MakeDoor(rl, r2);  
  
    aMaze->AddRoom(rl);  
    aMaze->AddRoom(r2);  
  
    rl->SetSide(North, factory.MakeWall());  
    rl->SetSide(East, aDoor);  
    rl->SetSide(South, factory.MakeWall());  
    rl->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;  
}
```

Мы можем создать фабрику EnchantedMazeFactory для производства волшебных лабиринтов, породив подкласс от MazeFactory. В этом подклассе заменены различные функции-члены, 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* rl, Room* r2) const
    { return new DoorNeedingSpell(rl, r2); }

protected:
    Spell* CastSpell() const;
};

```

А теперь предположим, что мы хотим построить для некоторой игры лабиринт, в одной из комнат которого заложена бомба. Если бомба взрывается, то она как минимум обрушивает стены. Тогда можно породить от класса Room подкласс, отслеживающий, есть ли в комнате бомба и взорвалась ли она. Также нам понадобится подкласс класса Wall, который хранит информацию о том, был ли нанесен ущерб стенам. 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 этом классе надо переопределить всего две функции:

```

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

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

```

Для того чтобы построить простой лабиринт, в котором могут быть спрятаны бомбы, просто вызовем функцию CreateMaze, передав ей в качестве параметра BombedMazeFactory:

```

MazeGame game;
BombedMazeFactory factory/-  

  
game.CreateMaze(factory);

```

Для построения волшебных лабиринтов CreateMaze может принимать в качестве параметра и EnchantedMazeFactory.

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 заменив 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 доступ к членам BombedWall, не имеющим аналога в его предках, то придется привести ссылку на объекты-стены от типа Wall* к типу BombedWall*. Такое приведение к типу подкласса безопасно при условии, что аргумент действительно принадлежит классу BombedWall*, а это обязательно 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 вместо ожидаемого объекта *подкласса* класса Wall. Использование абстрактной фабрики для создания стен предотвращает такие ошибки, гарантируя, что могут быть созданы лишь стены определенных типов.

Рассмотрим версию MazeFactory на языке Smalltalk, в которой есть единственная операция make, принимающая вид изготавливаемого объекта в качестве параметра. 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. Данную фабрику мы создадим с помощью метода 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». Так, для изготовления объектов пользовательского интерфейса с заданным внешним обликом определены абстрактные фабрики WidgetKit и DialogKit. В Interviews есть также класс Layout Kit, который генерирует разные объекты композиции в зависимости от того, какая требуется стратегия размещения. Например, размещение, которое концептуально можно было бы назвать «в строку», 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 Windows 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 Abstract Factory часто реализуются фабричными методами (см. паттерн фабричный метод), но могут быть реализованы и с помощью паттерна прототип.

Конкретная фабрика часто описывается паттерном одиночка.

Паттерн Builder

Название и классификация паттерна

Строитель - паттерн, порождающий объекты.

Назначение

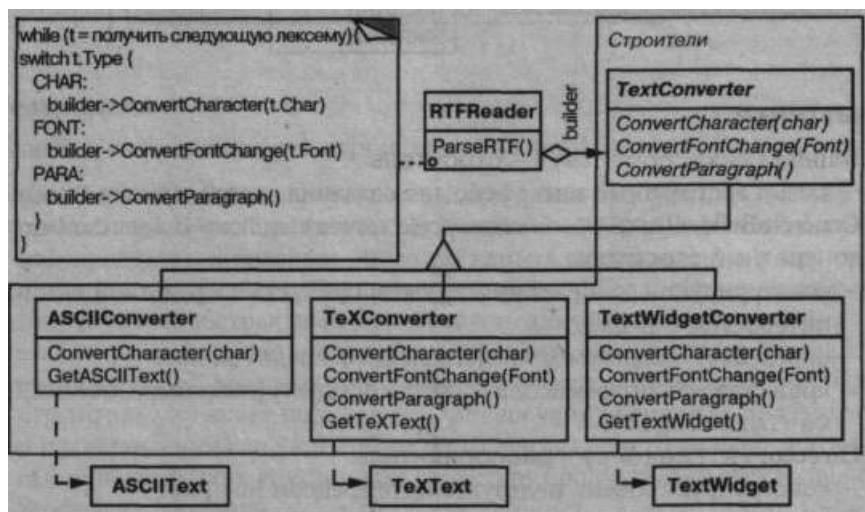
Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться различные представления.

Мотивация

Программа, в которую заложена возможность распознавания и чтения документа 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. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор.

Таким образом, нужно сконфигурировать класс 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 посыпается 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 все запросы для получения представления в формате редактора TJX, собирая по ходу необходимую информацию о стилях. А TextWidgetConverter станет 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.

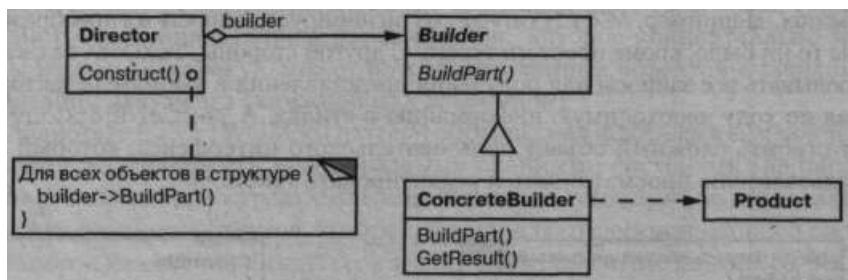
В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется *строителем*, а загрузчик - *распорядителем*. В применении к рассмотренному примеру строитель отделяет алгоритм интерпретации формата текста (то есть анализатор RTF-документов) от того, как создается и представляется 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 Text Converter.

Применимость

Используйте паттерн строитель, когда:

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

Структура



Участники

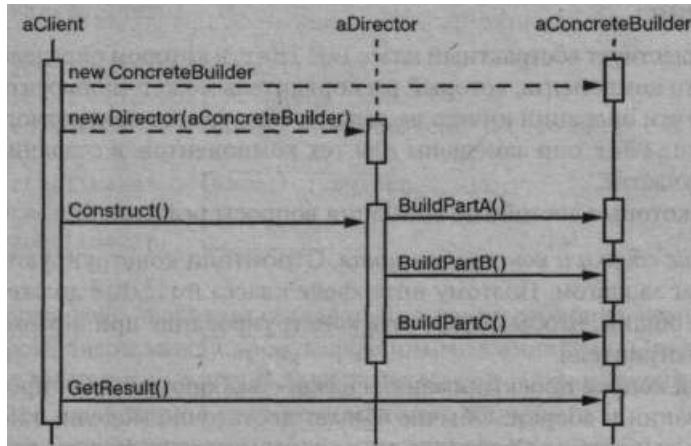
- a Director (RTFReader) - распорядитель:**
 - задает абстрактный интерфейс для создания частей объекта Product;
- a ConcreteBuilder(ASCIIConverter,TeXConverter,TextWidgetConverter)- конкретный строитель:**
 - конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;
 -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

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

Следующая диаграмма взаимодействий иллюстрирует взаимоотношения строителя и распорядителя с клиентом.



Результаты

Плюсы и минусы паттерна строитель и его применения:

- а позволяет изменять внутреннее представление продукта.* Объект Builder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление 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;
- а 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, они отсутствуют в интерфейсе 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 самыми классами *TextConverters* для генерирования представлений SGML-документов в виде ASCII-текста, TeX-текста или текстового виджета; а *дает более тонкий 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 распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя *ConcreteBuilder* они замещены для тех компонентов, в создании которых он принимает участие.

Вот еще некоторые достойные внимания вопросы реализации:

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

Но иногда может потребоваться доступ к 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 должен был бы вернуть узлы-потомки распорядителю, который затем передал бы их назад строителю, чтобы тот мог построить родительские узлы.

Q *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;

- а *пустые методы класса `Builder` по умолчанию*. В C++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

Пример кода

Определим вариант функции-члена `CreateMaze`, которая принимает в качестве аргумента строитель, принадлежащий 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 вещи: лабиринт, комнату с конкретным номером, двери между пронумерованными комнатами. Операция `GetMaze` возвращает лабиринт клиенту. В подклассах `MazeBuilder` данная операция определяется для возврата реально созданного лабиринта.

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

Имея интерфейс `MazeBuilder`, можно изменить функцию-член `CreateMaze`, чтобы она принимала строитель в качестве параметра:

```
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 относительно стен нет даже намека. За счет этого становится проще модифицировать способ представления лабиринта, поскольку ни одного 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 можно повторно использовать для построения лабиринтов разных видов. В качестве примера приведем функцию CreateComplexMaze:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    builder.BuildRoom(1001);
    return builder.GetMaze();
}
```

Обратите внимание, что 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. Чтобы следить за процессом создания, используется переменная _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 для двух комнат стены.

Конструктор StandardMazeBuilder просто инициализирует „currentMaze“:

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

BuildMaze инстанцирует объект класса Maze, который будет собираться различными операциями и, в конце концов, возвратится 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 Great eMaze 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 он для понимания и модификации, а StandardMazeBuilder легко отделяется от Maze. Еще важнее то, что разделение этих двух классов позволяет иметь множество разновидностей класса MazeBuilder, в каждом из которых есть собственные классы для комнат, дверей и стен.

Необычным вариантом MazeBuilder является класс CountingMazeBuilder. Этот строитель вообще не создает никакого лабиринта, он лишь подсчитывает число компонентов разного вида, которые могли бы быть созданы:

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

    virtual void BuildMaze0;
    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;
}
```

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]. В ней используется строитель для обработки текста, хранящегося в таком формате.

Паттерн строитель широко применяется в языке Smalltalk-80 [ParЭО]:

- а 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 синтаксического разбора Parser обращается к строителю за созданным деревом разбора и возвращает его клиенту;
- а ClassBuilder - это строитель, которым пользуются все классы для создания своих подклассов. В данном случае этот класс выступает одновременно в качестве распорядителя и продукта;
- а ByteCodeStream - это строитель, который создает откомпилированный метод в виде массива байтов. ByteCodeStream является примером нестандартного применения паттерна строитель, поскольку сложный объект представляется как массив байтов, а не как обычный объект Smalltalk. Но интерфейс к ByteCodeStream типичен для строителя, и этот класс легко можно было бы заменить другим, который представляет программу в виде составного объекта.

Родственные паттерны

Абстрактная фабрика похожа на строитель в том смысле, что может конструировать сложные объекты. Основное различие между ними в том, что строитель делает акцент 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 - это то, что часто создает строитель.

Паттерн Factory Method

Название и классификация паттерна

Фабричный метод - паттерн, порождающий классы.

Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу delegировать инстанцирование подклассам.

Известен также под именем

Virtual Constructor (виртуальный конструктор).

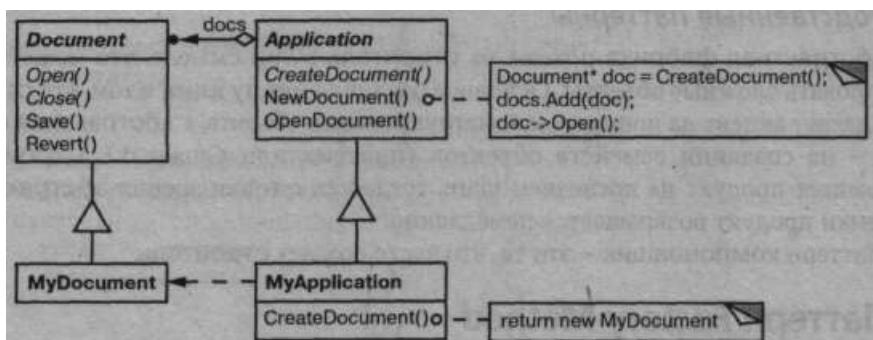
Мотивация

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 основные 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 решение о том, какой подкласс класса Document инстанцировать, зависит от приложения, то Application не может «предсказать», что именно понадобится. Этому классу известно лишь, когда нужно инстанцировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инстанцировать классы, но «знает» он лишь об абстрактных классах, которые инстанцировать нельзя.

Решение предлагает паттерн фабричный метод. В нем инкапсулируется информация о том, какой подкласс класса Document создать, и это знание выводится за пределы каркаса.



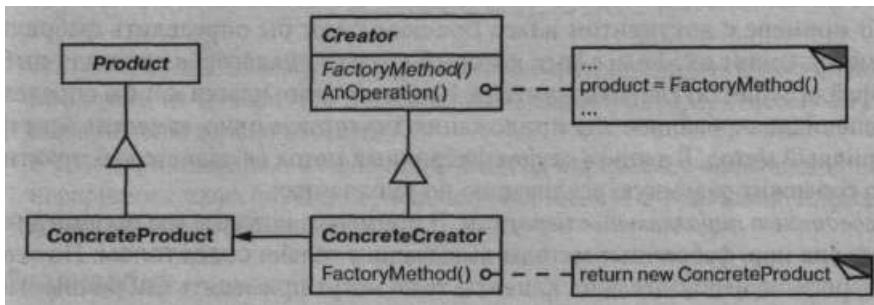
Подклассы класса Application переопределяют абстрактную операцию CreateDocument таким образом, чтобы она возвращала подходящий подкласс класса Document. Как только подкласс Application инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах. Операцию CreateDocument мы называем *фабричным методом*, поскольку она отвечает за «изготовление» объекта.

Применимость

Используйте паттерн фабричный метод, когда:

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

Структура



Участники

а Product (Document) - продукт:

- определяет интерфейс объектов, создаваемых фабричным методом;

а ConcreteProduct (MyDocument) - конкретный продукт:

- реализует интерфейс Product;

а 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.

а 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 оправдано, 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 сабричный метод:

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

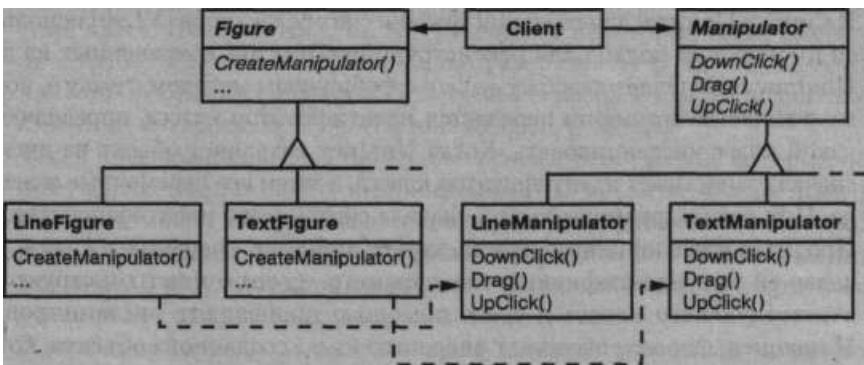
В примере с документом класс *Document* мог бы определить фабричный метод *CreateFileDialog*, который создает диалоговое окно для выбора файла существующего документа. Подкласс этого класса мог бы определить специализированное для приложения диалоговое окно, заместив этот фабричный метод. В данном случае фабричный метод не является абстрактным, а содержит разумную реализацию по умолчанию;

а *соединяет параллельные иерархии*. 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, не являющемуся производным от него. Рассмотрим, например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши. Реализация таких взаимодействий с пользователем - не всегда простое дело. Часто приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но это состояние нужно только во время самой манипуляции, поэтому помешать его в объект, представляющий фигуру, не следует. К тому же фигуры ведут себя по-разному, 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. У разных фигур будут 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 функционируют 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:

- а *две основных разновидности паттерна*. Во-первых, это случай, когда класс `Creator` является абстрактным и не содержит реализации объявленного в нем фабричного метода. Вторая возможность: `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 никакого разумного умолчания не существует. При этом обходится проблема, связанная с необходимостью инстанцировать заранее неизвестные классы. Во втором случае конкретный класс `Creator` использует фабричный метод, главным образом ради повышения гибкости. Выполняется правило: «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, инстанцируемых их родителем;
- а *параметризованные фабричные методы*. Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует 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 качестве аргумента передается идентификатор класса, определяющий, какой класс инстанцировать. Когда Unidraw сохраняет объект на диске, он сначала записывает идентификатор класса, а затем его переменные экземпляра. При реконструкции объекта сначала считывается идентификатор класса. Прочитав идентификатор класса, каркас вызывает операцию Create, передавая ей этот идентификатор как параметр. Create ищет конструктор соответствующего класса и с его помощью производит инстанцирование. 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;
    // выполнить для всех остальных продуктов...

    return 0;
}
```

Замещение параметризованного фабричного метода позволяет легко и избирательно расширять или заменять 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 больше ничего
                                // не осталось
}
```

Обратите внимание, что в самом конце операция вызывает метод Create родительского класса. Так делается постольку, поскольку MyCreator: :Create

обрабатывает только продукты YOURS, MINE и THEIRS иначе, чем родительский класс. Поэтому MyCreator расширяет некоторые виды создаваемых продуктов, а создание остальных поручает своему родительскому классу; а языково-зависимые вариации и проблемы. В разных языках возникают собственные интересные варианты и некоторые нюансы.

Так, в программах на Smalltalk часто используется метод, который возвращает класс подлежащего инстанцированию объекта. 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 инстанцируемого продукта ConcreteProduct происходит еще позже.

В версии примера Document на языке Smalltalk допустимо определить метод documentClass в классе Application. Данный метод возвращает подходящий класс Document для инстанцирования 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
```

что возвращает класс MyDocument, который должно инстанцировать приложение Application.

Еще более гибкий подход, родственный параметризованным фабричным методам, заключается в том, чтобы сохранить подлежащий созданию класс в качестве 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 исключительно виртуальными. Нужно быть осторожней и не вызывать фабричные методы в конструкторе класса Creator: в этот момент фабричный метод в производном классе ConcreteCreator еще недоступен.

Обойти такую сложность можно, если получать доступ 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. Если это не так, функция доступа создает продукт. Подобную технику 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;
}

```

а использование шаблонов, чтобы не порождать подклассы. К сожалению, допустима 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 помощью данного шаблона клиент передает только класс продукта, порождать подклассы от Creator не требуется:

```

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

StandardCreator<MyProduct> myCreator;

```

а соглашения об именовании. На практике 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 связанных с ней проблем 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* MakeMaze0 const  
    { return new Maze; }  
    virtual Room* MakeRoom(int n) const  
    { return new Room(n); }  
    virtual Wall* MakeWalk() const  
    { return new Wall; }  
    virtual Door* MakeDoor(Room* rl, Room* r2) const  
    { return new Door(rl, 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* rl = MakeRoom(1);  
    Room* r2 = MakeRoom(2);  
    Door* theDoor = MakeDoor(rl, r2);  
  
    aMaze->AddRoom(rl);  
    aMaze->AddRoom(r2);  
  
    rl->SetSide(North, MakeWall());  
    rl->SetSide(East, theDoor);  
    rl->SetSide(South, MakeWall());  
    rl->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. В этих подклассах допустимо переопределение некоторых или всех методов, 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); }
};
```

А в игре EnchantedMazeGame допустимо определить такие варианты:

```
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/вид/контроллер из языка Smalltalk-80 имеет метод defaultController, который создает контроллер, и этот метод выглядит как фабричный [ParЭО]. Но подклассы View специфицируют класс своего контроллера по умолчанию, определяя метод def aultControllerClass, возвращающий класс, экземпляры которого создает defaultController. Таким образом, реальным фабричным методом является def aultControllerClass, то есть метод, который должен переопределяться в подклассах.

Более необычным является пример фабричного метода parserClass, тоже взятый из Smalltalk-80, 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 предложения 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 часто реализуется с помощью фабричных методов. Пример в разделе «Мотивация» из описания абстрактной фабрики иллюстрирует также и паттерн фабричные методы.

Паттерн фабричные методы часто вызывается внутри шаблонных методов. В примере с документами NewDocument - это шаблонный метод.

Прототипы не нуждаются в порождении подклассов от класса 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

Название и классификация паттерна

Прототип - паттерн, порождающий объекты.

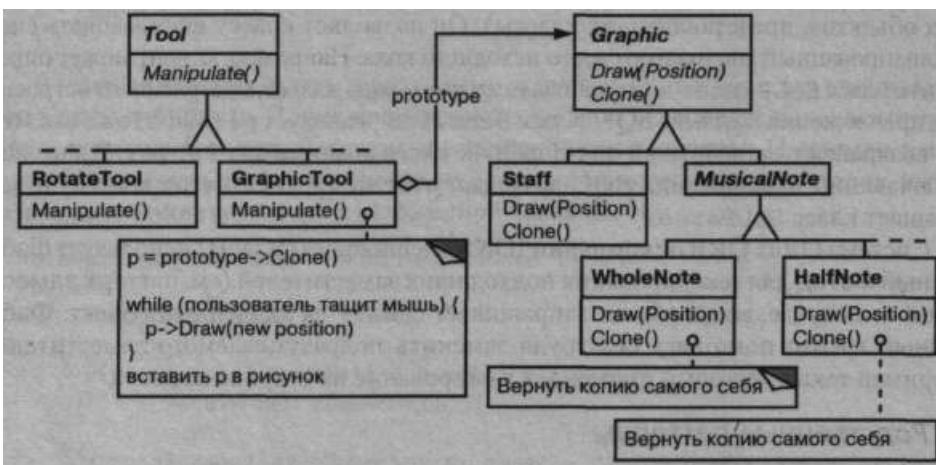
Назначение

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, пользователь, щелкнув, например, по значку четверти поместил бы ее тем самым в партитуру. Или, применив инструмент перемещения, :двигал 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 некую 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 инстанцируют. 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, копируя или «клонируя» экземпляр 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.

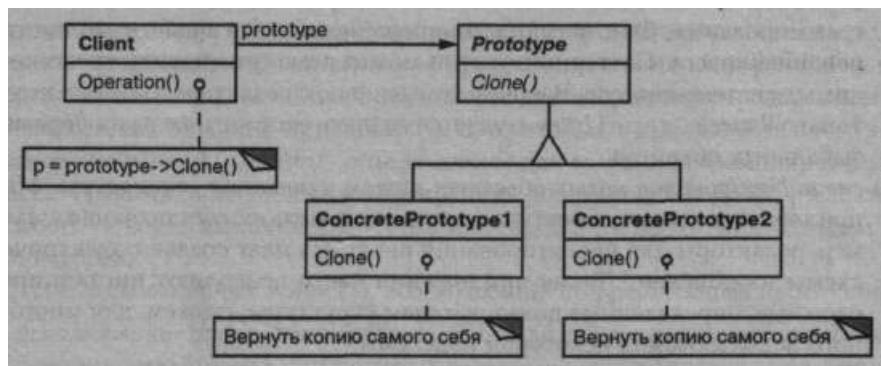
Можно воспользоваться паттерном прототип, чтобы еще больше сократить число классов. Для целых и половинных нот у нас есть отдельные классы, но, быть может, это излишне. Вместо этого они могли бы быть экземплярами одного и того же класса, инициализированного разными растровыми изображениями и длительностями 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:

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

Структура



Участники

- а **Prototype** (Graphic) - прототип:
 - объявляет интерфейс для клонирования самого себя;
- а **ConcretePrototype** (Staff - нотный стан, WholeNote - целая нота, HalfNote - половинная нота) - конкретный прототип:
 - реализует операцию клонирования себя;
- а **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 и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций.

Ниже перечислены дополнительные преимущества паттерна прототип:

- а *добавление и удаление продуктов во время выполнения*. Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе. 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;
- а *спецификация новых объектов путем изменения значений*. Динамичные системы позволяют определять поведение за счет композиции объектов - например, путем задания значений переменных объекта, - а не с помощью определения новых классов. По сути дела, вы определяете новые виды объектов, инстанцируя уже существующие классы 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 инстанцированию класса. 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;
- а *спецификация 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 инстанцировать сложные, определенные пользователем структуры, 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;
- а *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;
- а *динамическое конфигурирование 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++.

Для таких приложений характерны паттерны компоновщик и декоратор.

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 под-
пасс класса Prototype должен реализовывать операцию Clone, а это далеко не всегда просто. Например, сложно добавить операцию Clone, когда рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта есть другие объекты или наличествуют круговые ссылки.

Реализация

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

Рассмотрим основные вопросы, возникающие при реализации прототипов:

а *использование диспетчера прототипов*. Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), ведите 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 даже «просматривать» реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода;

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

В большинстве языков имеется некоторая поддержка для клонирования объектов. Например, Smalltalk предоставляет реализацию копирования, которую все подклассы наследуют от класса Object. В C++ есть копирующий конструктор. Но 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 разделяет с оригиналом эти переменные?

Поверхностное копирование просто, и часто его бывает достаточно. Именно такую возможность и предоставляет по умолчанию Smalltalk. В C++ копирующий конструктор по умолчанию выполняет почленное копирование,

- . то есть указатели разделяются 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 придется решать, что именно может разделяться и может 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 памяти, а Load создает дубликат, реконструируя объект из буфера;

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

Может оказаться, что в ваших классах прототипов уже определяются операции для установки и очистки 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: копии может понадобиться 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, принимающим в качестве аргументов прототипы:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);
    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
```

```

    virtual Wall* MakeWalk() 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-члены для создания стен, комнат и дверей похожи друг на друга: каждая клонирует, а затем инициализирует прототип. Вот определения функций `..akeWall` и `MakeDoor`:

```

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

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

Мы можем применить `MazePrototypeFactory` для создания прототипичного или принимаемого по умолчанию лабиринта, 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 инициализируем `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 стр. 91:

```
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* rl, Room* r2) {
    _room1 = rl;
    _room2 = r2;
}

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

Подкласс BombedWall должен заместить операцию Clone и реализовать ее 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, клонирующими прототип, не надо было знать о его конкретных подклассах. Клиентам никогда не придется приводить значение, возвращаемое `Clone`, к нужному типу.

В Smalltalk разрешается использовать стандартный метод копирования, унаследованный от класса `Object`, для клонирования любого прототипа `MapSite`. Можно воспользоваться фабрикой `MazeFactory` для изготовления любых необходимых прототипов. Например, допустимо создать комнату по ее номеру `#room`. В классе `MazeFactory` есть словарь, сопоставляющий именам прототипы. 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

Быть может, впервые паттерн прототип был использован в системе Sketchpad Ивана 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 отладчиков на базе ET++, где имеется интерфейс вида 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. Добавить к etgdb новые отладчики можно, связав ядро 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 являются конкурентами. Но их используют и совместно. Абстрактная фабрика может хранить набор прототипов, которые клонируются и возвращают изготовленные объекты.

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

Паттерн Singleton

Название и классификация паттерна

Одиночка - паттерн, порождающий объекты.

Назначение

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

Мотивация

Для некоторых классов важно, чтобы существовал только один экземпляр. Хотя в системе может быть много принтеров, но возможен лишь один спулер. Должны 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 инстанцировать класс в нескольких экземплярах.

Более удачное решение - сам класс контролирует то, что у него есть только один экземпляр, 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.

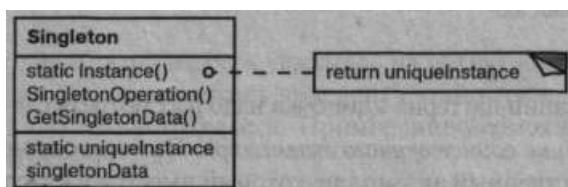
Применимость

Используйте паттерн одиночка, когда:

а должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;

а единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Участники

а **Singleton** - одиночка:

- определяет операцию `Instance`, которая позволяет клиентам получать доступ к единственному экземпляру. `Instance` - это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция-член в C++;
- может нести ответственность за создание собственного уникального экземпляра.

Отношения

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

Результаты

У паттерна одиничка есть определенные достоинства:

- Q **контролируемый доступ к единственному экземпляру.** Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- а **уменьшение числа имен.** Паттерн одиничка - шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;
- а **допускает уточнение операций и представления.** От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения;
- а **допускает переменное число экземпляров.** Паттерн позволяет вам легко изменить свое решение и разрешить появление более одного экземпляра класса Singleton. Вы можете применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton;
- а **большая гибкость, чем у операций класса.** Еще один способ реализовать функциональность одинички - использовать операции класса, то есть статические функции-члены в C++ и методы класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких экземпляров класса. Кроме того, статические функции-члены в C++ не могут быть виртуальными, так что их нельзя полиморфно заменить в подклассах.

Реализация

При использовании паттерна одиничка надо рассмотреть следующие вопросы:

- а **гарантирование единственного экземпляра.** Паттерн одиничка устроен так, что тот единственный экземпляр, который имеется у класса, - самый обычный, но больше одного экземпляра создать не удастся. Чаще всего для этого прячут операцию, создающую экземпляры, за операцией класса (то есть за статической функцией-членом 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 одиничка будет создан и инициализирован перед первым использованием.
- В C++ операция класса определяется с помощью статической функции-члена Instance класса Singleton. В этом классе есть также статическая

переменная-член „`instance`”, которая содержит указатель на уникальный экземпляр.

Класс `Singleton` объявлен следующим образом:

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

А реализация такова:

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

Клиенты осуществляют доступ к одиночке исключительно через функцию-член `Instance`. Переменная „`instance`” инициализируется нулем, а статическая функция-член `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 хранится вплоть до момента первого обращения.

Обратите внимание, что конструктор защищенный. Клиент, который попытается инстанцировать класс `Singleton` непосредственно, получит ошибку на этапе компиляции. Это дает гарантию, что будет создан только один экземпляр.

Далее, поскольку „`instance`” – указатель на объект класса `Singleton`, то функция-член `Instance` может присвоить этой переменной указатель на любой подкласс данного класса. Применение возможности мы увидим в разделе «Пример кода».

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

- мы не можем гарантировать, что будет объявлен только один экземпляр статического объекта;
- у нас может не быть достаточно информации для инстанцирования любого одиночки во время статической инициализации. Одиночке могут быть необходимы 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 (хотя и не слишком серьезный) недостаток глобальных/статических объектов в том, что приходится создавать всех одиночек, даже, если они не используются. Применение статической функции-члена 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: 'не удается создать новый объект'  

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

^ SoleInstance
```

а порождение подклассов Singleton. Основной вопрос не столько в том, как определить подкласс, а в том, как сделать, чтобы клиенты могли использовать его единственный экземпляр. 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 (скомпоновав программу с объектным файлом, содержащим другую реализацию), но от клиента одиночка будет по-прежнему скрыта.

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 в нем список объектов `NameSingletonPair`. Каждый такой объект отображает имя на одиночку. Операция `Lookup` ищет одиночку по имени. Предположим, что имя нужного одиночки передается в переменной среды:

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // пользователь или среда предоставляют это имя на стадии
        // запуска программы

        _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 инстанцирует класс, 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, где находится реализация `MySingleton`.

Теперь класс `Singleton` не отвечает за создание одиночки. Его основной обязанностью становится обеспечение доступа к объекту-одиночке из

любой части системы. Подход, сводящийся к применению статического объекта, по-прежнему имеет потенциальный недостаток: необходимо создавать экземпляры всех возможных подклассов Singleton, иначе они не будут зарегистрированы.

Пример кода

Предположим, нам надо определить класс `MazeFactory` для создания лабиринтов, описанный на стр. 99. `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 инстанцирование, в результате которого будет создан лишний экземпляр:

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

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

Реализация класса такова:

```
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 инстанцирует нужный

подкласс MazeFactory в зависимости от значения данной переменной. Лучше всего поместить код в операцию Instance, поскольку она уже и так инстанцирует 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 нужно модифицировать при определении каждого нового подкласса MazeFactory. В данном приложении это, может быть, и не проблема, 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 [ParЭO] 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 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 распределения событий в приложении. Он хранит пользовательские настройки стиля и управляет подключением к одному или нескольким физическим дисплеям. WidgetKit - это абстрактная фабрика для определения внешнего облика интерфейсных виджетов. Операция WidgetKit::instance() определяет конкретный инстанцируемый подкласс WidgetKit на основе переменной среды, которую устанавливает Session. Аналогичная операция в классе Session «выясняет», поддерживаются ли монохромные или цветные дисплеи, и соответственно конфигурирует одиночку Session.

Родственные паттерны

С помощью паттерна одиничка могут быть реализованы многие паттерны. См. описание абстрактной фабрики, строителя и прототипа.

Обсуждение порождающих паттернов

Есть два наиболее распространенных способа параметризовать систему классами создаваемых ей объектов. Первый способ - порождение подклассов от класса, создающего объекты. Он соответствует паттерну фабричный метод. Основной недостаток метода: требуется создавать новый подкласс лишь для того, чтобы изменить класс продукта. И таких изменений может быть очень много. 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, строитель и прототип. 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 возврат продукта.

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

- а 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;
- а 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;
- о применить паттерн прототип. Тогда в каждом подклассе `Graphic` будет реализована операция `Clone`, а `GraphicTool` параметризуется прототипом создаваемого графического объекта.

Выбор 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 равновеликой иерархии классов *GraphicsFactory*. Абстрактную фабрику следует предпочесть фабричному методу 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.

Очевидно, 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 инстанцирования.

В случае применения паттерна фабричный метод проект в большей степени поддается настройке и оказывается лишь немногим более сложным. 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 инстанцируемый класс никогда не изменяется или когда инстанцирование выполняется внутри операции, которую легко можно заменить 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. Особенно полезен этот паттерн, когда нужно организовать совместную работу нескольких независимо разработанных библиотек. Другой пример паттерна уровня класса - адаптер. 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**. Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать произвольно сложные структуры из примитивных и других составных объектов. В паттерне заместитель объект берет на себя функции другого объекта. У заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве. Или 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 определяет структуру для совместного использования объектов. Владельцы разделяют 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 контекстных зависимостей объекты-приспособленцы могут легко разделяться.

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

Паттерн декоратор 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 вкладывания 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

Название и классификация паттерна

Адаптер - паттерн, структурирующий классы и объекты.

Назначение

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

Известен также под именем

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.

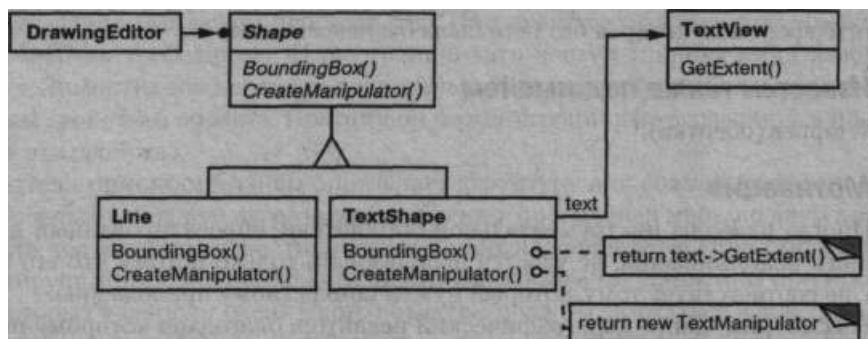
Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет

изменяемую форму и изображает сам 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, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс Text Shape, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно не-тривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс TextView, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать TextView для реализации Text Shape, но библиотека разрабатывалась без учета классов Shape, поэтому заставить объекты TextView и Shape работать совместно не удается.

Так каким же образом существующие и независимо разработанные классы вроде TextView могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса TextView, чтобы он соответствовал интерфейсу Shape, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять TextView; библиотека не должна приспособливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс Text Shape так, что он будет *адаптировать* интерфейс TextView к интерфейсу Shape. Это допустимо сделать двумя способами: наследуя интерфейс от Shape, а реализацию от TextView; включив экземпляр TextView в Text Shape и реализовав Text Shape в терминах интерфейса TextView. Два данных подхода соответствуют вариантам паттерна адаптер в его классовой и объектной ипостасях. Класс Text Shape мы будем называть *адаптером*.



На этой диаграмме показан адаптер объекта. Видно, как запрос BoundingBox, объявленный в классе Shape, преобразуется в запрос Get Extent, определенный

в классе TextView. Поскольку класс Text Shape 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 диаграмме показано, как адаптер выполняет такого рода функции. 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 объектов, которым известно, как 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 Text Shape. Возвращая экземпляр TextManipulator, объект класса TextShape добавляет новую функциональность, которой в классе TextView нет, а классу Shape требуется.

Применимость

Применяйте паттерн адаптер, когда:

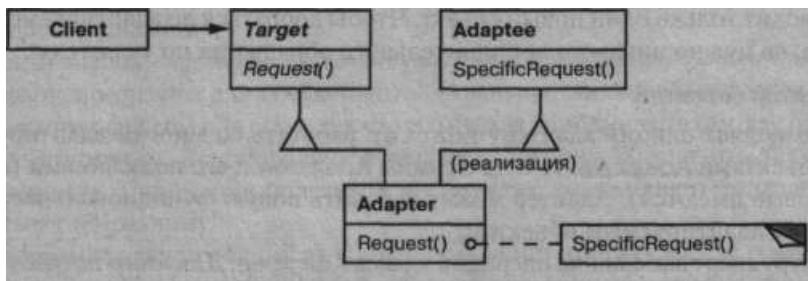
Q хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;

а собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;

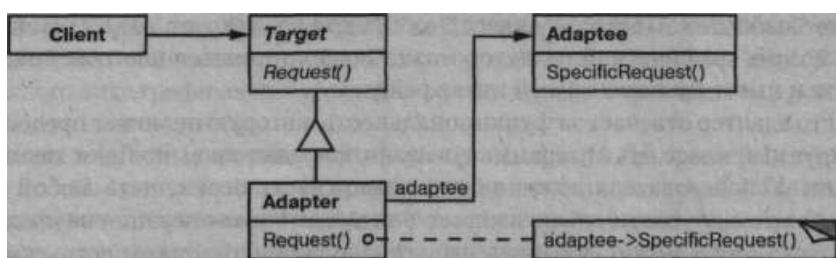
Q (только для адаптера объектов!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем рождения новых подклассов от каждого. В этом случае адаптер объектов может приспособливать интерфейс их общего родительского класса.

Структура

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



Адаптер объекта применяет композицию объектов.



Участники

a Target (Shape) – целевой:

- определяет зависящий от предметной области интерфейс, которым пользуется Client;

a Client (DrawingEditor) – клиент:

- вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;

a Adaptee (Textview) – адаптируемый:

- определяет существующий интерфейс, который нуждается в адаптации;

a Adapter (Text Shape) – 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

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- а адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
- а позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
- а вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- а позволяет одному адаптеру Adapter работать со многими адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если такие имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- а 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 приведены вопросы, которые следует рассмотреть, когда вы решаете применить паттерн адаптер:

- а *объем работы по адаптации*. Адаптеры сильно отличаются по тому объему работы, который необходим 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;
- а *сменные адAPTERы*. Степень повторной используемости класса тем выше, чем меньше предположений делается о тех классах, которые будут его применять. Встраивая адаптацию интерфейса в класс, вы отказываетесь от предположения, что другим классам станет доступен тот же самый интерфейс. Другими словами, адаптация интерфейса позволяет включить ваш класс в существующие системы, которые спроектированы для класса с другим интерфейсом. В системе *ObjectWorks\Smalltalk* [РагЭО] используется термин *сменный адAPTER* (*pluggable adapter*) для обозначения классов со встроенной адаптацией интерфейса.

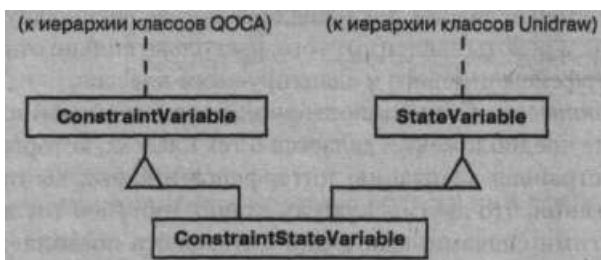
Рассмотрим виджет *TreeDisplay*, позволяющий графически отображать древовидные структуры. Если бы это был специализированный виджет, предназначенный только для одного приложения, то мы могли бы потребовать специального интерфейса от объектов, которые он отображает. Но если мы хотим сделать его повторно используемым (например, частью библиотеки полезных виджетов), то предъявлять такое требование неразумно. Разные приложения, скорей всего, 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 *GetSubdirector ies*, тогда как для иерархии наследования соответствующая операция может называться *Get Subclasses*. 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»;

- а *использование двусторонних адAPTERов для обеспечения прозрачности*. АдAPTERы непрозрачны для всех клиентов. Адаптированный объект уже не обладает интерфейсом *Adaptee*, так что его нельзя использовать там, где *Adaptee* был применим. *Двусторонние адAPTERы* способны обеспечить такую прозрачность. Точнее, они полезны в тех случаях, когда клиент должен видеть объект по-разному.

Рассмотрим двусторонний адAPTER, 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. 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, кое о чем все же стоит помнить:

а 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; а сменные адAPTERы. Рассмотрим три способа реализации сменных адаптеров для описанного выше виджета TreeDisplay, который может автоматически отображать иерархические структуры.

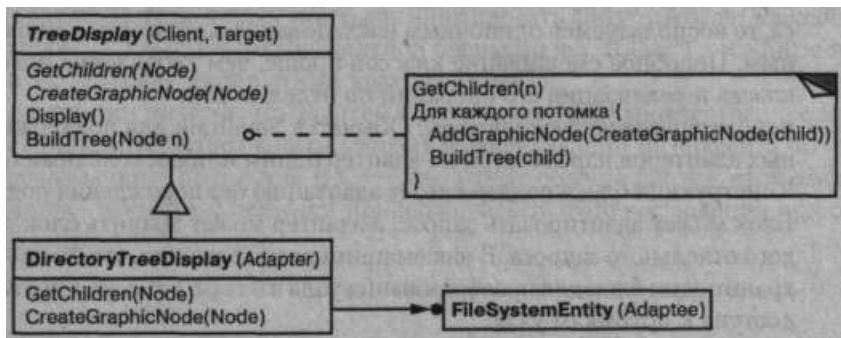
Первый шаг, общий для всех трех реализаций, - найти «узкий» интерфейс для Adaptee, то есть наименьшее подмножество операций, позволяющее выполнить адаптацию. «Узкий» интерфейс, состоящий всего из пары итераций, легче адаптировать, чем интерфейс из нескольких десятков операций. Для TreeDisplay адаптации 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

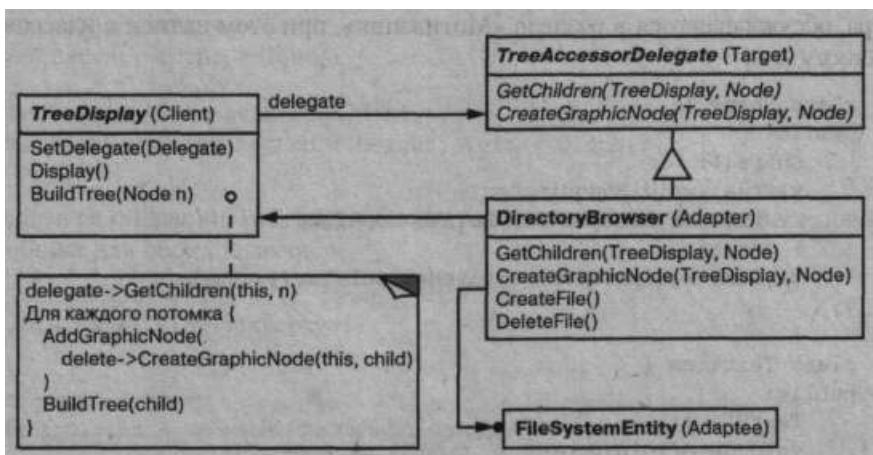
и адаптировать иерархически структурированный объект. Например, подкласс DirectoryTreeDisplay при их реализации будет осуществлять доступ к структуре каталогов файловой системы.

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



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

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



В статически типизированных языках вроде C++ требуется явно определять интерфейс для уполномоченного. Специфицировать такой интерфейс можно, поместив «узкий» интерфейс, который необходим классу TreeDisplay, в абстрактный класс TreeAccessorDelegate. После этого допустимо добавить этот интерфейс к выбранному уполномоченному - в данном случае DirectoryBrowser - с помощью наследования. Если у DirectoryBrowser еще нет существующего родительского класса, то воспользуемся одиночным наследованием, если есть - множественным. Подобное смешивание классов проще, чем добавление нового подкласса 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 вы встраиваете интерфейс адаптации 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(  
        Points bottomLeft, Point& topRight  
    ) const;  
    virtual Manipulator* CreateManipulator() const;  
};  
  
class TextView {  
public:  
    TextView();  
    void GetOrigin(Coord& x, Coords y) const;
```

```
void GetExtent(Coord& width, Coords height) const;
virtual bool IsEmpty() const;
};
```

В классе Shape предполагается, что ограничивающий фигуру прямоугольник определяется двумя противоположными углами. Напротив, в классе TextView он характеризуется 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 Text Shape 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 Text Shape:

```
class TextShape : public Shape, private TextView {
public:
    TextShape();
    virtual void BoundingBox(
        Point& bottomLeft, Points topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

Операция BoundingBox преобразует интерфейс TextView к интерфейсу Shape:

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

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

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

На примере операции IsEmpty демонстрируется прямая переадресация запросов, общих для обоих классов:

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

CreateManipulator - это пример фабричного метода.

Наконец, мы определим операцию CreateManipulator (отсутствующую в классе TextView) 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 подходе адаптер TextShape содержит указатель на TextView:

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

    virtual void BoundingBox(
        Point& bottomLeft, Points 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 мы предположим, что клиент создает объект TextView и передает его конструктору класса TextShape:

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Points 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

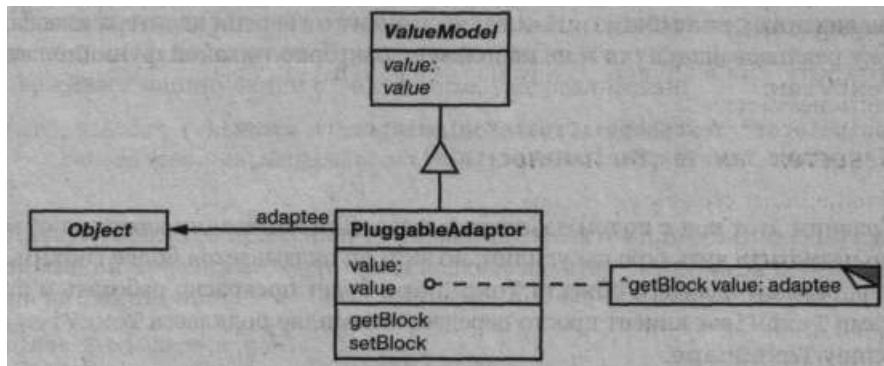
Пример, приведенный в разделе «Мотивация», заимствован из графического приложения ET++Draw, основанного на каркасе ET++ [WGM88]. ET++Draw повторно использует классы ET++ для редактирования текста, применяя для адаптации класс TextShape.

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), и 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 видов, которые отображают единственное значение. **ValueModel** определяет интерфейс **value**, **value:** для доступа к значению. Это абстрактные методы. 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**, называемыйся **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, обеспечивая тем самым некоторое синтаксическое удобство. Данные 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 табличей для получения от объекта значения в колонках.

В некоторых классах библиотеки NeXT AppKit [Add94] используются объекты-уполномоченные для реализации интерфейса адаптации. В качестве примера можно привести класс NXBrowser, который способен отображать иерархические списки данных. NXBrowser пользуется объектом-уполномоченным для доступа и адаптации данных.

Придуманная Скоттом Мейером (Scott Meyer) конструкция «брак по расчёту» (Marriage of Convenience) [Mey88] это разновидность адаптера класса. Мейер описывает, как класс FixedStack адаптирует реализацию класса Array к интерфейсу класса Stack. Результатом является стек, содержащий фиксированное число элементов.

Родственные паттерны

Структура паттерна мост 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 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 определяет представителя или суррогат другого объекта, но не изменяет его интерфейс.

Паттерн Bridge

Название и классификация паттерна

Мост - паттерн, структурирующий объекты.

Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Известен также под именем

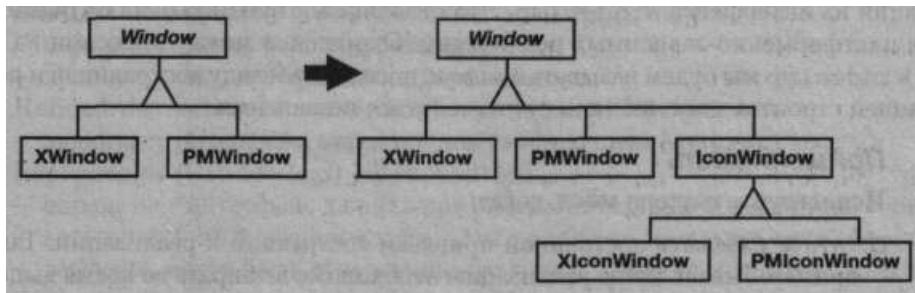
Handle/Body (описатель/тело).

Мотивация

Если для некоторой абстракции возможно несколько 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 реализаций.

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 мы могли бы определить абстрактный класс `Window` и его подклассы `XWindow` и `PMWindow`, реализующие интерфейс окна для разных платформ. Но у такого решения есть два недостатка:

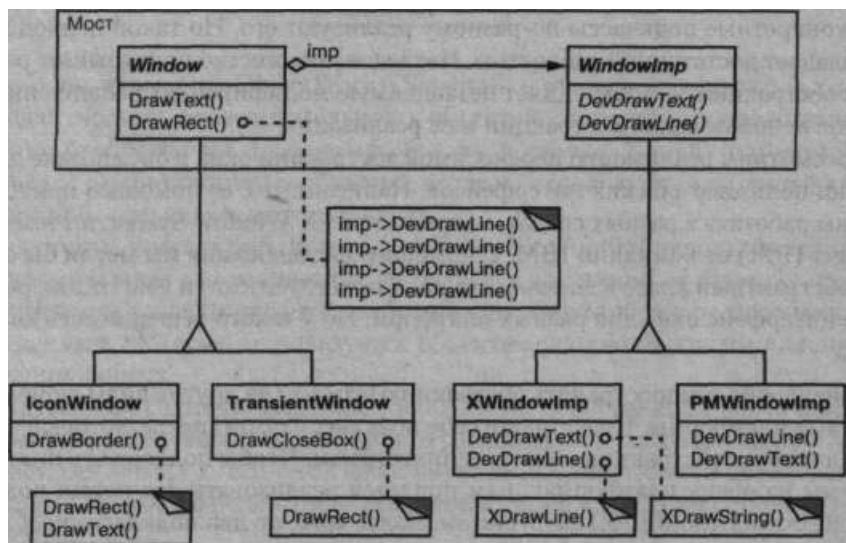
- а неудобно распространять абстракцию `Window` на другие виды окон или новые платформы. Представьте себе подкласс `IconWindow`, который специализирует абстракцию окна для пиктограмм. Чтобы поддержать пиктограммы на обеих платформах, нам придется реализовать *два* новых подкласса `XIconWindow` и `PMIconWindow`. 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`;



- а клиентский код становится платформенно-зависимым. При создании окна клиент инстанцирует конкретный класс, имеющий вполне определенную реализацию. Например, создавая объект `XWindow`, мы привязываем абстракцию окна к ее реализации для системы X Window и, следовательно, делаем код клиента ориентированным именно на эту оконную систему. Таким образом усложняется перенос клиента на другие платформы.

Клиенты должны иметь возможность создавать окно, не привязываясь к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение. Поэтому в клиентском коде не может быть никаких упоминаний о платформах.

С помощью паттерна мост эти проблемы решаются. 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 платформенно-зависимых реализаций. Так, подкласс XWindowImp предоставляет реализацию в системе X Window System.



Все операции подклассов Window реализованы в терминах 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.

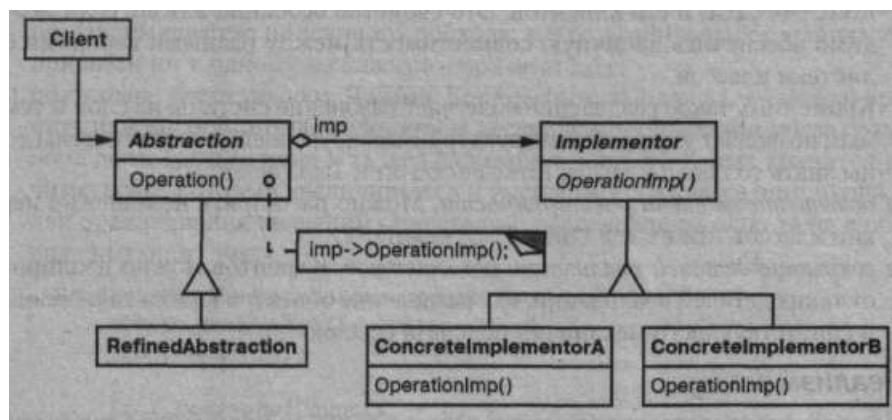
Применимость

Используйте паттерн мост, когда:

- а хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
- а и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- о изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

- а (*только для C++!*) вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;
- а число классов начинает быстро расти, как мы видели 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];
- а вы хотите разделить одну реализацию между несколькими объектами (быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента. Простой пример - это разработанный Джеймсом Коплином класс **String** [Cop92], в котором разные объекты могут разделять одно и то же представление строки (StringRep).

Структура



Участники

- а **Abstraction** (Window) - абстракция:
 - определяет интерфейс абстракции;
 - хранит ссылку на объект типа **Implementor**;
- а **RefinedAbstraction** (iconWindow) - уточненная абстракция:
 - расширяет интерфейс, определенный абстракцией **Abstraction**;
- а **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, базирующиеся на этих примитивах;
- а **ConcreteImplementor** (XWindowImp, PMWindowImp) - конкретный реализатор:
 - содержит конкретную реализацию интерфейса класса **Implementor**.

Отношения

Объект *Abstraction* перенаправляет своему объекту *Implementor* запросы клиента.

Результаты

Результаты применения паттерна мост таковы:

- а *отделение реализации от интерфейса*. Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию.

Разделение классов *Abstraction* и *Implementor* устраниет также зависимости от реализации, устанавливаемые на этапе компиляции. Чтобы изменить класс реализации, вовсе 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*;

- а *повышение степени расширяемости*. Можно расширять независимо иерархии классов *Abstraction* и *Implementor*;
- а *скрытие деталей реализации от клиентов*. Клиентов можно изолировать от таких деталей реализации, как разделение объектов класса *Implementor* и сопутствующего механизма подсчета ссылок.

Реализация

Если вы предполагаете применить паттерн мост, то подумайте о таких вопросах реализации:

- а *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 нужно, чтобы изменение реализации класса не отражалось 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;

- а *создание правильного объекта Implementor*. Как, когда и где принимается решение о том, какой из нескольких классов *Implementor* инстанцировать? Если у класса *Abstraction* есть информация о конкретных классах *ConcreteImplementor*, то он может инстанцировать один из них в своем конструкторе; какой именно - зависит от переданных конструктору параметров.

Так, если класс коллекции поддерживает несколько реализаций, то решение можно принять в зависимости от размера коллекции. Для небольших коллекций применяется реализация в виде связанного списка, для больших - в виде хэшированных 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 становится больше некоторой условной величины, то мы переключаемся с одной реализации на другую, более эффективную. Можно также делегировать решение другому объекту. В примере с иерархиями Window/WindowImp уместно было бы ввести фабричный объект (см. паттерн абстрактная фабрика), 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, при этом понятно, что объект получит то, что нужно. Преимущество описанного подхода: класс Abstraction напрямую не привязан ни к одному из классов Implementor;

а *разделение реализаторов*. Джеймс Копlien 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, разделяющим одно тело, в общем виде выглядит так:

```
Handles Handle::operator= (const Handles other) {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;

        _body = other._body;
    }

    return *this;
```

```

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 Points, 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 Points) = 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 () {
    GetView0 ->DrawOn(this) ;
}
```

А в классе IconWindow содержится 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);
    }
}
```

Могут существовать и другие разновидности 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 ZconDockWindow (контейнер пиктограмм) хранит окна класса IconWindow и располагает их в ряд.

Операции класса Window определены в терминах 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 Points 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 специфичное для X Window состояние,
    // в том числе:
    Display* _dpy;
    Drawable _winid; // идентификатор окна
    GC _gc;          // графический контекст окна
};

```

Для Presentation Manager (PM) мы определяем класс PMWindowImp:

```

class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // прочие операции открытого интерфейса...

private:
    // переменные, описывающие специфичное для PM Window состояние,
    // в том числе:
    HPS_hps;
};

```

Эти подклассы реализуют операции WindowImp в терминах примитивов оконной системы. Например, DeviceRect для X Window реализуется так:

```

void XWindowImp::DeviceRect (
    Coord xO, Coord yO, Coord xl, Coord yl
) {
    int x = round(min(xO, xl));
    int y = round(min(yO, yl));
    int w = round(abs(xO - xl));
    int h = round(abs(yO - yl));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}

```

А для PM - так:

```

void PMWindowImp::DeviceRect (
    Coord xO, Coord yO, Coord xl, Coord yl
) {
    Coord left = min(xO, xl);
    Coord right = max(xO, xl);
    Coord bottom = min(yO, yl);
    Coord top = max(yO, yl);

    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, OL);
}
}
}

```

Как окно получает экземпляр нужного подкласса WindowImp? В данном примере мы предположим, что за это отвечает класс Window. Его операция GetWindowImp получает подходящий экземпляр от абстрактной фабрики (см. описание паттерна абстрактная фабрика), 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 системно-зависимые объекты. Для простоты мы сделали эту фабрику одиночкой 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 платформенно-зависимых 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 извещения Window о событиях, специфичных для WindowPort: поступлений 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 вопросах управления памятью, например разделении представления строк и поддержке объектов переменного размера. Нас же в первую очередь интересует поддержка независимых расширений абстракции и ее реализации.

В библиотеке libg++ [Lea88] определены классы, которые реализуют универсальные структуры данных: Set (множество), LinkedSet (множество как связанный список), HashSet (множество как хэш-таблица), LihkedList (связанный список) и HashTable (хэш-таблица). Set - это абстрактный класс, определяющий абстракцию множества, а LinkedList и HashTable - конкретные реализации связанного списка и хэш-таблицы. LinkedSet и HashSet - реализаторы абстракции Set, перекидывающие 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 выяснить, какой реализацией пользоваться в конкретных условиях.

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 также 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 может создать и сконфигурировать мост.

Для обеспечения совместной работы не связанных между собой классов прежде всего предназначен паттерн адаптер. 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

Название и классификация паттерна

Компоновщик - паттерн, структурирующий объекты.

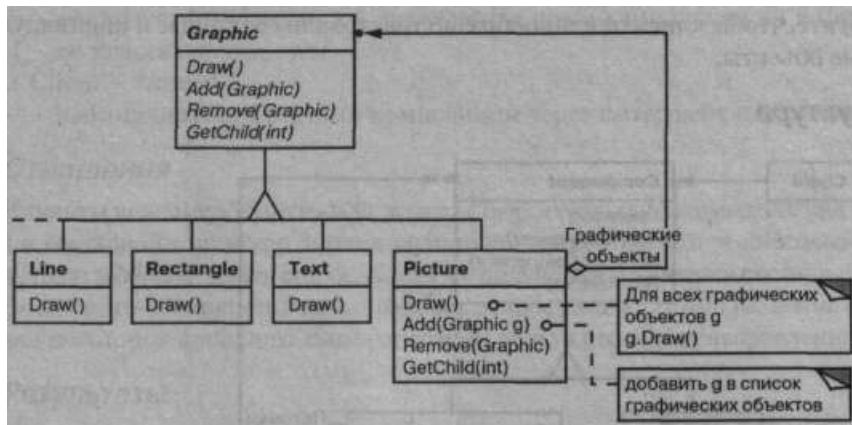
Назначение

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 диаграммы из более простых компонентов. Проектировщик может сгруппировать мелкие компоненты для формирования более крупных, которые, в свою очередь, могут стать основой для создания еще более крупных. В простой реализации допустимо было бы определить классы графических примитивов, например текста и линий, а также классы, выступающие в роли 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.

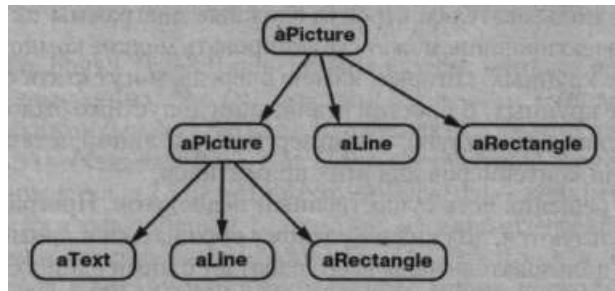


Ключом к паттерну компоновщик является абстрактный класс, который представляет одновременно и примитивы, и контейнеры. В графической системе этот класс может называться *Graphic*. В нем объявлены операции, специфичные для каждого вида графического объекта (такие как *Draw*) и общие для всех составных объектов, например операции для доступа и управления потомками.

Подклассы *Line*, *Rectangle* и *Text* (см. диаграмму 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.

Ниже на диаграмме показана типичная структура составного объекта, рекурсивно скомпонованного из объектов класса Graphic.

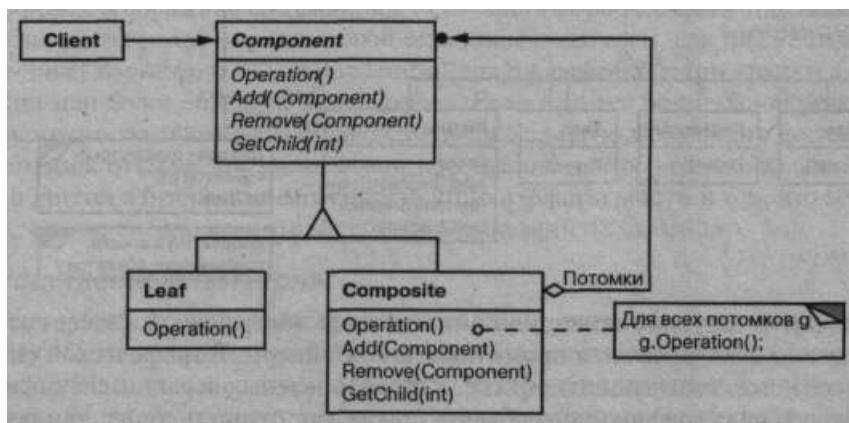


Применимость

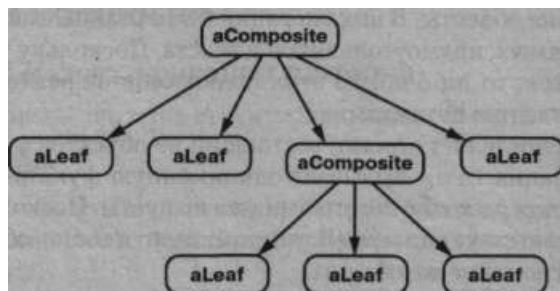
Используйте паттерн компоновщик, когда:

- а нужно представить иерархию объектов вида часть-целое;
- а хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Структура



Структура типичного составного объекта могла бы выглядеть так:



Участники

Э Component (Graphic) - компонент:

- объявляет интерфейс для компонуемых объектов;
- предоставляет подходящую реализацию операций по умолчанию, общую для всех классов;
- объявляет интерфейс для доступа к потомкам и управления ими;
- определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;

а Leaf (Rectangle, Line, Text, и т.п.) - лист:

- представляет листовые узлы композиции и не имеет потомков;
- определяет поведение примитивных объектов в композиции;

а Composite (Picture) - составной объект:

- определяет поведение компонентов, у которых есть потомки;
- хранит компоненты-потомки;
- реализует относящиеся к управлению потомками операции в интерфейсе класса Component;

а 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, выполняя некоторые дополнительные операции до или после перенаправления.

Результаты

Паттерн компонент:

- а 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 примитивного объекта, может работать и с составным;
- а 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;
- а облегчает добавление новых видов компонентов. Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;

а способствует созданию общего дизайна. Однако такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого следует проводить проверки во время выполнения.

Реализация

При реализации паттерна компоновщик приходится рассматривать много вопросов:

а 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 поддержать паттерн цепочки обязанностей.

Обычно ссылку на родителя определяют в классе Component. Классы Leaf и Composite могут унаследовать саму ссылку и операции с ней.

При наличии ссылки на родителя важно поддерживать следующий инвариант: если некоторый объект в составной структуре ссылается на другой составной объект как на своего родителя, то для последнего первый является потомком. 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 и, значит, инвариант будет поддерживаться автоматически;

а разделение компонентов. Часто бывает полезно разделять компоненты, например для уменьшения объема занимаемой памяти. Но если у компонента может быть более одного родителя, то разделение 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 показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения родителей. Работает он в тех случаях, когда потомки могут не посыпать сообщений своим родителям, вынеся за свои границы часть внутреннего состояния;

а 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 подклассах операции. Класс Component поддерживает много операций, не имеющих смысла для класса Leaf. Как же тогда предоставить для них реализацию по умолчанию?

Иногда, проявив изобретательность, удается перенести в класс Component операцию, которая, на первый взгляд, имеет смысл только для составных объектов. Например, интерфейс для доступа 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 мы можем определить в классе Component операцию доступа 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 для управления потомками довольно хлопотны, мы обсудим их в следующем разделе;

- а *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?

Решая этот вопрос, мы должны выбирать между безопасностью и прозрачностью:

- если определить интерфейс для управления потомками в корне иерархии классов, то мы добиваемся прозрачности, так как все компоненты удается трактовать единообразно. Однако расплачиваться приходится безопасностью, поскольку клиент может попытаться выполнить бессмысленное действие, например добавить или удалить объект из листового узла;
- если управление потомками сделать частью класса Composite, то безопасность удастся обеспечить, ведь любая попытка добавить или удалить объекты из листьев в статически типизированном языке вроде C++ будет перехвачена на этапе компиляции. Но прозрачность мы утрачиваем, ибо у листовых и составных объектов оказываются разные интерфейсы.

В паттерне компоновщик мы придаём особое значение прозрачности, а не безопасности. Если для вас важнее безопасность, будьте готовы к тому, что иногда вы можете потерять информацию о типе и придется преобразовывать компонент 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 нулевой указатель. А в классе Composite эта операция переопределена и возвращает указатель this на сам объект:

```

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); // не добавит лист
}

```

Аналогичные проверки на принадлежность классу `Composite` в C++ выполняют и с помощью оператора `dynamic_cast`.

Разумеется, при таком подходе мы не обращаемся со всеми компонентами единообразно, что плохо. 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 появится новая проблема: нельзя реализовать `Component` : : `Add` так, чтобы она никогда не приводила к ошибке. 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 клиент может быть не рассчитанным на это.

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 «удаления». Если компонент хранит ссылку на родителя, то можно было бы считать, что Component::Remove удаляет самого себя. Но для операции Add по-прежнему нет разумной интерпретации;

а *должен ли Component реализовывать список компонентов.* Может возникнуть желание определить множество потомков в виде переменной экземпляра класса Component, в котором объявлены операции доступа и управления потомками. Но размещение указателя на потомков в базовом классе приводит к непроизводительному расходу памяти во всех листовых узлах, хотя у листа потомков быть не может. Такой прием можно применить, только если в структуре не слишком много потомков;

а *упорядочение потомков.* Во многих случаях порядок следования потомков составного объекта важен. 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;

а *кэширование для повышения производительности.* Если приходится часто обходить композицию или производить в ней поиск, то класс Composite может кэшировать информацию об обходе и поиске. Кэшировать разрешается либо 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 выборе эта информация позволила бы пропускать тех потомков, которые не видимы в текущем окне.

Любое изменение компонента должно делать кэши всех его родителей недействительными. 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 их кэшей;

а *кто должен удалять компоненты.* В языках, где нет сборщика 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 разделяться;

а какая структура данных лучше всего подходит для хранения компонентов. Составные объекты могут хранить своих потомков в самых разных структурах данных, включая связанные списки, деревья, массивы и хэш-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 стереокомпоненты, 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<Equipment*>* 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 C) для доступа 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, СБИС 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<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char* );
private:
    List<Equipment*> _equipment;
};
```

CompositeEquipment определяет операции для доступа и управления внутренними аппаратными блоками. Операции Add и Remove добавляют и удаляют оборудование из списка, хранящегося в переменной-члене _equipment. Операция CreateIterator возвращает итератор (точнее, экземпляр класса List Iterator), который будет обходить этот список.

Подразумеваемая реализация операции Net Price могла бы использовать CreateIterator для суммирования цен на отдельные блоки:¹

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

¹ Очень легко забыть об удалении итератора после завершения работы с ним. При обсуждении паттерна итератор рассказано, как защититься от таких ошибок.

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();  
};
```

Мы можем аналогично 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.Sin Floppy"));  
  
cout << "Полная стоимость равна " << chassis->NetPrice() << endl;
```

Известные применения

Примеры паттерна компоновщик можно найти почти во всех объектно-ориентированных системах. Первоначально класс View в схеме модель/вид/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/вид/контроллер была пересмотрена, в нее ввели класс Visual-Component, 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 которых 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:

- а примитивные присваивания, которые выполняют операцию над двумя регистрами и сохраняют результат в третьем;
- а присваивание, у которого есть исходный, но нет целевого регистра. Следовательно, регистр используется после возврата из процедуры;
- а 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 описывает, как можно компоновать и упорядочивать объекты Command с помощью класса компоновщика MacroCommand.

Родственные паттерны

Отношение компонент-родитель используется в паттерне цепочка обязанностей.

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 позволяет разделять компоненты, но ссылаясь на своих родителей они уже не могут.

Итератор можно использовать для обхода составных объектов.

Посетитель локализует операции и поведение, которые в противном случае пришлось бы распределять между классами Composite и Leaf.

Паттерн Decorator

Название и классификация паттерна

Декоратор - паттерн, структурирующий объекты.

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Известен также под именем

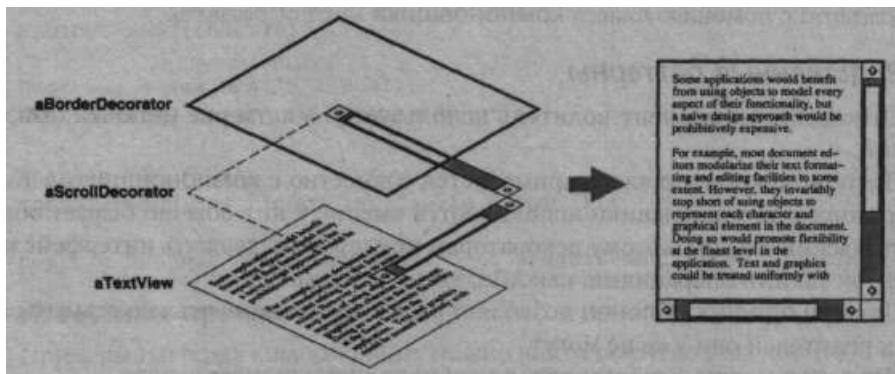
Wrapper (обертка).

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).

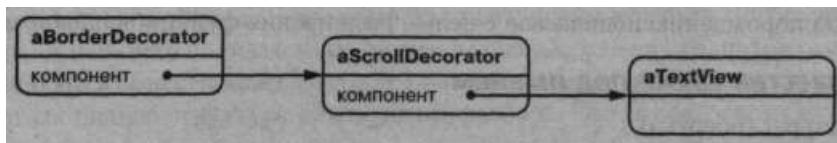
Добавить новые обязанности допустимо с помощью наследования. При наследовании классу с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако это решение статическое, а значит, недостаточно гибкое. Клиент не может управлять оформлением компонента рамкой.

Более гибким является другой подход: 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 любое число новых обязанностей.



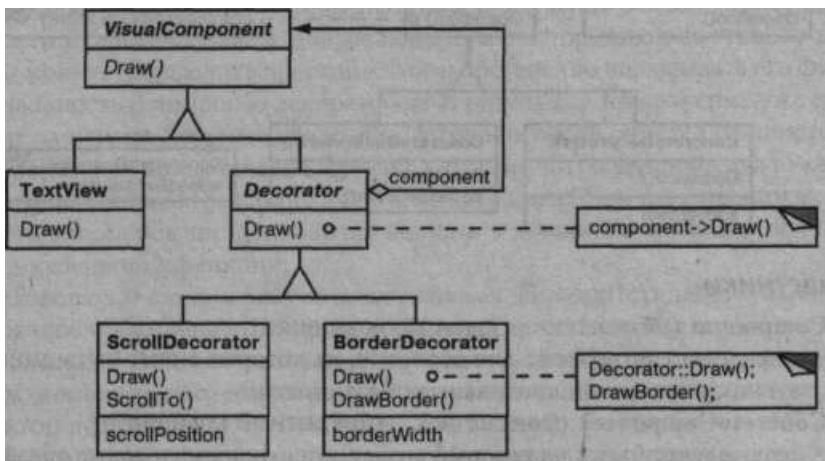
Предположим, что имеется объект класса *Text View*, который отображает текст в окне. По умолчанию *Text View* не имеет полос прокрутки, поскольку они не всегда нужны. Но при необходимости их удастся добавить с помощью декоратора *ScrollDecorator*. Допустим, что еще мы хотим добавить жирную сплошную рамку вокруг объекта *Text View*. Здесь может помочь декоратор *BorderDecorator*. Мы просто компонуем оба декоратора с *BorderDecorator* и получаем искомый результат.

Ниже на диаграмме показано, как композиция объекта *Text View* с объектами *BorderDecorator* и *ScrollDecorator* порождает 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.



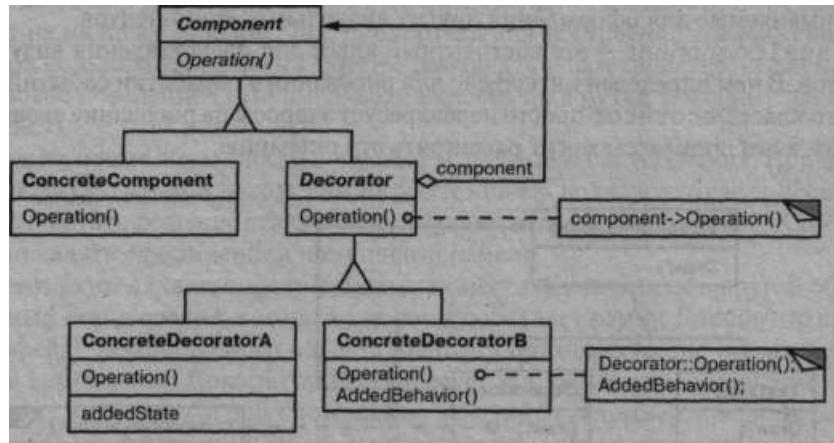
Подклассы Decorator могут добавлять любые операции для обеспечения необходимой функциональности. Так, операция ScrollTo объекта ScrollDecorator позволяет другим объектам выполнять прокрутку, если им известно о присутствии объекта ScrollDecorator. Важная особенность этого паттерна состоит в том, что декораторы могут употребляться везде, где возможно появление самого объекта VisualComponent. Поэтому клиент не может отличить декорированный объект от недекорированного, а значит, и никоим образом не зависит от наличия или отсутствия оформлений.

Применимость

Используйте паттерн декоратор:

- а для динамического, прозрачного для клиентов добавления обязанностей объектам;
- а для реализации обязанностей, которые могут быть сняты с объекта;
- а когда 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 комбинаторному 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

- a Component** (VisualComponent) - компонент:
 - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- a ConcreteComponent** (TextView) - конкретный компонент:
 - определяет объект, на который возлагаются дополнительные обязанности;
- a Decorator** - декоратор:
 - хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;
- a 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:

- а 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. При использовании же наследования требуется создавать новый класс для каждой дополнительной обязанности (например, `BorderedScrollView`, `BorderedTextView`), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких

декораторов к одному компоненту позволяет произвольным образом сочетать обязанности.

Декораторы 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;

- а *позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.* Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддерживать 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 платит за неиспользуемые функции. 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;
- а *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;
- а *множество мелких объектов.* При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

Реализация

Применение паттерна декоратор требует рассмотрения нескольких вопросов:

- а *соответствие интерфейсов.* Интерфейс декоратора должен соответствовать интерфейсу декорируемого компонента. Поэтому классы ConcreteDecorator должны наследовать общему классу (по крайней мере, в C++);
- а *отсутствие абстрактного класса Decorator.* Нет необходимости определять абстрактный класс Decorator, если планируется 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;
- а *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. 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;

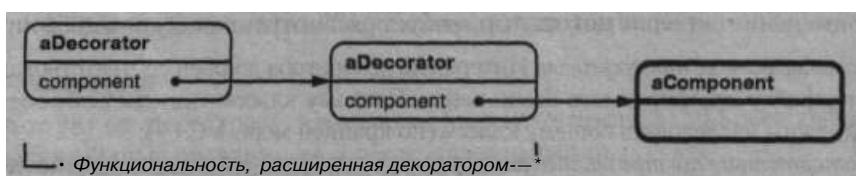
- a *изменение облика, а не внутреннего устройства объекта.* Декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива - изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн стратегия.

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 функциональностициальному объекту-стратегии, поэтому изменить или расширить поведение компонента допустимо, заменив этот объект.

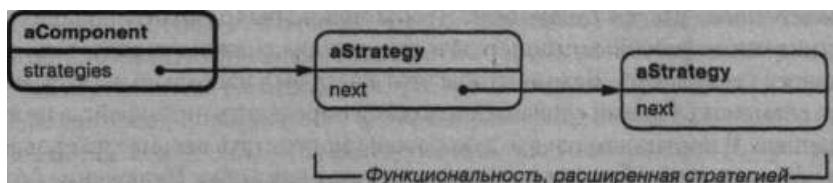
Например, мы можем поддержать разные стили рамок, поручив рисование рамки специальному объекту Border. Объект Border является примером объекта-стратегии: в данном случае он инкапсулирует стратегию рисования рамки. Число стратегий может быть любым, поэтому эффект такой же, как от рекурсивной вложенности декораторов.

Например, в системах MacApp 3.0 [App89] и Bedrock [Sym93a] графические компоненты, называемые видами (views), хранят список объектов-оформителей (adraer), которые могут добавлять различные оформления вроде границ к виду. Если к виду присоединены такие объекты, он дает им возможность выполнить свои функции. MacApp и Bedrock вынуждены предоставить доступ к этим операциям, поскольку класс View весьма тяжел. 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 модифицировать компонент, 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 видов, но и для расширения особенностей поведения объектов, связанных с обработкой событий. В обеих системах вид ведет список объектов поведения, которые могут модифицировать и перехватывать события. Каждому зарегистрированному объекту поведения вид 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

В следующем примере показано, как реализовать декораторы пользовательского интерфейса в программе на C++. Мы будем предполагать, 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 можно реализовать аналогично.

Теперь нам удастся скомпоновать экземпляры этих классов для получения различных оформлений. Ниже показано, как использовать декораторы для создания прокручиваемого компонента TextView с рамкой.

Во-первых, нужен какой-то способ поместить визуальный компонент в оконный объект. Предположим, что в нашем 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 - подкласс VisualComponent, значит, мы могли бы поместить его в окно:

```
window->SetContents(textView);
```

Но нам нужно поле ввода с рамкой и возможностью прокрутки. Поэтому предварительно мы его надлежащим образом оформим:

```
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 [РагЭО]. Другие варианты применения паттерна декоратор - это класс DebuggingGlyph из библиотеки Interviews и PassivityWrapper из ParcPlace Smalltalk. DebuggingGlyph печатает отладочную информацию до и после того, как переадресует запрос на размещение своему компоненту. Эта информация может быть полезна для анализа и отладки стратегии размещения объектов в сложном контейнере. Класс PassivityWrapper позволяет разрешить или запретить взаимодействие компонента с пользователем.

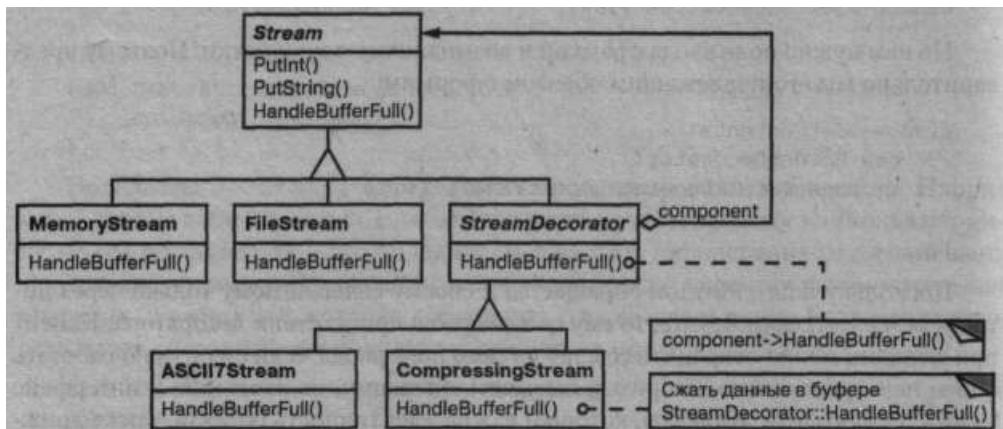
Но применение паттерна декоратор никоим образом не ограничивается графическими интерфейсами пользователя, как показывает следующий пример, основанный на потоковых классах из каркаса ET++ [WGM88].

Поток - это фундаментальная абстракция в большинстве 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 нам хотелось бы еще уметь:

а компрессировать данные в потоке, применяя различные алгоритмы сжатия (кодирование повторяющихся серий, алгоритм Лемпеля-Зива и т.д.);

а преобразовывать данные в 7-битные символы кода ASCII для передачи по каналу связи.

Паттерн декоратор позволяет весьма элегантно добавить такие обязанности потокам. На диаграмме 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.

Ключевым здесь является 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** конвертирует 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: если декоратор изменяет только обязанности объекта, но не его интерфейс, то адаптер придает объекту совершенно новый интерфейс.

Компоновщик: 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: декоратор позволяет изменить внешний облик объекта, стратегия - его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

Паттерн Facade

Название и классификация паттерна

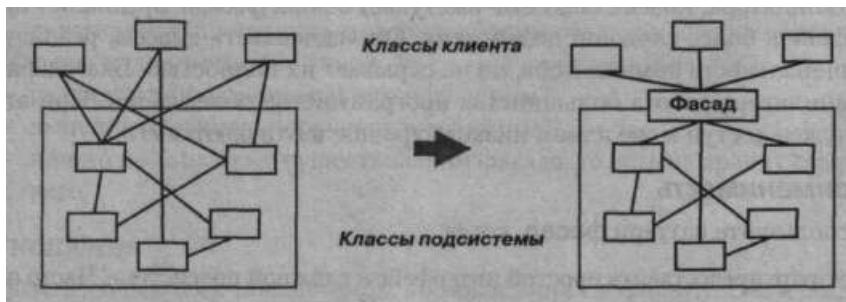
Фасад - паттерн, структурирующий объекты.

Назначение

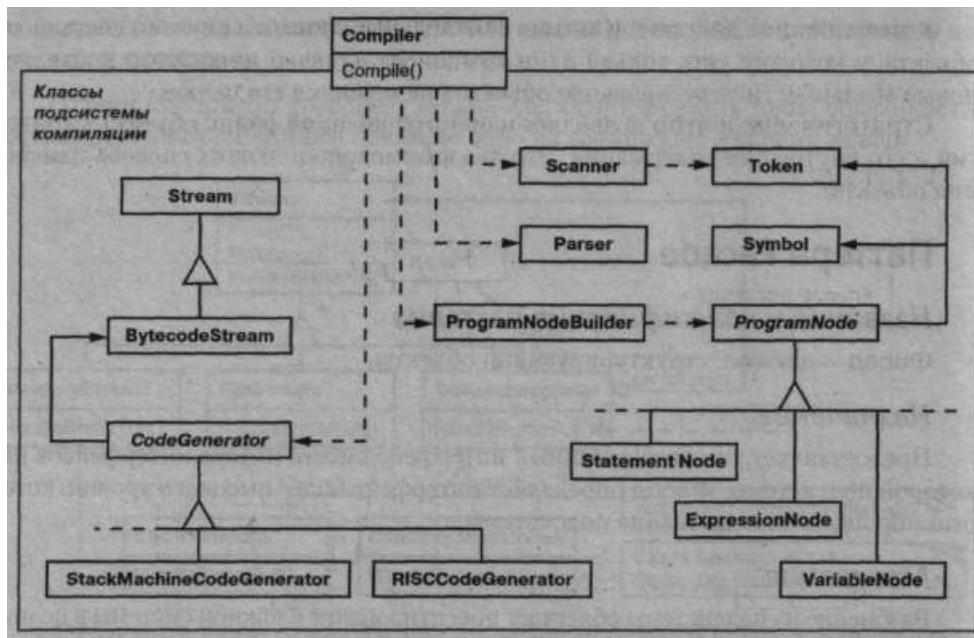
Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

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, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.



Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как Scanner (лексический анализатор), Parser (синтаксический анализатор), ProgramNode (узел программы), BytecodeStream (поток байтовых кодов) и ProgramNodeBuilder (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.



Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен также класс **Compiler** (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс **Compiler** выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.

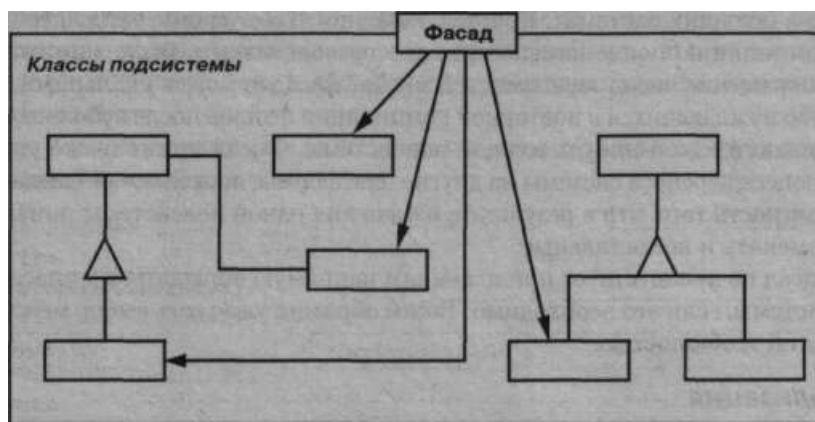
Применимость

Используйте паттерн фасад, когда:

- а хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- а между клиентами и классами реализации абстракции существует много зависимостей. Фасад позволяет отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;

а вы хотите разложить подсистему на отдельные слои. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Структура



Участники

a Facade (Compiler) - фасад:

- «знает», каким классам подсистемы адресовать запрос;
- делегирует запросы клиентов подходящим объектам внутри подсистемы;

a 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, посыпая 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

У паттерна фасад есть следующие преимущества:

- а изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;

- а 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 слои 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;
- а 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:
- а *уменьшение степени 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. Это 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;
 - а *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 системе компиляции классы Parser и Scanner - часть открытого интерфейса.
 - Делать классы подсистемы закрытыми иногда полезно, но это поддерживается немногими объектно-ориентированными языками. И в C++, и в Smalltalk для классов традиционно использовалось глобальное пространство имен.

Однако комитет по стандартизации C++ добавил к языку пространства имен [Str94], и это позволило разрешать доступ только к открытым классам подсистемы.

Пример кода

Рассмотрим более подробно, как возвести фасад вокруг 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(Scanners, ProgramNodeBuilder&);  
};
```

Parser вызывает ProgramNodeBuilder для инкрементного построения дерева. Взаимодействие этих классов описывается паттерном строитель:

```
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. Класс ProgramNode определяет интерфейс для манипулирования узлом программы и его потомками, если таковые имеются:

```

class ProgramNode {
public:
    // манипулирование узлом программы
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // манипулирование потомками
    virtual void Add(ProgramNode* );
    virtual void Remove(ProgramNode* );
    // ...

    virtual void Traverse(CodeGenertor);
protected:
    ProgramNode();
};

```

Операция Traverse (обход) принимает объект CodeGenerator (кодогенератор) в качестве параметра. Подклассы ProgramNode используют этот объект для генерации машинного кода в форме объектов Bytecode, которые помещаются в поток BytecodeStream. Класс CodeGenerator описывается паттерном посетитель:

```

class CodeGenerator {
public:
    virtual void Visit(StatementNode* );
    virtual void Visit(ExpressionNode* );
    // ...
protected:
    CodeGenerator(BytecodeStream& );
protected:
    BytecodeStream& _output;
};

```

У CodeGenerator есть подклассы, например StackMachineCodeGenerator и RISCCodeGenerator, генерирующие машинный код для различных аппаратных архитектур.

Каждый подкласс ProgramNode реализует операцию Traverse и обращается к ней для обхода своих потомков. Каждый потомок рекурсивно делает то же самое для своих потомков. Например, в подклассе ExpressionNode (узел выражения) операция Traverse определена так:

```
void ExpressionNode::Traverse (CodeGenerator& eg) {  
    eg.Visit(this);  
  
    ListIterator<ProgramNode*> i(_children);  
  
    for (i.First (); !i.IsDone () ; i.Next ()) {  
        i.CurrentItem()->Traverse(eg);  
    }  
}
```

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 генерации 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 есть всего одна такая архитектура. 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

инстанцировании объекта 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 - представление упрощенного интерфейса для наиболее распространенного случая.

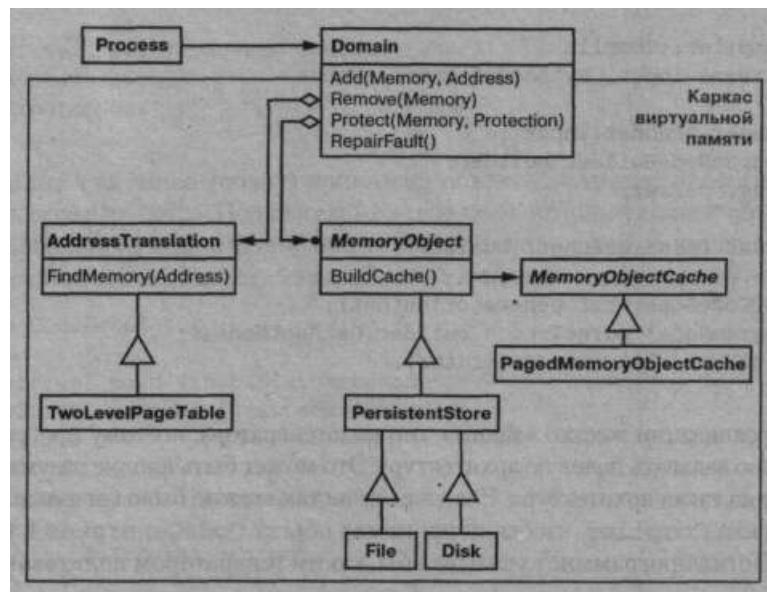
Известные применения

Пример компилятора в разделе «Пример кода» навеян идеями из системы компиляции языка ObjectWorks\Smalltalk [ParЭО].

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 связности 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 платформы. У двух таких подсистем есть «представители», то есть фасады. Они называются *FileSystemInterface* (память) и *Domain* (адресные пространства).



Например, для каркаса виртуальной памяти фасадом служит Domain. Класс Domain представляет 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:

- Q MemoryObject представляет объекты данных;
- а MemoryObjectCache кэширует данные из объектов MemoryObjects в физической памяти. MemoryObjectCache - это не что иное, как объект Стратегия, в котором локализована политика кэширования;
- а AddressTranslation инкапсулирует особенности оборудования трансляции адресов.

Операция RepairFault вызывается при возникновении ошибки из-за отсутствия страницы. Domain находит объект в памяти по адресу, где произошла ошибка и делегирует операцию RepairFault кэшу, ассоциированному с этим объектом. Поведение объектов Domain можно настроить, заменив их компоненты.

Родственные паттерны

Паттерн абстрактная фабрика допустимо использовать вместе с фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, не зависимым от этих подсистем. Абстрактная фабрика может выступать и как альтернатива фасаду, чтобы скрыть платформенно-зависимые классы.

Паттерн посредник 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 обмениваются информацией именно с ним, а не напрямую между собой. Напротив, 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 требуется только один фасад. Поэтому объекты фасадов часто являются одиночками.

Паттерн Flyweight

Название и классификация паттерна

Приспособленец - паттерн, структурирующий объекты.

Назначение

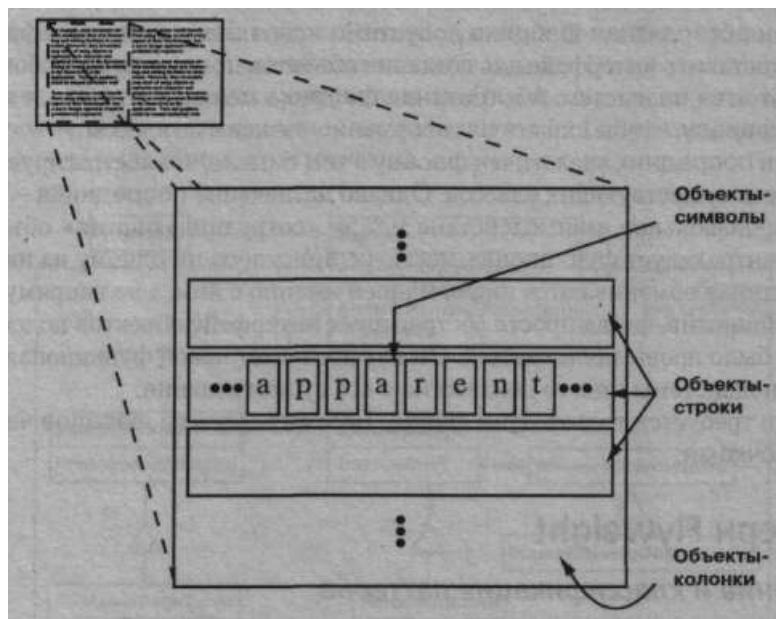
Использует разделение для эффективной поддержки множества мелких объектов.

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 используют объекты для представления каждого символа, несмотря на то что это увеличило бы гибкость на самых нижних уровнях приложения. Ведь тогда к рисованию и форматированию символов и встроенных элементов можно был бы применить единообразный подход. И для поддержки новых наборов символов не пришлось бы как-либо затрагивать остальные функции редактора. Да и 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 - стоимость. 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 разделять очень мелкие объекты без недопустимо высоких издержек.



Приспособленец - это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется. Приспособленцы не могут делать предположений о контексте, в котором работают.

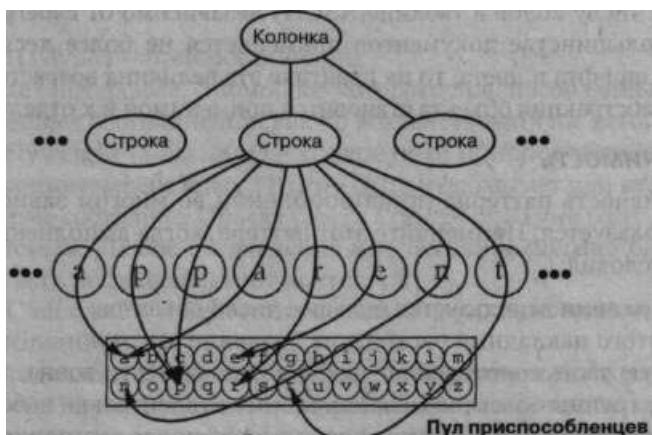
Ключевая идея здесь - различие между *внутренним* и *внешним* состояниями. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. 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.

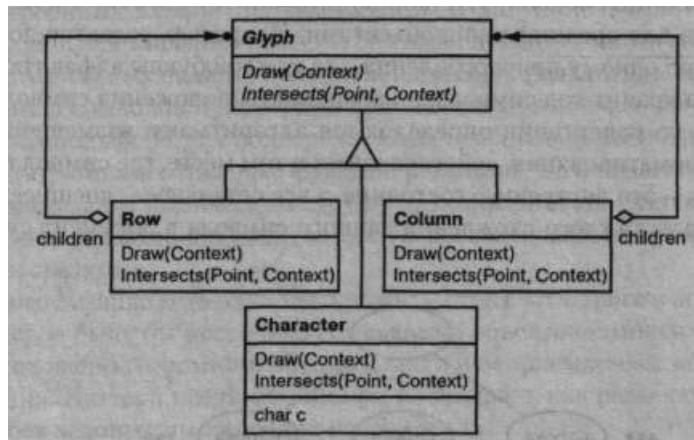
Логически для каждого вхождения данного символа в документ существует объект.



Физически, однако, есть 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 разделяемом 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 показать, чтобы это выглядело как горизонтальная строка. Поэтому вместе с запросом на рисование он может передавать каждому потомку координаты.

Поскольку число различных объектов-символов гораздо меньше, чем число 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.

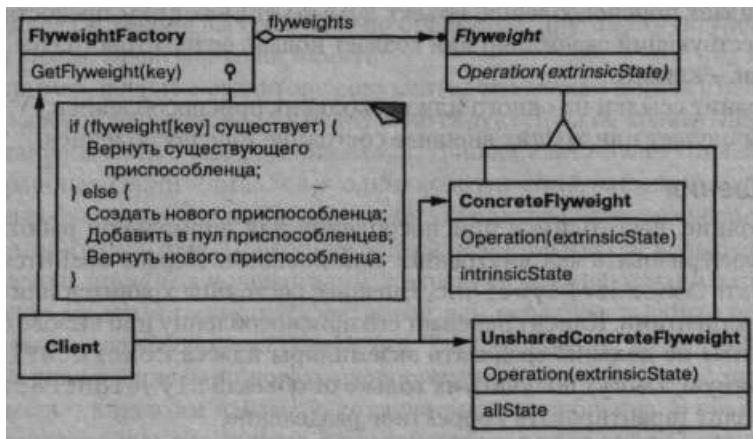
Применимость

Эффективность паттерна приспособленец во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены *все* нижеперечисленные условия:

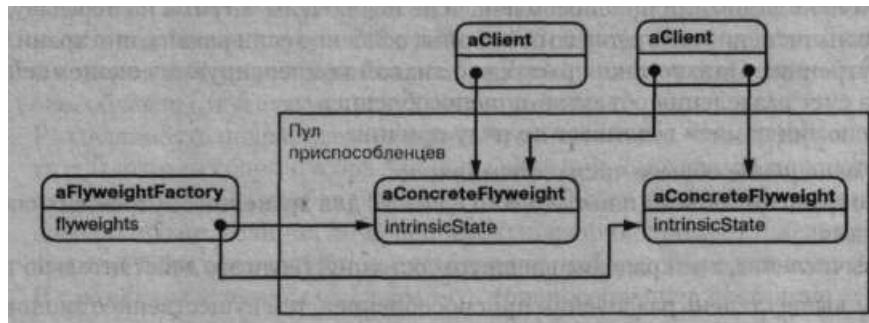
- а в приложении используется большое число объектов;
- а из-за этого накладные расходы на хранение высоки;
- а большую часть состояния объектов можно вынести вовне;
- а многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено;

а приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов.

Структура



На следующей диаграмме показано, как приспособленцы разделяются.



Участники

- a **Flyweight** (Glyph) - приспособленец:
 - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- a **ConcreteFlyweight** (Character) - конкретный приспособленец:
 - реализует интерфейс класса Flyweight и добавляет при необходимости внутреннее состояние. Объект класса ConcreteFlyweight должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
- a **UnsharedConcreteFlyweight** (Row, Column) - неразделяемый конкретный приспособленец:
 - не все подклассы Flyweight обязательно должны быть разделяемыми. Интерфейс Flyweight допускает разделение, но не наязывает 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`;

a FlyweightFactory - фабрика приспособленцев:

- создает объекты-приспособленцы и управляет ими;
- обеспечивает должное разделение приспособленцев. Когда клиент запрашивает приспособленца, объект `FlyweightFactory` предоставляет существующий экземпляр или создает новый, если готового еще нет;

a Client - клиент:

- хранит ссылки на одного или нескольких приспособленцев;
- вычисляет или хранит внешнее состояние приспособленцев.

Отношения

а состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте `ConcreteFlyweight`. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций;
 а клиенты не должны создавать экземпляры класса `ConcreteFlyweight` напрямую, а могут получать их только от объекта `FlyweightFactory`. Это позволит гарантировать корректное разделение.

Результаты

При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния, особенно если раньше оно хранилось как внутреннее. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев.

Экономия памяти возникает по ряду причин:

- а уменьшение общего числа экземпляров;
- а сокращение объема памяти, необходимого для хранения внутреннего состояния;
- а вычисление, а не хранение внешнего состояния (если это действительно так).

Чем выше степень разделения приспособленцев, тем существеннее экономия. С увеличением объема разделяемого 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 разделение 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 разделяемыми листовыми узлами. Из-за разделения указатель на родителя не может храниться в листовом узле-приспособлении, а должен передаваться ему как часть внешнего состояния. Это оказывает заметное влияние на способ взаимодействия объектов иерархии между собой.

Реализация

При реализации приспособленца следует обратить внимание на следующие вопросы:

а *вынесение внешнего состояния*. Применимость паттерна в значительной степени зависит от того, насколько легко идентифицировать внешнее состояние и вынести его за пределы разделяемых объектов. Вынесение внешнего состояния не уменьшает стоимости хранения, если различных внешних состояний так же много, как и объектов до разделения. Лучший вариант – внешнее состояние вычисляется по объектам с другой структурой, требующей значительно меньшей памяти.

Например, в нашем редакторе документов мы можем поместить карту с типографской информацией в отдельную структуру, а не хранить шрифт и начертание вместе с каждым символом. Данная карта будет отслеживать непрерывные серии символов с одинаковыми типографскими атрибутами. Когда объект-символ изображает себя, он получает типографские атрибуты от алгоритма обхода. Поскольку обычно в документах используется немногого разных шрифтов и начертаний, то хранить эту информацию отдельно от объекта-символа гораздо эффективнее, чем непосредственно в нем;

а *управление разделяемыми объектами*. Так как объекты разделяются, клиенты не должны инстанцировать их напрямую. Фабрика Flyweight Factory позволяет клиентам найти подходящего приспособленца. В объектах этого класса часто есть хранилище, организованное в виде ассоциативного массива, 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.

Разделяемость подразумевает также, что имеется некоторая форма подсчета ссылок или сбора 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. Логически глифы – это составные объекты, которые обладают графическими атрибутами и умеют изображать себя (см. описание паттерна компоновщик). 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 просто код символа:
```

```

class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);

private:
    char _charcode;
};

};

Чтобы не выделять память для шрифта каждого глифа, будем хранить этот атрибут во внешнем объекте класса GlyphContext. Данный объект 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 операциями обхода и манипулирования потомками обновляется GlyphContext:
```

```

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);

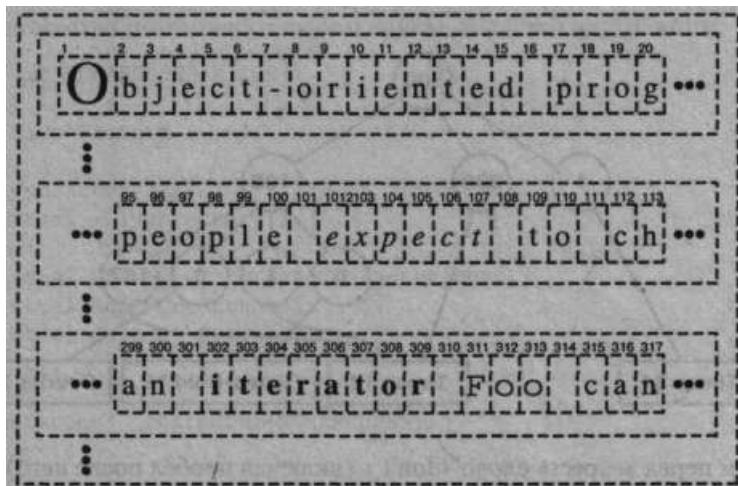
private:
    int _index;
    BTTree* _fonts;
};

};

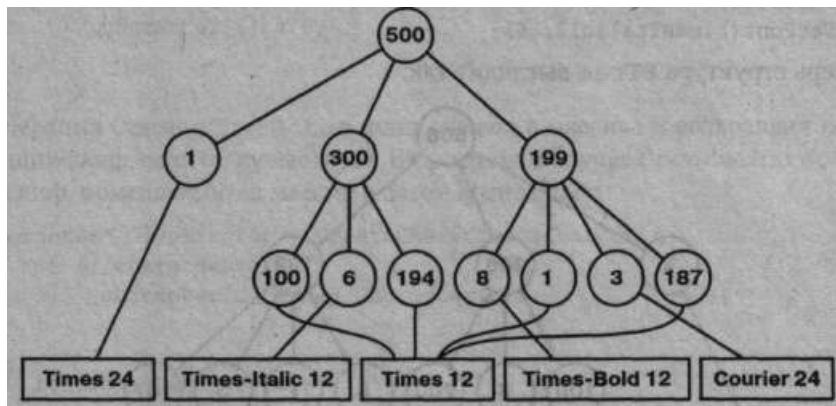
Объекту GlyphContext должно быть известно о текущем положении в структуре глифов во время ее обхода. Операция GlyphContext: .-Next увеличивает переменную _index по мере обхода структуры. Подклассы класса Glyph, имеющие потомков (например, Row и Column), должны реализовывать операцию Next так, чтобы она вызывала GlyphContext: :Next в каждой точке обхода.
```

Операция GlyphContext::GetFont использует переменную _index в качестве ключа для структуры BTtree, в которой хранится отображение 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.



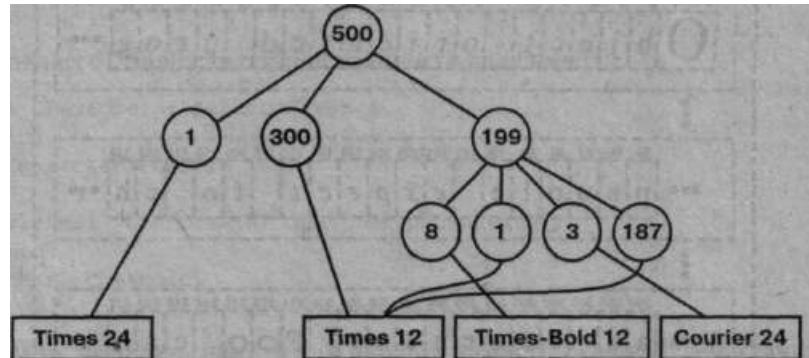
Структура BTtree, в которой хранится информация о шрифтах, может выглядеть так:



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 «ехрест» таким же, как у близлежащего текста (то есть times 12 - экземпляр класса Font для шрифта Times Roman размером 12 пунктов):

```
GlyphContext gc;
Font* timesl2 = new Font("Times-Roman-12");
Font* timesltalic12 = new Font("Times-Italic-12");
// ...
gcSetFont(timesl2, 6);
```

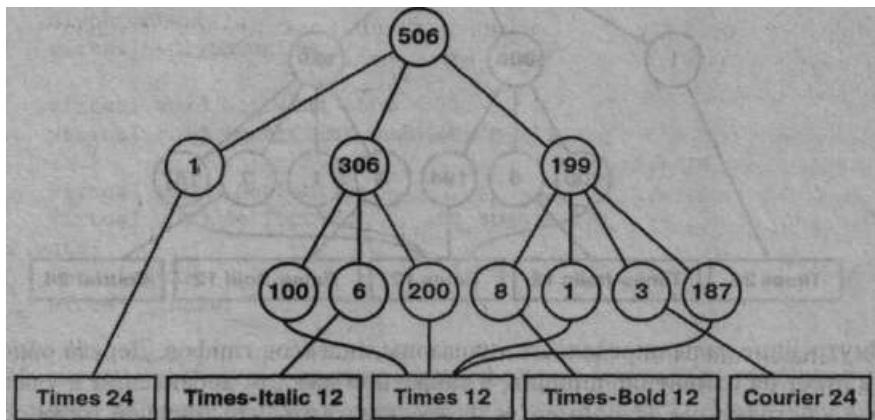
Новая структура BTee выглядит так (изменения выделены более ярким цветом):



Добавим перед «expect» слово «don't » (включая пробел после него), написанное шрифтом Times Italic размером 12 пунктов. В предположении, что текущей позиции все еще соответствует индекс 102, следующий код проинформирует объект gc об этом:

```
gc.Insert(6);
gcSetFont(timesltalic12, 6);
```

Теперь структура BTee выглядит так:



При запрашивании 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 мал по сравнению с размером структуры глифов. Это позволяет уменьшить расходы на хранение без заметного увеличения времени поиска.¹

И наконец, нам нужна еще фабрика FlyweightFactory, которая создает глифы и обеспечивает их корректное разделение. Класс GlyphFactory создает объекты Character и глифы других видов. Разделению подлежат 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* CreateRow0 ;
    virtual Column* CreateColumn0 ;
    // ...
private:
    Character* _character [NCHARCODES] ;
};
```

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 разделяются:

¹ Время поиска в этой схеме пропорционально частоте смены шрифта. Наименьшая производительность бывает, когда смена шрифта происходит на каждом символе, но на практике это бывает редко.

```

Row* GlyphFactory::rCreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}

```

Эти операции можно было бы опустить и позволить клиентам инстанцировать неразделяемые глифы напрямую. Но если позже мы решим сделать разделяемыми и их тоже, то придется изменять клиентский код, в котором они создаются.

Известные применения

Концепция объектов-приспособленцев впервые была описана и использована как техника 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 таблицу 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 разделение символов-приспособленцев весьма эффективно. В типичном случае для документа из 180 тысяч знаков необходимо создать только 480 объектов-символов.

В каркасе ET++ [WGM88] приспособленцы используются для поддержки независимости от внешнего облика.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 работу о своем расположении и изображениициальному объекту Layout. Изменение этого объекта ведет к изменению внешнего облика даже во время выполнения.

Для каждого класса виджета имеется соответствующий класс Layout (например, ScrollbarLayout, MenubarLayout и т.д.). В данном случае очевидная проблема состоит в том, что удваивается число объектов пользовательского интерфейса, ибо для каждого интерфейсного объекта есть дополнительный объект Layout. Чтобы избавиться от расходов, объекты Layout реализованы 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, который необходим для изображения объекта.

¹ В приведенном выше примере кода информация о стиле вынесена наружу, так что внутреннее состояние – это только код символа.

² Другой подход к обеспечению независимости от внешнего облика см. в описании паттерна абстрактная фабрика.

Объекты Layout создаются и управляются объектами класса Look. Класс Look - это абстрактная фабрика, 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). Таким образом, мы имеем пример объекта-стратегии, реализованный в виде приспособленца.

Родственные паттерны

Паттерн приспособленец часто используется в сочетании с компоновщиком для реализации иерархической структуры в виде ациклического направленного графа с разделяемыми листовыми вершинами.

Часто наилучшим способом реализации объектов состояния и стратегии является паттерн приспособленец.

Паттерн Proxy

Название и классификация паттерна

Заместитель - паттерн, структурирующий объекты.

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Известен также под именем

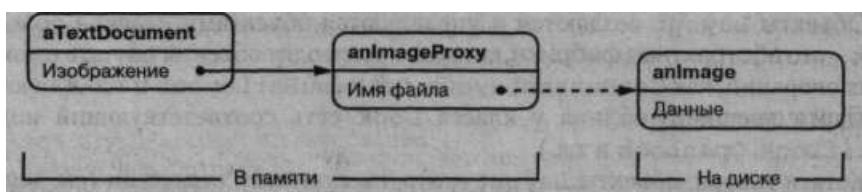
Surrogate (суррогат).

Мотивация

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да 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, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

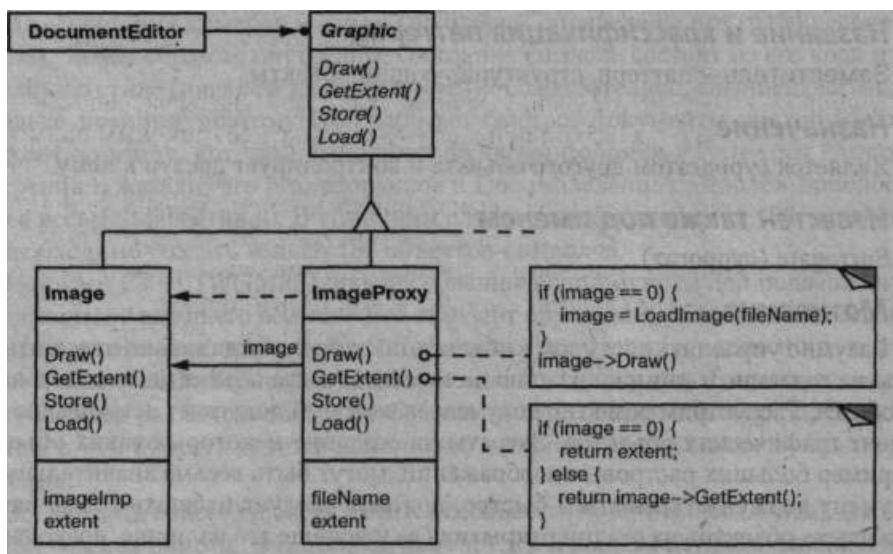
Решение состоит в том, чтобы использовать другой объект - заместитель изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.



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 инстанцируя изображение.

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 инстанцирует реальное изображение. Операцией Draw гарантируется, что изображение будет создано до того, как заместитель переадресует ему запрос. Операция Get Extent переадресует запрос

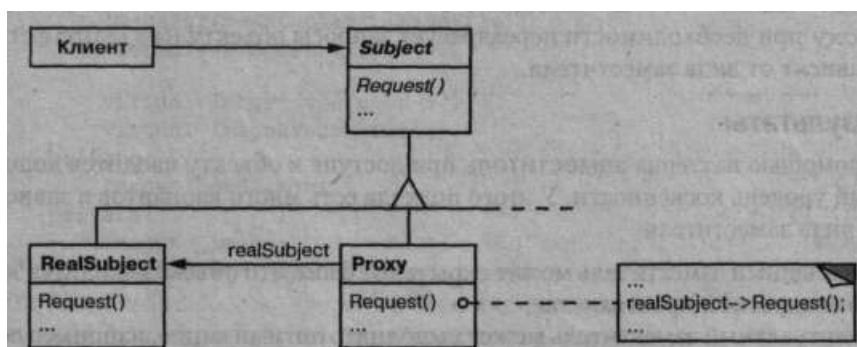
изображению, только если оно уже инстанцировано; в противном случае ImageProxy возвращает размеры, которые хранит сам.

Применимость

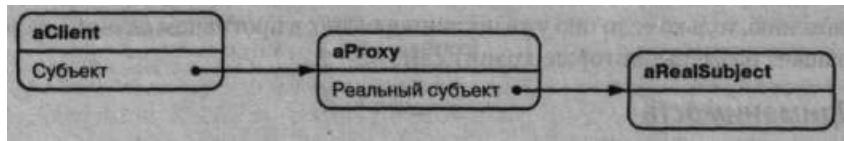
Паттерн заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель. Вот несколько типичных ситуаций, где заместитель оказывается полезным:

- а **удаленный заместитель** предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве. В системе NEXTSTEP [Add94] для этой цели применяется класс NXProxy. Заместителя такого рода Джеймс Коплиен [Cop92] называет «послом»;
- а **виртуальный заместитель** создает «тяжелые» объекты по требованию. Примером может служить класс ImageProxy, описанный в разделе «Мотивация»;
- а **захищающий заместитель** контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа. Например, в операционной системе Choices [CIRM93] объекты Kernel Proxy ограничивают права доступа к объектам операционной системы;
- а **«умная» ссылка** - это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применением 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 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 диаграмма объектов для структуры с заместителем во время выполнения.



Участники

a 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;

a Subject (Graphic) - субъект:

- определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;

a 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:

- а удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- а виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- а 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 дорогая операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс копирования, применив 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:

а *перегрузку оператора доступа к членам в C++*. Язык C++ поддерживает перегрузку оператора доступа 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 = LoadAnlmaeFile(_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, что они не являются частью интерфейса 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 - лучшее решение далеко не для всех видов заместителей. Некоторым из них должно быть точно известно, какая операция вызывается, а в таких случаях перегрузка оператора доступа не работает. Рассмотрим 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 для его автома[^]тической генерации используют препроцессор;

а метод `doesNotUnderstand` в `Smalltalk`. В языке `Smalltalk` есть возможность, позволяющая автоматически поддерживать переадресацию запросов. При отправлении клиентом сообщения, для которого у получателя нет соответствующего метода, `Smalltalk` вызывает метод `doesNotUnderstand`: `aMessage`.

Заместитель может переопределить `doesNotUnderstand` так, что сообщение будет переадресовано субъекту.

Дабы 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 суперкласса¹.

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` (следовательно, могущей затронуть заместителей), является тождество `==`.

Если вы собираетесь применять `doesNotUnderstand`: для реализации заместителя, то должны как-то решить вышеописанную проблему. Нельзя же ожидать, что совпадение заместителей - это то же самое, что и совпадение реальных субъектов. К сожалению, `doesNotUnderstand`: изначально создавался для обработки ошибок, а не для построения заместителей, поэтому его быстродействие оставляет желать лучшего;

а *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 реального субъекта `RealSubject`; заместитель может обращаться к любому из них единообразно. Но если заместитель должен инстанцировать 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 еще не инстанцированному субъекту. Некоторые заместители должны обращаться к своим субъектам вне зависимости от того, где они находятся - диске или в памяти. Это означает, что нужно использовать какую-то форму не зависящих от адресного пространства идентификаторов объектов. В разделе «Мотивация» для этой цели использовалось имя файла.

Пример кода

В коде реализовано два вида заместителей: виртуальный, описанный в разделе «Мотивация», и реализованный с помощью метода `doesNotUnderstand`:²

а *виртуальный заместитель*. В классе `Graphic` определен интерфейс для графических объектов:

```
class Graphic {
public:
    virtual ~Graphic();
```

¹ Эта техника 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.

² Еще один вид заместителя дает паттерн итератор.

```

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 файлов изображений. В нем замещена операция HandleMouse, посредством которой пользователь может интерактивно изменять размер изображения:

```

class Image : public Graphic {
public:
    Image(const char* file); // загрузка изображения из файла
    virtual ~Image();

    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* fileName);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Points 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 члены `_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 члены:

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Наконец, предположим, что есть класс `Text Document` для представления документа, который может содержать объекты класса `Graphic`:

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

```
void Insert(Graphic* );
// ...
};
```

Мы можем вставить объект ImageProxy в документ следующим образом:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(newImageProxy("anImageFileName"));
```

а *заместители, использующие метод doesNotUnderstand*. В языке Smalltalk можно создавать обобщенных заместителей, определяя классы, для которых нет суперкласса¹, 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: - он способен выполнить произвольную обработку. Например, можно было бы создать защищающего заместителя, определив набор legalMessages-сообщений, которые следует принимать, и снабдив заместителя следующим методом:

```
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 допустимо. Если это не так, doesNotUnderstand: посыпает 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.

Практически для любого класса Object является суперклассом самого верхнего уровня. Поэтому выражение «нет суперкласса» означает то же самое, что «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 посыпает их обратно объекту NXProxy.

В работе McCullough [McC87] обсуждается применение заместителей в Smalltalk для доступа к удаленным объектам. Джеки Пэско (Geoffrey Pascoe) [Pas86] описывает, как обеспечить побочные эффекты при вызове методов и реализовать контроль доступа с помощью «инкапсуляторов».

Родственные паттерны

Паттерн адаптер 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.

Несколько замечаний относительно декоратора. 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 то, что структурные паттерны похожи между собой, особенно когда речь идет об их участниках и взаимодействиях. Вероятное объяснение такому явлению: 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). Но имеющееся сходство может быть обманчиво, ибо с помощью разных паттернов можно решать совершенно разные задачи. В этом разделе сопоставлены группы структурных паттернов, и вы сможете яснее почувствовать их сравнительные достоинства и недостатки.

Адаптер и мост

У паттернов адаптер и мост 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.

Основное различие между адаптером и мостом в их назначении. Цель первого - устраниТЬ несовместимость 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 описанными различиями адаптер и мост часто используются в разные моменты жизненного цикла системы. Когда выясняется, что два несовместимых класса должны работать вместе, следует обратиться к адаптеру. Тем самым удастся 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 можно представлять себе как адаптер к набору других объектов. Но при такой интерпретации легко не заметить такой нюанс: фасад определяет *новый* интерфейс, тогда как адаптер повторно использует уже имеющийся. Подчеркнем, что адаптер заставляет работать вместе два *существующих* интерфейса, а не определяет новый.

Компоновщик, декоратор и заместитель

У компоновщика и декоратора аналогичные структурные диаграммы, свидетельствующие о том, 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 различные взаимосвязанные объекты удавалось трактовать единообразно, а несколько объектов рассматривать как один. Акцент здесь делается не на оформлении, а на представлении.

Указанные цели различны, но 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. Для декоратора компоновщик является конкретным компонентом. А для компоновщика декоратор - это листовый узел. Разумеется, их необязательно использовать вместе, и, как мы видели, цели данных паттернов различны.

Заместитель - еще один паттерн, структура которого напоминает декоратор. Оба они описывают, как можно предоставить косвенный доступ к объекту, и в реализации объектов-декораторов и заместителей хранится ссылка на другой объект, которому переадресуются запросы. Но и здесь цели различаются.

Как и декоратор, 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 основную функциональность, а заместитель разрешает или запрещает доступ к ней. В декораторе компонент обладает лишь частью функциональности, а остальное привносят один или несколько декораторов. Декоратор позволяет справиться с ситуацией, когда полную функциональность объекта нельзя определить на этапе компиляции или это по тем или иным причинам неудобно. Такая неопределенность делает рекурсивную композицию неотъемлемой частью декоратора. Для заместителя 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 сочетание разных паттернов невозможно. 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 способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В паттернах поведения уровня класса используется наследование - чтобы распределить поведение между разными классами. В этой главе описано два таких паттерна. Из них более простым и широко распространенным является шаблонный метод, который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм «обрастает мясом» за счет подклассов, где определены абстрактные операции. Другой паттерн поведения уровня класса - интерпретатор, который 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. Объекты-коллеги 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. Посредник, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрываения лишних связей.

Паттерн цепочка обязанностей позволяет и дальше уменьшать степень связанности. Он дает возможность посыпать 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 определяет и отвечает за зависимости между объектами. Классический пример наблюдателя встречается в схеме модель/вид/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 инкапсулирует алгоритм объекта,

упрощая его спецификацию и замену. Паттерн команда инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн состояние инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять поведение. Паттерн посетитель инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн итератор абстрагирует способ доступа и обхода объектов из некоторого агрегата.

Паттерн Chain of Responsibility

Название и классификация паттерна

Цепочка обязанностей - паттерн поведения объектов.

Назначение

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Мотивация

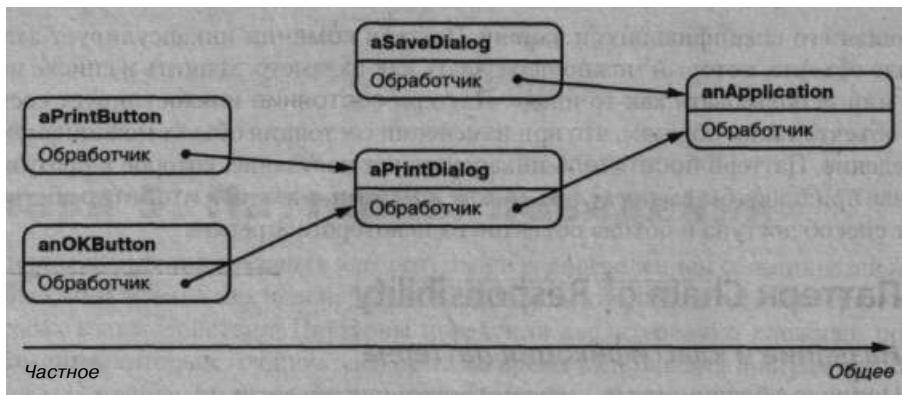
Рассмотрим контекстно-зависимую оперативную справку в графическом интерфейсе пользователя, который может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание справки зависит от того, какая часть интерфейса и в каком контексте выбрана. Например, справка 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.

Поэтому 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, каким именно - зависит от контекста и имеющейся в наличии информации.

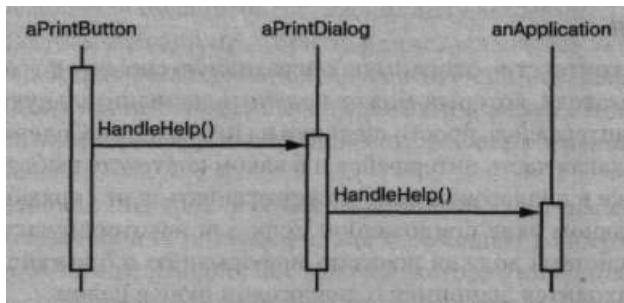
Проблема в том, что объект, *инициирующий* запрос (например, кнопка), не располагает информацией о том, какой объект в конечном итоге предоставит справку. Нам необходим какой-то способ отделить кнопку-инициатор запроса от объектов, владеющих справочной информацией. Как этого добиться, показывает паттерн цепочка обязанностей.

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

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



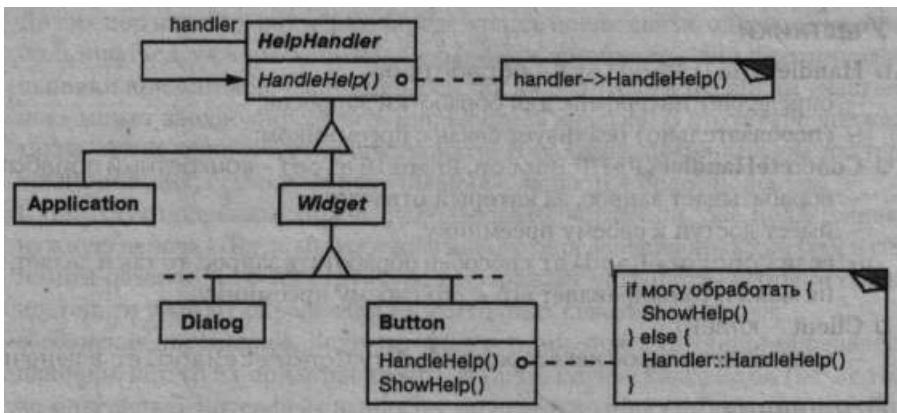
Предположим, что пользователь запрашивает справку по кнопке **Print** (печать). Она находится в диалоговом окне **PrintDialog**, содержащем информацию об объекте приложения, которому принадлежит (см. предыдущую диаграмму объектов). На представленной диаграмме взаимодействий показано, как запрос на получение 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 подмешиваемый класс (*mixin class*)) с операцией `HandleHelp`. Тогда классы, которые будут обрабатывать запрос, смогут его передать своему родителю.

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.

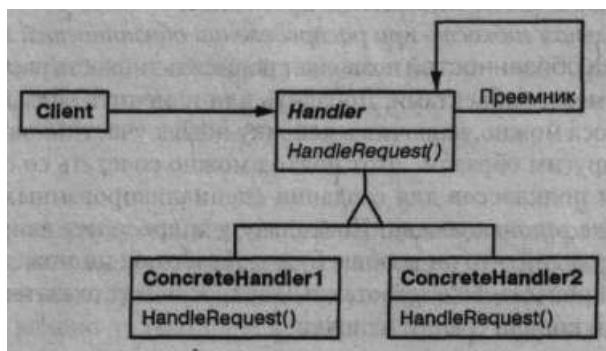


Применимость

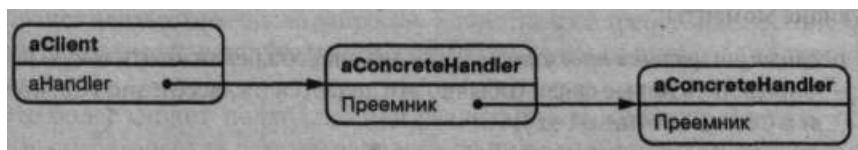
Используйте цепочку обязанностей, когда:

- а есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- а вы хотите отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- а набор объектов, способных обработать запрос, должен задаваться динамически.

Структура



Типичная структура объектов.



Участники

a Handler (HelpHandler) - обработчик:

- определяет интерфейс для обработки запросов;
- (необязательно) реализует связь с преемником;

a 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 - его своему преемнику;

a Client - клиент:

- отправляет запрос некоторому объекту ConcreteHandler в цепочке.

Отношения

Когда клиент инициирует запрос, он продвигается по цепочке, пока некоторый объект ConcreteHandler не возьмет на себя ответственность за его обработку.

Результаты

Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

а ослабление связности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту - о структуре цепочки.

Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;

а 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;

а 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 моменты:

а реализация цепочки преемников. Есть два способа реализовать такую цепочку:

- 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, ссылки на родителей обсуждаются более подробно.

Существующие связи можно использовать, когда они уже поддерживают нужную цепочку. Тогда мы избежим явного определения новых связей и сэкономим память. Но если структура не отражает устройства цепочки обязанностей, то уйти от определения избыточных связей не удастся;

- а *соединение преемников*. Если готовых ссылок, пригодных 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 подкласс ConcreteHandler не заинтересован в запросе, то ему и не надо замещать эту операцию, поскольку по умолчанию запрос как раз и отправляется дальше.

Вот 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();  
    }  
}
```

- а *представление запросов*. Представлять запросы можно по-разному. В простейшей форме, например в случае класса HandleHelp, запрос жестко кодируется как вызов некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе Handler.

Альтернатива – использовать одну функцию-обработчик, 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.

Это более гибкий подход, но при реализации нужно использовать условные операторы для раздачи запросов по их коду. 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 инкапсулированы параметры запроса. Класс Request может представлять некоторые запросы явно, а их новые типы описываются в подклассах. Подкласс может определить другие параметры. Обработчик должен иметь информацию о типе запроса (какой именно подкласс Request используется), чтобы разобрать эти параметры.

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 походящему 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. 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);  
}  
}
```

а *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-предемнику. Поэтому осуществлять перенаправление вручную необязательно. Класс обрабатывает только запросы, в которых заинтересован, и ожидает, что механизм `doesNotUnderstand` выполнит все остальное.

Пример кода

В следующем примере иллюстрируется, 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. (Можно было, конечно, построить реализацию и на основе подмешиваемого класса.)

```

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 принимает 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 в классе 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 отображает и поиск заканчивается:

```

Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // предложить 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 (HasHelpO) {
        // предложить 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:

```
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, касающиеся печати:

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2 ;
const Topic APPLICATIONJTOPIC = 3;

Application* application = new Application (APPLICATIONJTOPIC) ;
Dialog* dialog = new Dialog (application, PRINTJTOPIC) ;
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] Responder.

В каркасе графических редакторов Unidraw определены объекты Command, которые инкапсулируют запросы к объектам Component и ComponentView [VL90]. Объекты Command - это запросы, которые компонент или вид компонента могут интерпретировать как команду 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 вид 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. Гарантируется, что к тому моменту, как Window получит запрос, недействительный прямоугольник будет трансформирован 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. В этом случае родитель компонента может выступать в роли его преемника.

Паттерн Command

Название и классификация паттерна

Команда - паттерн поведения объектов.

Назначение

Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

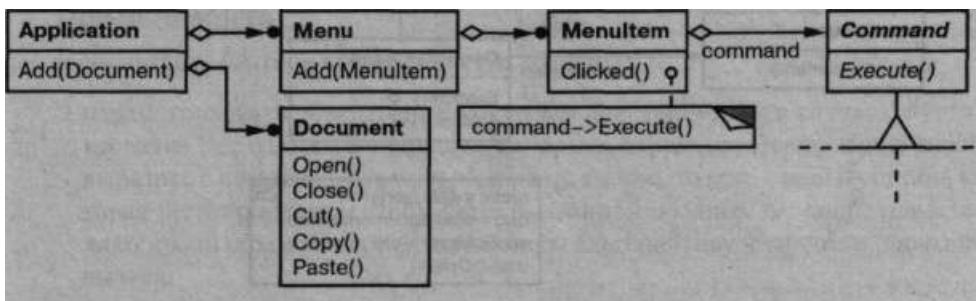
Известен также под именем

Action (действие), Transaction (транзакция).

Мотивация

Иногда необходимо посыпать 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 посыпают запрос в ответ на действие пользователя. Но в саму библиотеку не заложена возможность обрабатывать этот запрос, так как только приложение, использующее ее, располагает информацией о том, что следует сделать. Проектировщик библиотеки не владеет никакой информацией о получателе запроса и о том, какие операции тот должен выполнить.

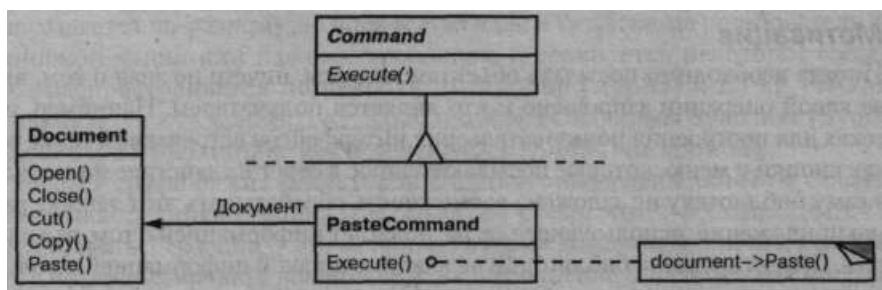
Паттерн команда позволяет библиотечным объектам отправлять запросы неизвестным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой. В основе списываемого паттерна лежит абстрактный класс Command, в котором объявлен интерфейс для выполнения 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 посыпала запрос. У получателя есть информация, необходимая для выполнения запроса.



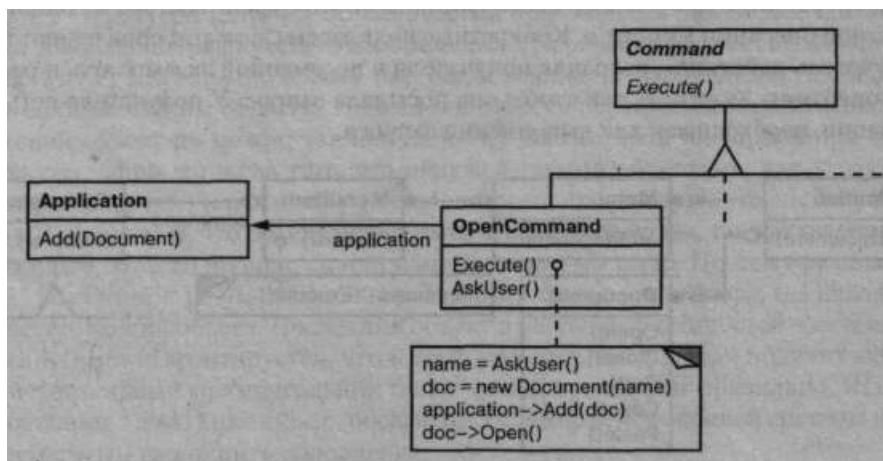
С помощью объектов Command легко реализуются меню. Каждый пункт меню - это экземпляр класса MenuItem. Сами меню и все их пункты создает класс Application наряду со всеми остальными элементами пользовательского интерфейса. Класс Application отслеживает также открытые пользователем документы.

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 Document, который был передан при создании объекта. Операция Execute вызывает операцию Paste документа-получателя.

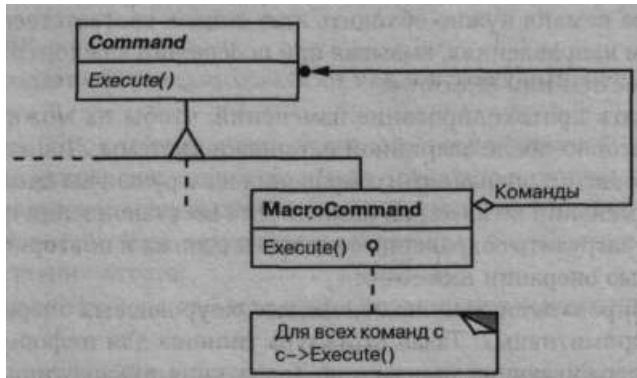


Для подкласса OpenCommand операция Execute ведет себя по-другому: она запрашивает у пользователя имя документа, создает соответствующий объект Document, извещает о новом документе приложение-получатель и открывает этот документ.



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 NormalizeCommand. 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, что в каждом из приведенных примеров паттерн команда отделяет объект, инициирующий операцию, от объекта, который «знает», как ее выполнить. 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 присвоить обоим элементам один и тот же экземпляр конкретного подкласса класса Command. Мы можем динамически подменять команды, что очень полезно для реализации контекстно-зависимых меню. Можно также поддерживать 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 его выполнить.

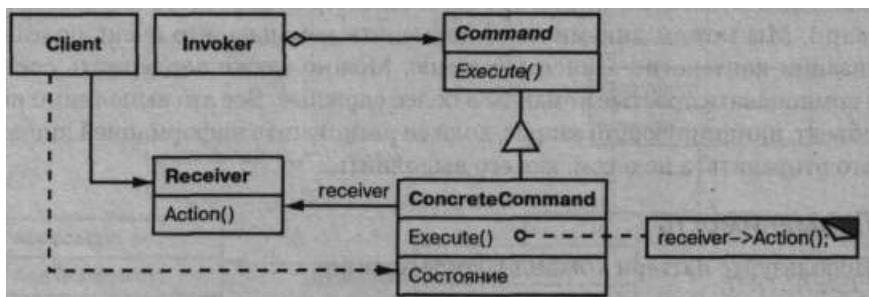
Применимость

Используйте паттерн команда, когда хотите:

- а параметризовать объекты 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;
- а определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта Command необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;

- а поддержать отмену операций. Операция Execute объекта Command может сохранить состояние, необходимое для отката 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;
- а поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. 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;
- а структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. 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

- a Command** - команда:
 - объявляет интерфейс для выполнения операции;
- a ConcreteCommand** (PasteCommand, OpenCommand) - конкретная команда:
 - определяет связь между объектом-получателем Receiver и действием;
 - реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
- a Client (Application)** - клиент:
 - создает объект класса ConcreteCommand и устанавливает его получателя;

а **Invoker** (MenuItem) - инициатор:

- обращается к команде для выполнения запроса;

а **Receiver** (Document, Application) - получатель:

- располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

Отношения

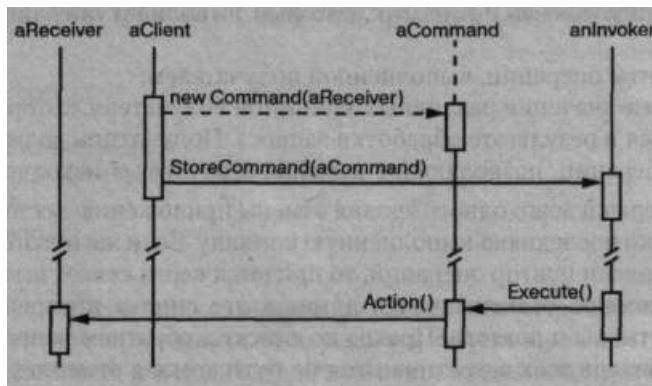
а клиент создает объект ConcreteCommand и устанавливает для него получателя;

а инициатор Invoker сохраняет объект ConcreteCommand;

а инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката;

а объект ConcreteCommand вызывает операции получателя для выполнения запроса.

На следующей диаграмме видно, как Command разрывает связь между инициатором и получателем (а также запросом, который должен выполнить последний).



Результаты

Результаты применения паттерна команда таковы:

- а команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить;
- а команды - это самые настоящие объекты. Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами;
- а из простых команд можно собирать составные, например класс MacroCommand, рассмотренный выше. В общем случае составные команды описываются паттерном компоновщик;
- а добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

Реализация

При реализации паттерна команда следует обратить внимание на следующие аспекты:

а *насколько «умной» должна быть команда*. У команды может быть широкий круг обязанностей. На одном полюсе стоит простое определение связи между получателем и действиями, которые нужно выполнить для удовлетворения запроса. На другом - реализация всего самостоятельно, 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; и *поддержка отмены и повтора операций*. Команды могут поддерживать отмену и повтор операций, если имеется возможность отменить результаты выполнения (например, операцию 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 откат результатов всех встретившихся ito пути команд отменяет их действие; проход в прямом направлении и выполнение встретившихся команд приводит к повтору действий.

Команду, допускающую отмену, возможно, 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. Поэтому 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);

- а *как избежать 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 необходимо 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;
- а *применение шаблонов в C++.* Для команд, которые не допускают отмену и не имеют аргументов, 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. Сначала абстрактный класс 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 в конструкторе передается объект Document, являющийся получателем:

```

class PasteCommand : public Command {
public:
    PasteCommand(Document* );
    virtual void Execute();
private:
    Document* „document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}

```

В случае с простыми командами, не допускающими отмены и не требующими аргументов, можно воспользоваться шаблоном класса для параметризации получателя. Определим для них шаблонный подкласс SimpleCoiranand, который параметризуется типом получателя Receiver и хранит связь между объектом-получателем и действием, представленным указателем на функцию-член:

```

template <class Receiver>
class SimpleCoiranand : 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<Receiver>::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 Macr@Command 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-член 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, что если бы в классе MacroCommand была реализована операция отмены Unexecute, то при ее выполнении подкоманды должны были бы отменяться в порядке, *обратном* тому, который применяется 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

Быть может, 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 всю функциональность команд. Есть и шаблон ActionCallback, параметризованный действием Action, который автоматически инстанцирует подклассы 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, 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 *функции* - объекты, ведущие себя, как функции [Cop92]. За счет перегрузки оператора вызова operator() он становится более понятным. 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 можно использовать для реализации макрокоманд.

Паттерн хранитель иногда проектируется так, что сохраняет состояние команды, необходимое для отмены ее действия.

Команда, которую нужно копировать перед помещением в список истории, ведет себя, как прототип.

Паттерн Interpreter

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 возникает часто, то имеет смысл представить ее конкретные проявления в виде предложений на простом языке. Затем можно будет 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, не проще ли построить алгоритм поиска так, чтобы он мог интерпретировать регулярное выражение, описывающее множество строк-образцов?

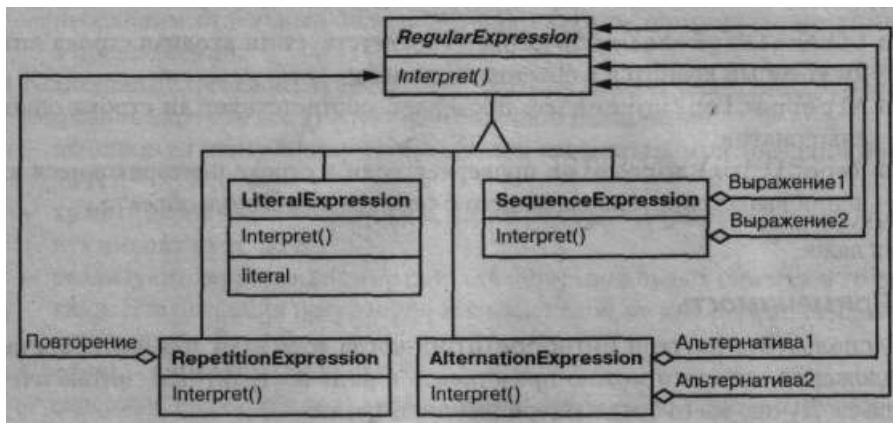
Паттерн интерпретатор определяет грамматику простого языка, представляя предложения на этом языке и интерпретирует их. Для приведенного примера паттерн описывает определение грамматики и интерпретации языка регулярных выражений.

Предположим, что они описаны следующей грамматикой:

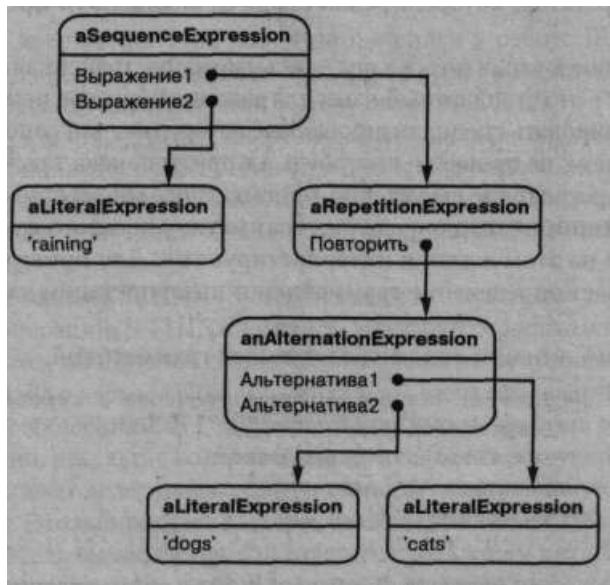
```
expression ::= literal | alternation | sequence | repetition |
             '(' expression ')'
alternation ::= expression | ' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

где **expression** - это 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) *`

Мы можем создать интерпретатор регулярных выражений, определив в каждом подклассе `RegularExpression` операцию `Interpret`, принимающую в качестве аргумента контекст, где нужно интерпретировать выражение. Контекст состоит из входной строки и информации о том, как далеко по ней мы уже продвинулись. В каждом подклассе `RegularExpression` операция `Interpret` производит со-поставление с оставшейся частью входной строки. Например:

- а `LiteralExpression` проверяет, соответствует ли входная строка литералу, который хранится в объекте подкласса;
- Q `AlternationExpression` проверяет, соответствует ли строка одной из альтернатив;
- Q `RepetitionExpression` проверяет, если в строке повторяющиеся вхождения выражения, совпадающего с тем, что хранится в объекте.

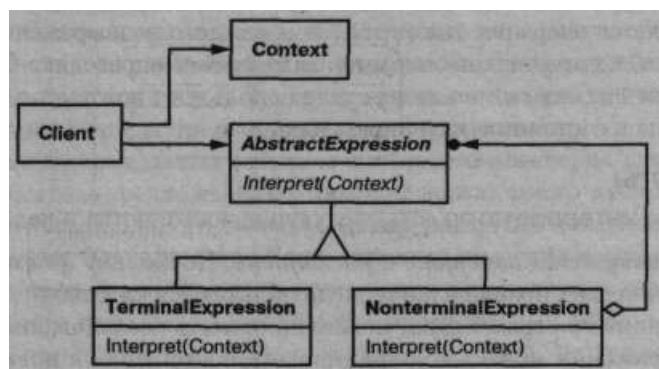
И так далее.

Применимость

Используйте паттерн интерпретатор, когда есть язык для интерпретации, предложения которого можно представить в виде абстрактных синтаксических деревьев. Лучше всего этот паттерн работает, когда:

а грамматика проста. Для сложных грамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять генераторы синтаксических анализаторов, поскольку они могут интерпретировать выражения, не строя абстрактных синтаксических деревьев, что экономит память, а возможно, и время;
 а 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 преобразуют в конечный автомат. Но даже в этом случае сам *транслятор* можно реализовать с помощью паттерна интерпретатор.

Структура



Участники

- a **AbstractExpression (RegularExpression)** - абстрактное выражение:
 - объявляет абстрактную операцию **Interpret**, общую для всех узлов в абстрактном синтаксическом дереве;
- a **TerminalExpression (LiteralExpression)** - терминальное выражение:
 - реализует операцию **Interpret** для терминальных символов грамматики;
 - необходим отдельный экземпляр для каждого терминального символа в предложении;
- Q **NonterminalExpression(AlternationExpression, RepetitionExpression, SequenceExpressions)** - нетерминальное выражение:
 - по одному такому классу требуется для каждого грамматического правила $R ::= R_1 \ R_2 \dots R_n$;
 - 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 $./?, \dots /?_n$;
- p **Context** - контекст:
 - содержит информацию, глобальную по отношению к интерпретатору;

a Client - клиент:

- строит (или получает в готовом виде) абстрактное синтаксическое дерево, представляющее отдельное предложение на языке с данной грамматикой. Дерево составлено из экземпляров классов Nonterminal-Expression и Terminal-Expression;
- вызывает операцию **Interpret**.

Отношения

- а клиент строит (или получает в готовом виде) предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов NonterminalExpression и Terminal-Expression. Затем клиент инициализирует контекст и вызывает операцию **Interpret**;
- а в каждом узле вида NonterminalExpression через операции **Interpret** определяется операция **Interpret** для каждого подвыражения. Для класса TerminalExpression операция **Interpret** определяет базу рекурсии;
- а операции **Interpret** в каждом узле используют контекст для сохранения и доступа к состоянию интерпретатора.

Результаты

У паттерна интерпретатор есть следующие достоинства и недостатки:

- а *грамматику легко изменять и расширять*. 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;
- а *простая реализация грамматики*. Реализации классов, описывающих узлы абстрактного синтаксического дерева, похожи. Такие классы легко кодировать, а зачастую их может автоматически генерировать компилятор или генератор синтаксических анализаторов;
- а *сложные грамматики трудно сопровождать*. 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 синтаксических анализаторов;
- а *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. Это поможет избежать изменения классов, описывающих грамматику.

Реализация

У реализаций паттернов интерпретатор и компоновщик есть много общего. Следующие вопросы относятся только к интерпретатору:

- а *создание абстрактного синтаксического дерева*. Паттерн интерпретатор не поясняет, как *создавать* дерево, то есть разбор выражения не входит в его задачу. Создать дерево разбора может таблично-управляемый или написанный вручную (обычно методом рекурсивного спуска) анализатор, а также сам клиент;
- а *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;
- а *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 попадаться много раз. 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. 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**, - безотносительно к тому, в каком месте дерева этот экземпляр встречается.

Пример кода

Мы приведем два примера. Первый - законченная программа на Smalltalk для проверки того, соответствует ли данная последовательность регулярному выражению. 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 'Γ' expression
sequence ::= expression '&' expression
repetition ::= expression 'repeat'
literal ::= 'a1' | 'b' | 'c1' | ... { 'a1' I 'b1' I 'c' I ... }*
```

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 мы употребляем 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 стр. 237. 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. В каждом из классов, входящих в абстрактное синтаксическое дерево, эта операция реализована. 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 & 'be'
```

строки "abc", "aabc", "aaabc" и т.д. Но при наличии 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 входят в его состояние `inputState`.

```
match: inputState
  ^ expression2 match: (expression1 match: inputState).
```

AlternationExpression возвращает результат, объединяющий состояния всех альтернатив. Вот определение match: для этого случая:

```
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
  I finalState tStream I
  finalState := Set new.
  inputState
  do:
    [:stream I 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 уверенными, что сопоставление с литералом никогда не изменяет входной поток. Это существенно, поскольку все альтернативы в `AlternationExpression` должны «видеть» одинаковые копии входного потока.

Таким образом, мы определили классы, составляющие абстрактное синтаксическое дерево. Теперь опишем, как его построить. Вместо того чтобы создавать анализатор 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 анализатор синтаксиса регулярных выражений.

Для построения дерева нам понадобятся операции "`I`", "`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 мы определили эти операции выше в иерархии классов, например `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 булевые переменные, то есть константы `true` и `false`. Нетерминальные

символы представляют выражения, содержащие операторы and, or и not. Приведем определение грамматики:¹

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |  
    (' BooleanExp ')'  
AndExp ::= BooleanExp 'and' BooleanExp  
OrExp ::= BooleanExp 'or' BooleanExp  
NotExp ::= 'not' BooleanExp  
Constant ::= 'true' | 'false'1  
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 мы подробно опишем только классы BooleanExp, VariableExp и AndExp. Классы OrExp и NotExp аналогичны классу AndExp. Класс Constant представляет булевые константы.

В классе BooleanExp определен интерфейс всех классов, которые описывают булевые выражения:

```
class BooleanExp {  
public:  
    BooleanExp();  
    virtual ~BooleanExp();  
    virtual bool Evaluate (Contextk) = 0;  
    virtual BooleanExp* Replace (const char*, BooleanExp&) = 0;  
    virtual BooleanExp* Copy () const = 0;  
};
```

Класс Context определяет отображение между переменными и булевыми значениями, которые в C++ представляются константами true и false. Интерфейс этого класса следующий:

```
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();
```

¹ Упрощая задачу, мы игнорируем приоритеты операторов и предполагаем, что их учет возложен на объект, строящий дерево разбора.

```

    virtual bool Evaluate(Contexts);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};

```

Конструктор класса принимает в качестве аргумента имя переменной:

```

VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}

```

Вычисление переменной возвращает ее значение в текущем контексте:

```

bool VariableExp::Evaluate (Contexts 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 у переменной то же имя, что было передано ранее в качестве аргумента:

```

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(Contexts);
    virtual BooleanExp* Replace(const char*, BooleanExpS);
    virtual BooleanExp* Copy0 const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

```

```
AndExp::AndExp (BooleanExp* op1 , BooleanExp* op2 ) {
    _operand1 = op1;
    _operand2 = op2;
}
```

При решении AndExp вычисляются его операнды и результат применения к ним операции логического И возвращается:

```
bool AndExp::Evaluate (Context^ aContext) {
    return
        _operand1->Evaluate(aContext)&&
        _operand2->Evaluate (aContext) ;
}
```

В классе AndExp операции Copy и Replace реализованы 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

$(\text{true} \text{ and } x) \text{ or } (y \text{ and } (\text{not } x))$

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 OrExpt
    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.

И наконец, мы можем заменить переменную у новым выражением и повторить вычисление:

```
VariableExp* z = new VariableExpf("Z") ;
NotExp not_z(z) ;
BooleanExp* replacement = expression->Replace("Y", not_z) ;
context.Assign(z, true) ;
result = replacement->Evaluate(context) ;
```

На этом примере проиллюстрирована важная особенность паттерна интерпретатор: «интерпретация» предложения может означать самые разные действия. 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 операцию Copy допустимо рассматривать 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 демонстрируют, что все три операции разрешается вынести в отдельный объект-посетитель «интерпретатор», тогда аналогия станет более очевидной.

Паттерн интерпретатор - это нечто большее, чем распределение некоторой операции по иерархии классов, составленной с помощью паттерна компоновщик. 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 считать такие операции, как Weight (вес) и Copy (копирование), интерпретаторами, несмотря на то что они распределены по всей иерархии классов, - просто мы не воспринимаем агрегаты автомобиля как язык. Тут все дело в точке зрения: опубликуй мы грамматику агрегатов автомобиля, 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 QOSA для разрешения ограничений 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 применять паттерн интерпретатор лучше в тех случаях, когда иерархию классов можно представлять себе как описание языка.

Родственные паттерны

Компоновщик: абстрактное синтаксическое дерево - это пример применения паттерна компоновщик.

Приспособленец показывает варианты разделения терминальных символов в абстрактном синтаксическом дереве.

Итератор: интерпретатор может пользоваться итератором для обхода структуры-

Посетителя можно использовать для инкапсуляции в одном классе поведения каждого узла абстрактного синтаксического дерева.

Паттерн Iterator

Название и классификация паттерна

Итератор - паттерн поведения объектов.

Назначение

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Известен также под именем

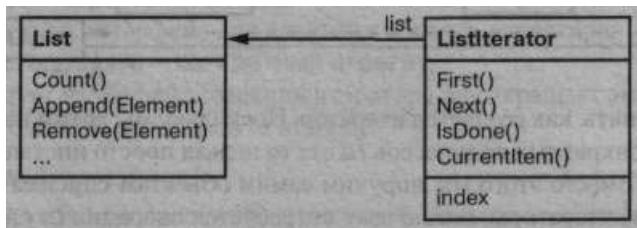
Cursor (курсор).

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 предвидеть заранее. Кроме того, иногда нужно, чтобы в один и тот же момент было определено несколько активных обходов списка.

Все это позволяет сделать паттерн итератор. Основная его идея в том, чтобы за доступ 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, класс `List` мог бы предусмотреть класс `ListIterator`.



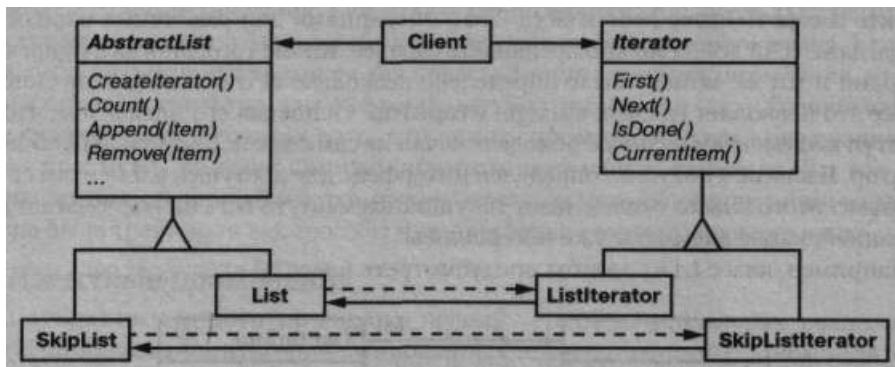
Прежде чем создавать экземпляр класса `ListIterator`, необходимо иметь список, подлежащий обходу. С объектом `ListIterator` вы можете последовательно посетить все элементы списка. Операция `Current Item` возвращает текущий элемент списка, операция `First` инициализирует текущий элемент первым элементом списка, `Next` делает текущим следующий элемент, а `IsDone` проверяет, не оказались ли мы за последним элементом, если да, то обход завершен.

Отделение механизма обхода от объекта `List` позволяет определять итераторы, реализующие различные стратегии обхода, не перечисляя их в интерфейсе класса `List`. Например, `FilteringListIterator` мог бы предоставлять доступ только к тем элементам, которые удовлетворяют условиям фильтрации.

Заметим: между итератором и списком имеется тесная связь, клиент должен иметь информацию, что он обходит именно *список*, а не какую-то другую агрегированную структуру. Поэтому клиент привязан к конкретному способу агрегирования. Было бы лучше, если бы мы могли изменять класс агрегата, не трогая код клиента. Это можно сделать, обобщив концепцию итератора и рассмотрев *полиморфную итерацию*.

Например, предположим, что у нас есть еще класс `SkipList`, реализующий список. Список с пропусками (skipList) [Pug90] - это вероятностная структура данных, по характеристикам напоминающая сбалансированное дерево. Нам нужно научиться 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 оказывается не зависящим от конкретных агрегированных классов.



Остается понять, как создается итератор. Поскольку мы хотим написать код, не зависящий от конкретных подклассов `List`, то нельзя просто инстанцировать конкретный класс. Вместо этого мы поручим самим объектам-спискам создавать для себя подходящие итераторы, вот почему потребуется операция `CreateIterator`, посредством которой клиенты смогут запрашивать объект-итератор.

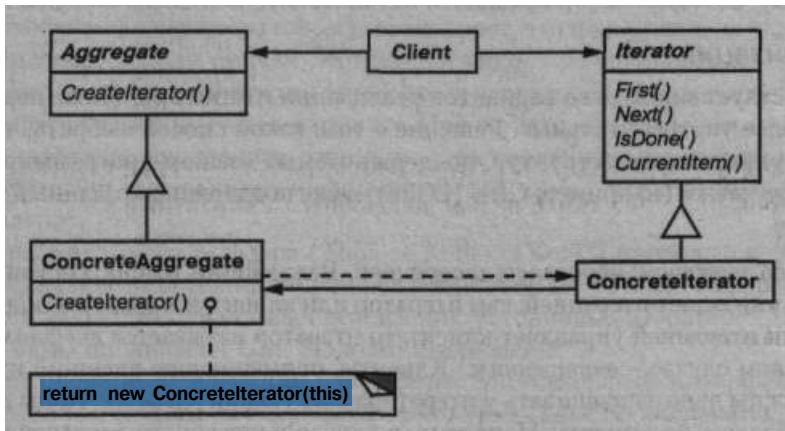
`CreateIterator` - это пример использования паттерна фабричный метод. В данном случае он служит для того, чтобы клиент мог запросить у объекта-списка подходящий итератор. Применение фабричного метода приводит к появлению двух иерархий классов - одной для списков, другой для итераторов. Фабричный метод `CreateIterator` «связывает» эти две иерархии.

Применимость

Используйте паттерн итератор:

- а для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
- а для поддержки нескольких активных обходов одного и того же агрегированного объекта;
- а для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).

Структура



Участники

a Iterator - итератор:

- определяет интерфейс для доступа и обхода элементов;

a Concretelterator - конкретный итератор:

- реализует интерфейс класса **Iterator**;
- следит за текущей позицией при обходе агрегата;

a Aggregate - агрегат:

- определяет интерфейс для создания объекта-итератора;

a ConcreteAggregate - конкретный агрегат:

- реализует интерфейс создания итератора и возвращает экземпляр подходящего класса **Concretelterator**.

Отношения

Concretelterator отслеживает текущий объект в агрегате и может вычислить идущий за ним.

1

Результаты

У паттерна итератор есть следующие важные особенности:

- а поддерживает различные виды 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`;

- а итераторы упрощают интерфейс класса `Aggregate`. Наличие интерфейса для обхода в классе `Iterator` делает излишним дублирование этого интерфейса в классе `Aggregate`. Тем самым интерфейс агрегата упрощается;
- а одновременно для данного агрегата может быть активно несколько обходов. Итератор следит за инкапсулированным в нем самом состоянием обхода. Поэтому одновременно разрешается осуществлять несколько обходов агрегата.

Реализация

Существует множество вариантов реализации итератора. Ниже перечислены наиболее употребительные. Решение о том, какой способ выбрать, часто зависит от управляющих структур, поддерживаемых языком программирования. Некоторые языки (например, CLU [LG86]) даже поддерживают данный паттерн напрямую.

- а 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 запрашивать у итератора следующий элемент, чтобы двигаться дальше по агрегату. Напротив, 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. Но, с другой стороны, внутренние итераторы проще в использовании, поскольку они вместо вас определяют логику обхода;
- а 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 мы называем *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

¹ Грэди 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 общих черт.

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;

- а *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 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 подправляет 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) [МигЭЗ] описывает реализацию устойчивых итераторов для класса *List* из библиотеки USL Standard Components;

- о *дополнительные операции итератора*. Минимальный интерфейс класса *Iterator* состоит из операций *First*, *Next*, *IsDone* и *CurrentItem*.¹ Но могут оказаться полезными и некоторые дополнительные операции. Например, упорядоченные агрегаты могут предоставлять операцию *Previous*, позиционирующую итератор на предыдущий элемент. Для отсортированных или индексированных коллекций интерес представляет операция *SkipTo*, которая позиционирует итератор на объект, удовлетворяющий некоторому критерию;

- а *использование полиморфных итераторов в C++*. С полиморфными итераторами связаны определенные накладные расходы. Необходимо, чтобы объект-итератор создавался в динамической памяти фабричным методом. Поэтому использовать их стоит только тогда, когда есть необходимость в полиморфизме. В противном случае применяйте конкретные итераторы, которые вполне можно распределять в стеке.

У полиморфных итераторов есть и еще один недостаток: за их удаление отвечает клиент. Здесь открывается большой простор для ошибок, так как очень легко забыть об освобождении распределенного из кучи объекта-итератора после завершения работы с ним. Особенно велика вероятность этого, если у операции есть несколько точек выхода. А в случае возбуждения

Этот интерфейс можно и еще уменьшить, если объединить операции *Next*, *IsDone* и *CurrentItem* в одну, которая будет переходить к следующему объекту и возвращать его. Если обход завершен, то эта операция вернет специальное значение (например, 0), обозначающее конец итерации.

исключения память, занимаемая объектом-итератором, вообще никогда не будет освобождена.

Эту ситуацию помогает исправить паттерн заместитель. Вместо настоящего итератора мы используем его заместителя, память для которого выделена в стеке. Заместитель уничтожает итератор в своем деструкторе. Поэтому, как только заместитель выходит из области действия, вместе с ним уничтожается и настоящий итератор. Заместитель гарантирует выполнение надлежащей очистки даже при возникновении исключений. Это пример применения хорошо известной в C++ техники, которая называется «выделение ресурса - это инициализация» [ES90]. В разделе «Пример кода» она проиллюстрирована подробнее;

a итераторы могут иметь привилегированный доступ. Итератор можно рассматривать как расширение создавший его агрегат. Итератор и агрегат тесно связаны. В C++ такое отношение можно выразить, сделав 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 того чтобы решить эту проблему, 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;

a итераторы для составных объектов. Реализовать внешние агрегаты для рекурсивно агрегированных структур (таких, например, которые возникают в результате применения паттерна компоновщик) 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 лучшее решение дает итератор курсорного типа. 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 можно поддержать отдельным итератором;

a пустые итераторы. Пустой итератор **NullIterator** - это вырожденный итератор, полезный при обработке граничных условий. По определению, **NullIterator** всегда считает, что обход завершен, то есть его операция **IsDone** неизменно возвращает истину.

Применение пустого итератора может упростить обход древовидных структур (например, объектов Composite). В каждой точке обхода мы запрашиваем у текущего элемента 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 C) и две реализации класса `Iterator`: одну для обхода списка от начала к концу, а другую - от конца к началу (в базовой библиотеке поддержан только первый способ). Затем мы покажем, 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.

а интерфейсы классов `List` и `Iterator`. Сначала обсудим ту часть интерфейса класса `List`, которая имеет отношение к реализации итераторов. Полный интерфейс см. в приложении C:

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

В открытом интерфейсе класса `List` предусмотрен эффективный способ поддержки итераций. Его достаточно для реализации обоих видов 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());
};
```

а 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` не вызывает затруднений. В нем хранится экземпляр `List` и индекс `_current`, указывающий текущую позицию в списке:

```
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 внутри списка:

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
```

Наконец, операция `Current Item` возвращает элемент, соответствующий текущему индексу. Если итерация уже завершилась, то мы возбуждаем исключение `IteratorOutOfBoundsException`:

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBoundsException();
    }
```

```
    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;

а *использование итераторов.* Предположим, что 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`, принимающую в качестве аргумента итератор. Она пользуется этим итератором для обхода и печати содержимого списка:

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.NextO) {
        i.Current!item()->Print();
    }
}
```

Поскольку у нас есть итераторы для обхода списка от начала к концу и от конца к началу, то мы можем повторно воспользоваться той же самой операцией для печати списка служащих в обоих направлениях:

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward (employees) ;
ReverseListIterator<Employee*> backward (employees) ;

PrintEmployees (forward) ;
PrintEmployees(backward),-
```

а *как избежать зависимости от 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 `SkipList Iterator`, 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);
```

Оптимальное решение в данной ситуации - вообще не привязываться к конкретной реализации списка, например `SkipList`. Мы можем 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 подмешиваемого класса `Traversable`, в котором определен интерфейс для создания итератора. Для поддержки полиморфных итераций агрегированные классы могут являться потомками `Traversable`.

Класс `List` замещает `CreateIterator`, для того чтобы возвратить объект `ListIterator`:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Теперь мы можем написать код для печати служащих, который не будет зависеть от конкретного представления списка:

```
// мы знаем только, что существует класс AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

а как гарантировать удаление итераторов? Заметим, что `CreateIterator` возвращает только что созданный в динамической памяти объект-итератор. Ответственность за его удаление лежит на нас. Если мы забудем это сделать, то возникнет утечка памяти. Для того чтобы упростить задачу клиентам, мы введем класс `IteratorPtr`, который замещает итератор. Он уничтожит объект `Iterator` при выходе из области определения.

Объект класса `IteratorPtr` всегда распределяется 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 можно рассматривать как указатель на итератор. Функции-члены класса `IteratorPtr` встраиваются, поэтому при их вызове накладных расходов нет:

```
template <class Item>
class IteratorPtr {
```

¹ Это можно проверять на этапе компиляции, если объявить операторы `new` и `delete` закрытыми. Реализовывать их при этом не надо.

```

public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    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);

```

а **внутренний ListIterator**. В последнем примере рассмотрим, как можно было бы реализовать внутренний или пассивный класс **ListIterator**. Теперь итератор сам управляет итерацией и применяет к каждому элементу некоторую операцию.

Нерешенным остается вопрос о 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. Существует, по крайней мере, два варианта: передать указатель на функцию (глобальную или статическую) или 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 аккумулировать некоторую информацию, а функции для этого плохо подходят — пришлось бы использовать статические переменные для запоминания состояния. Подкласс класса **Iterator** предоставляет удобное место для хранения аккумулированного состояния — переменную экземпляра. Но создавать подкласс для каждого вида обхода слишком трудоемко.

Вот набросок 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 принимает экземпляр **List** в качестве параметра. Внутри себя он использует внешний итератор **List Iterator** 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 печати первых десяти служащих из списка. С этой целью надо породить подкласс от **ListTraverser** и определить в нем операцию **Process Item**. Подсчитывать число напечатанных служащих будем в переменной 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 печатает первые 10 служащих:

```
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. Работы, правда, 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-члена **Test Item**, которая и реализует проверку условия. В подклассах **Test Item** замещается для задания конкретного условия. Посредством операции **Traverse** выясняется, нужно ли продолжать обход, основываясь на результате 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 есть в большинстве библиотек коллекций классов.

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 – отсутствует необходимость в фабричном методе, который запрашивал бы у очереди соответствующий ей итератор. Однако, чтобы итератор можно было реализовать эффективно, интерфейс абстрактного класса `Queue` должен быть 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 принимает блок (то есть замыкание). Каждый элемент коллекции привязывается к локальной переменной в блоке, а затем блок выполняется. Smalltalk также включает набор классов `Stream`, которые поддерживают похожий на итератор интерфейс. `ReadStream` – это, по существу, класс `Iterator` и внешний итератор для всех последовательных коллекций. Для непоследовательных коллекций типа `Set` и `Dictionary` нет стандартных итераторов.

Операция `Traverse` в этих примерах – это не что иное, как шаблонный метод с примитивными операциями `TestItem` и `ProcessItem`.

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 [Бог94] 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: итераторы довольно часто применяются для обхода рекурсивных структур, создаваемых компоновщиком.

Фабричный метод: полиморфные итераторы поручают фабричным методам инстанцировать подходящие подклассы класса **Iterator**.

Итератор может использовать хранитель для сохранения состояния итерации и при этом содержит его внутри себя.

Паттерн Mediator

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.

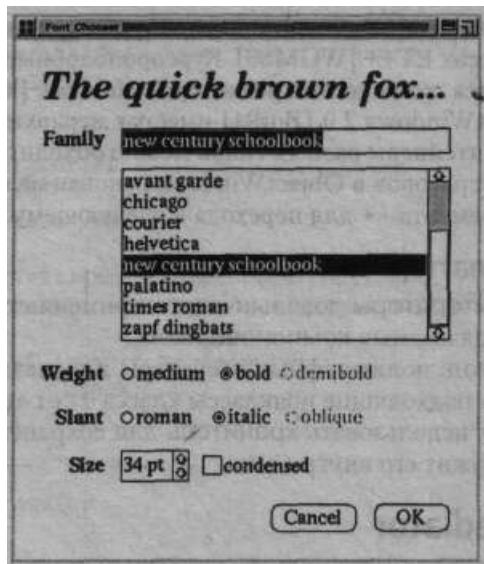
Мотивация

Объектно-ориентированное проектирование способствует распределению некоторого поведения между объектами. Но при этом в получившейся структуре объектов может возникнуть много связей или (в худшем случае) каждому объекту придется иметь информацию обо всех остальных.

Несмотря на то что разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако изобилие 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. Если вы предпримете подобную попытку, то для настройки поведения системы вам придется определять множество подклассов.

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 разными виджетами в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка недоступна. При выборе из списка может измениться содержимое поля ввода. И наоборот,

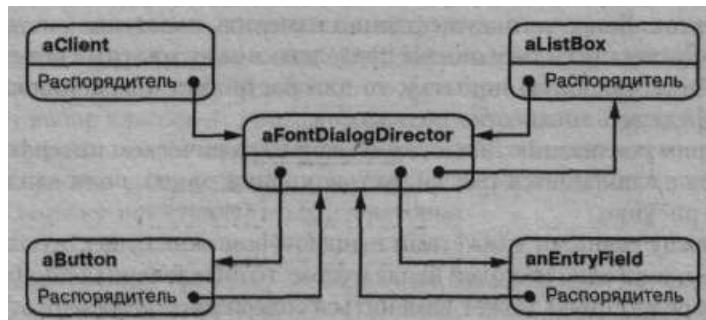


ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом, например изменить либо удалить его.

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.

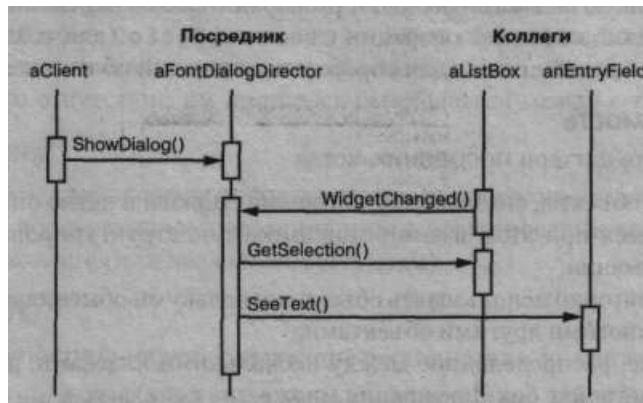
Всех этих проблем можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией 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 диаграмме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка.

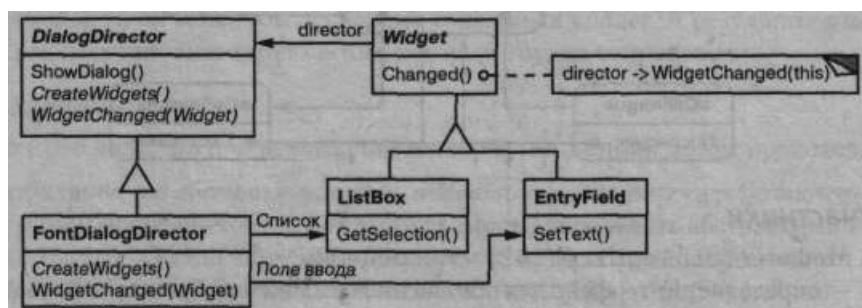


Последовательность событий, в результате которых информация о выбранном элементе списка передается в поле ввода, следующая:

1. Список информирует распорядителя 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, позволяющие выполнить определенное действие (например, изменить шрифт 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 через распорядителя. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. А коль скоро поведение локализовано в одном классе, то его несложно модифицировать или сделать совершенно другим путем расширения или замены этого класса.

Абстракцию FontDialogDirector можно было бы интегрировать в библиотеку классов так, как показано на рисунке.



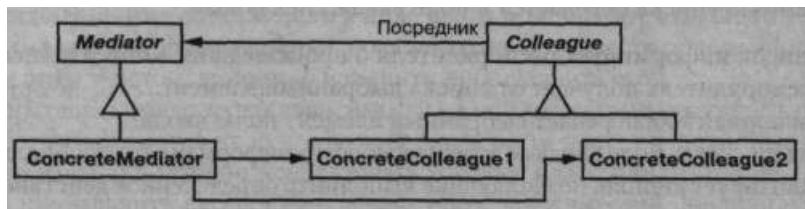
DialogDirector - это абстрактный класс, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию ShowDialog для отображения окна на экране. CreateWidgets - это абстрактная операция для создания виджетов в диалоговом окне. WidgetChanged - еще одна абстрактная операция; с ее помощью виджеты сообщают распорядителю об изменениях. Подклассы DialogDirector замещают операции CreateWidgets (для создания нужных виджетов) и WidgetChanged (для обработки извещений об изменениях).

Применимость

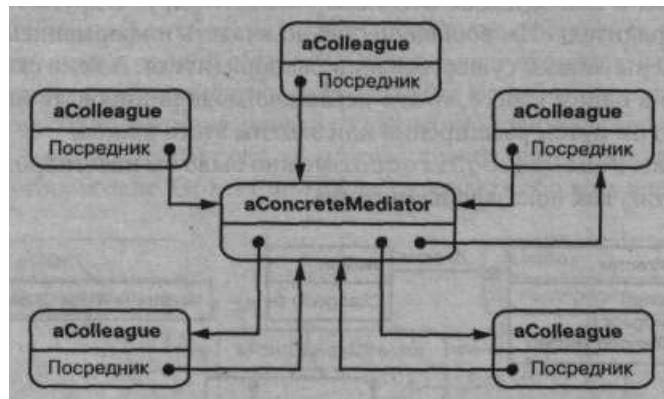
Используйте паттерн посредник, когда

- о имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- а нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
- а поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Структура



Типичная структура объектов.



Участники

Mediator (DialogDirector) - посредник;

- определяет интерфейс для обмена информацией с объектами **Colleague**;

- а **ConcreteMediator** (*FontDialogDirector*) - конкретный посредник:
 - реализует кооперативное поведение, координируя действия объектов *Colleague*;
 - владеет информацией о коллегах и подсчитывает их;
- а **Классы Colleague** (*ListBox*, *EntryField*) - коллеги:
 - каждый класс *Colleague* «знает» о своем объекте *Mediator*;
 - все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Отношения

Коллеги посыпают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

Результаты

У паттерна посредник есть следующие достоинства и недостатки:

- а *снижает число порождаемых подклассов*. Посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими 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;
- а *устраняет связанность между коллегами*. Посредник обеспечивает слабую связанность коллег. Изменять классы *Colleague* и *Mediator* можно независимо друг от друга;
- а *упрощает протоколы взаимодействия объектов*. Посредник заменяет дисциплину взаимодействия «все со всеми» дисциплиной «один со всеми», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения;
- а *абстрагирует способ 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 дает возможность прояснить имеющиеся в системе взаимодействия;
- а *централизует управление*. Паттерн посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник становится монолитом, который трудно сопровождать.

Реализация

Имейте в виду, что при реализации паттерна посредник может происходить:

- а *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;

а 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. Тогда классы коллег действуют как субъекты, посылающие извещения посреднику о любом изменении своего состояния. Посредник реагирует на них, сообщая об этом другим коллегам.

Другой подход: в классе Mediator определяется специализированный интерфейс уведомления, который позволяет коллегам обмениваться информацией более свободно. В Smalltalk/V для Windows применяется некоторая форма делегирования: общаясь с посредником, коллега передает себя в качестве аргумента, 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 - это 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 произошедших 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 качестве аргумента WidgetChanged, чтобы распорядитель имел информацию об изменившемся виджете. Подклассы DialogDirector переопределяют исключительно виртуальную функцию CreateWidgets для размещения в диалоговом окне нужных виджетов.

ListBox, Entry-Field и Button - это подклассы Widget для специализированных элементов интерфейса. В классе ListBox есть операция GetSelection для получения текущего множества выделенных элементов, а в классе Entry-Field - операция SetText для помещения 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) {
        // уничтожить диалоговое окно
    }
}
```

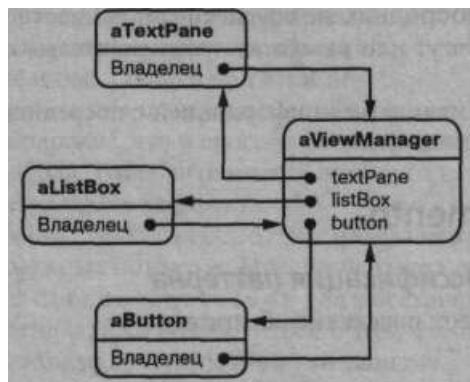
Сложность операции WidgetChanged возрастает пропорционально сложности окна диалога. Создание очень больших диалоговых окон нежелательно и по другим причинам, но в других приложениях сложность посредника может свести на нет его преимущества.

Известные применения

И в EГ++ [WGM88], и в библиотеке классов THINK C [Sym93b] применяются похожие на нашего распорядителя объекты для осуществления 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 Рапе, 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 диаграмме объектов показан мгновенный снимок работающего приложения.



В Smalltalk/V для обмена информацией между объектами Рапе и ViewManager используется механизм событий. Панель генерирует событие для получения данных от своего посредника или для информирования его о чем-то важном. С каждым 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. 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 разрешает ограничения связанности и обновляет позиции коннекторов так, чтобы отразить изменения.

Родственные паттерны

Фасад отличается от посредника тем, что абстрагирует некоторую подсистему объектов для предоставления более удобного интерфейса. Его протокол односторонний, 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 посредством паттерна наблюдатель.

Паттерн Memento

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.

Известен также под именем

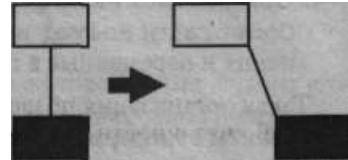
Token (лексема).

Мотивация

Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отката, 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

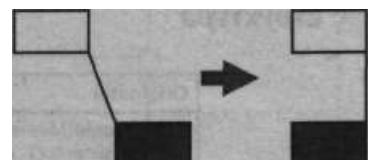
свое состояние или хотя бы его часть, делая его недоступным для других объектов, так что сохранить состояние извне невозможно. Раскрытие же состояния явилось бы нарушением принципа инкапсуляции и поставило бы под угрозу надежность и расширяемость приложения.

Рассмотрим, например, графический редактор, который поддерживает связанность объектов. Пользователь может соединить два прямоугольника линией, и они останутся в таком положении 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. Результаты вычислений объект **ConstraintSolver** использует для перерисовки графики так, чтобы были сохранены все соединения.

Поддержка отката операций в приложениях не так проста, как может показаться на первый взгляд. Очевидный способ откатить операцию перемещения - это сохранить расстояние между старым и новым положением, а затем переместить объект на такое же расстояние назад. Однако при этом не гарантируется, что все объекты окажутся там же, где находились. Предположим, что в способе расположения соединительной линии есть некоторая свобода. Тогда, переместив прямоугольник на прежнее место, мы можем не добиться желаемого эффекта.



Открытого интерфейса **ConstraintSolver** иногда не хватает для точного отката всех изменений смежных объектов. Механизм отката должен работать в тесном взаимодействии с **ConstraintSolver** для восстановления предыдущего состояния, но необходимо также позаботиться о том, чтобы внутренние детали **ConstraintSolver** не были доступны этому механизму.

Паттерн хранитель поможет решить данную проблему. Хранитель — это объект, в котором сохраняется внутреннее состояния другого объекта — хозяина хранителя. Для работы механизма отката 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 отката характеризуется такой 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 уравнения и переменные в первоначальное состояние.

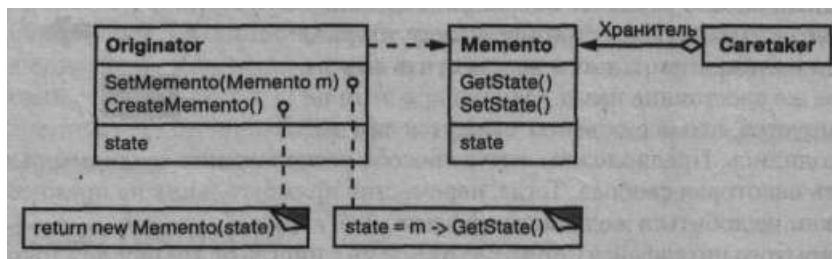
Такая организация позволяет объекту ConstraintSolver «знакомить» другие объекты с информацией, которая ему необходима для возврата в предыдущее состояние, не раскрывая в то же время свою структуру и представление.

Применимость

Используйте паттерн хранитель, когда:

- а необходимо сохранить мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии;
- а прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Структура



Участники

a 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 двойной интерфейс. «Посыльный» Caretaker «видит» лишь «з/зкмм» интерфейс хранителя - он может только передавать хранителя 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;

a Originator (ConstraintSolver) – хозяин:

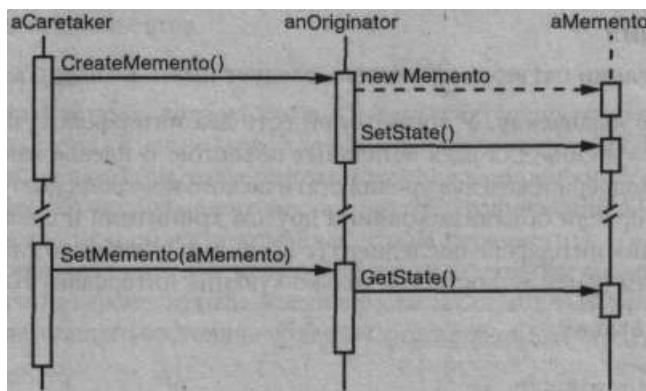
- создает хранитель, содержащего снимок текущего внутреннего состояния;
- использует хранитель для восстановления внутреннего состояния;

а **Caretaker** (механизм отката) - посыльный:

- отвечает за сохранение хранителя;
- не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

Отношения

а посыльный запрашивает хранитель у хозяина, некоторое время держит его у себя, а затем возвращает хозяину, как видно на представленной диаграмме взаимодействий.



Иногда этого не происходит, так как последнему не нужно восстанавливать прежнее состояние;

а хранители пассивны. Только хозяин, создавший хранитель, имеет доступ к информации о состоянии.

Результаты

Характерные особенности паттерна хранитель:

- а *сохранение границ инкапсуляции*. Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции;
- а *упрощение структуры хозяина*. При других вариантах дизайна, направленного на сохранение границ 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;
- а *значительные издержки при использовании хранителей*. С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой

объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит (см. также обсуждение инкрементности в разделе «Реализация»); а *определенение «узкого» и «широкого» интерфейсов*. В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию хранителя; а *скрытая плата за содержание хранителя*. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Реализация

При реализации паттерна хранитель следует иметь в виду:

а *языковую поддержку*. 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. Например:

```
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 доступны только хозяину Originator
    friend class Originator;
    Memento();

    void SetState(State* );
    State* GetState() ;
    // ...
private:
    State* state;
    // ...
};
```

а *сохранение инкрементных изменений*. Если хранители создаются и возвращаются своему хозяину в предсказуемой последовательности, то хранитель может сохранить лишь *изменения* во внутреннем состоянии хозяина.

Например, допускающие отмену команды в списке истории могут пользоваться хранителями для восстановления первоначального состояния (см. описание паттерна команда). 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 содержать 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) 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 перемещения 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-член, называемая *Solve*, отменяет ограничения, регистрируемые операцией *AddConstraint*. Для поддержки отмены действий состояние объекта ConstraintSolver можно разместить в экземпляре класса ConstraintSolverMemento с помощью операции *CreateMemento*. В предыдущее состояние объект ConstraintSolver возвращается посредством операции *SetMemento*. ConstraintSolver является примером паттерна одиночка:

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instanced;

    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 связаннысти
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // закрытое состояние Solver
};

```

С такими интерфейсами мы можем реализовать функции-члены 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 хозяина
    solver->Solve();
}

```

Execute запрашивает хранитель 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 связаннысти 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, по сути дела, 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.

Если есть класс `ItemType`, то обойти коллекцию, составленную из его экземпляров, можно так:¹

```
class ItemType {
public:
    void Process () ;
    // ...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();

while ( !aCollection.IsDone(state) ) {
    aCollection.CurrentItem(state) ->Process()
    aCollection.Next(state);
}
delete state;
```

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 стр. 258.

У интерфейса итерации, основанного на паттерне хранитель, есть два преимущества:

- а с одной коллекцией может быть связано несколько активных состояний (то же самое верно и для паттерна итератор);
- а не требуется нарушать инкапсуляцию коллекции для поддержки итерации. 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). В случае с хранителем ситуация противоположная: класс коллекции `Collection` является другом класса `IteratorState`.

В библиотеке QOSA 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. QOSA использует механизм ведения истории для возврата к прежним решениям.

Родственные паттерны

Команда: команды помещают информацию о состоянии, необходимую для отмены выполненных действий, в хранителе.

Итератор: хранители можно использовать для итераций, как было показано выше.

Паттерн Observer

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.

Известен также под именем

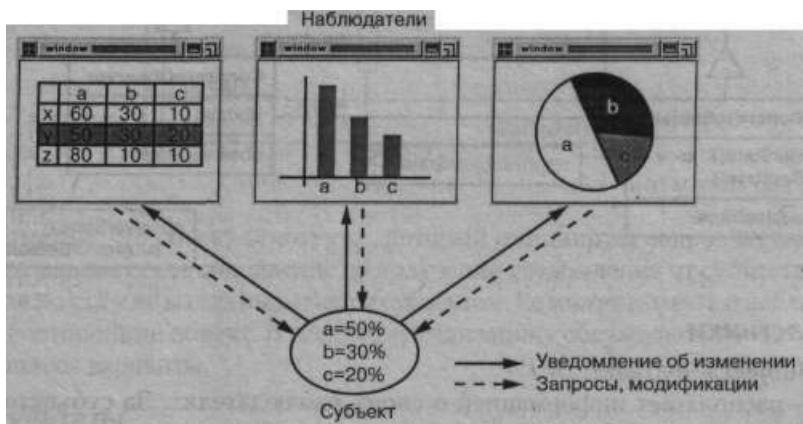
Dependents (подчиненные), Publish-Subscribe (издатель-подписчик).

Мотивация

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных

объектов. Но не хотелось бы, чтобы за согласованность надо было платить жесткой связанныстью классов, так как это в некоторой степени уменьшает возможности 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. Электронная таблица и диаграмма не имеют информации друг о друге, поэтому вы вправе использовать их по отдельности. Но *ведут* они себя так, как будто «знают» друг о друге. Когда пользователь работает с таблицей, 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 любых 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 устанавливать 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 рассыпает их, даже не имея информации о том, какие объекты являются подписчиками. На получение уведомлений может подписаться неограниченное количество наблюдателей.

Применимость

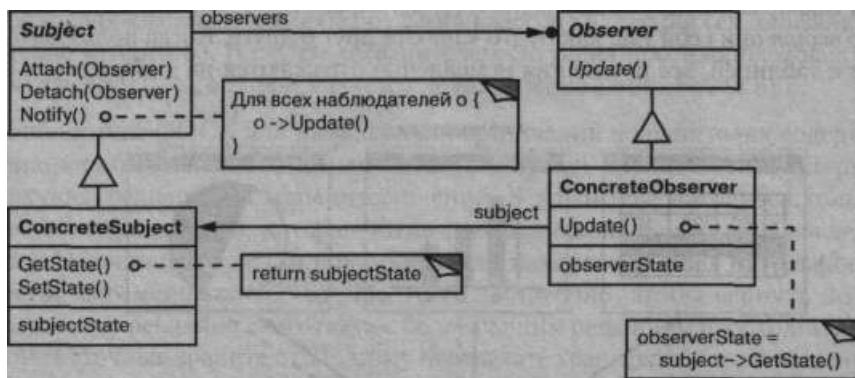
Используйте паттерн наблюдатель в следующих ситуациях:

а когда у абстракции есть два аспекта, один из которых зависит от другого.

Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;

- а когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить;
- а когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Структура



Участники

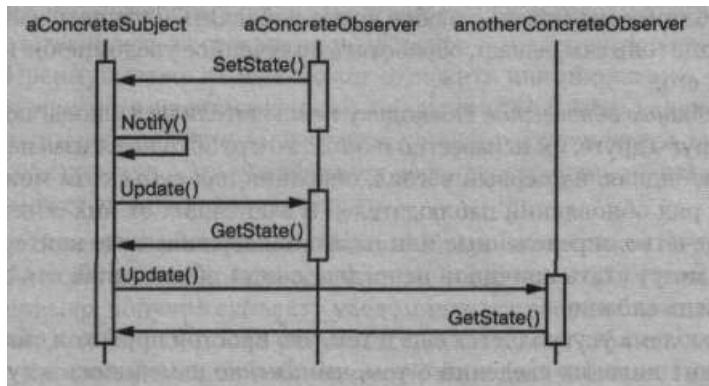
- а **Subject** - субъект:
 - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
 - предоставляет интерфейс для присоединения и отделения наблюдателей;
- а **Observer** - наблюдатель:
 - определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;
- а **ConcreteSubject** - конкретный субъект:
 - сохраняет состояние, представляющее интерес для конкретного наблюдателя **ConcreteObserver**;
 - посылает информацию своим наблюдателям, когда происходит изменение;
- а **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

- а объект **ConcreteSubject** уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;
- а 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 диаграмме взаимодействий показаны отношения между субъектом и двумя наблюдателями.



Отметим, что объект Observer, который инициирует запрос на изменение, откладывает свое обновление до получения уведомления от субъекта. Операция *Notify* не всегда вызывается субъектом. Ее может вызвать и наблюдатель, и посторонний объект. В разделе «Реализация» обсуждаются часто встречающиеся варианты.

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.

Рассмотрим некоторые достоинства и недостатки паттерна наблюдатель:

а 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, то 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 (компрометируя абстракцию уровня);

- а *поддержка широковещательных коммуникаций.* В отличие от обычного запроса для уведомления, посыпаного субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его;
- а *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.
Эта проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что *именно* изменилось в субъекте. Без дополнительного протокола, помогающего выяснить характер изменений, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Реализация

В этом разделе обсуждаются вопросы, относящиеся к реализации механизма зависимостей:

- а *отображение субъектов на наблюдателей.* С помощью этого простейшего способа субъект может отследить всех наблюдателей, которым он должен посыпать уведомления, то есть хранить на них явные ссылки. Однако при наличии большого числа субъектов и всего нескольких наблюдателей это может оказаться накладно. Один из возможных компромиссов в пользу экономии памяти за счет времени состоит в том, чтобы использовать ассоциативный массив (например, хэш-таблицу) 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;
- а *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 качестве параметра операции Update, тем самым сообщая наблюдателю, что именно нужно обследовать;
- а *кто инициирует обновление.* Чтобы сохранить согласованность, субъект и его наблюдатели полагаются на механизм уведомлений. Но какой именно объект вызывает операцию-Notify для инициирования обновления? Есть 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 следующем: при выполнении каждой из нескольких последовательных операций будут производиться обновления, что может стать причиной неэффективной работы 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;

а *висячие ссылки на удаленные субъекты*. Удаление субъекта не должно приводить к появлению висячих ссылок у наблюдателей. Избежать этого можно, например, поручив субъекту уведомлять все свои наблюдатели 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;

а *гарантии непротиворечивости 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 очень легко 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!)  
}
```

Избежать этой ловушки можно, отправляя уведомления из шаблонных методов (см. описание паттерна шаблонный метод) 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); // переопределена в подклассах  
    Notify();  
}
```

Кстати, всегда желательно фиксировать, какие операции класса *Subject* инициируют обновления;

а как избежать зависимостей 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 посыпает наблюдателям детальную информацию об изменении независимо от того, нужно ли им это. На другом - *модель вытягивания* (*pull model*), когда субъект не посыпает ничего, кроме минимального уведомления, а наблюдатели запрашивают детали позднее.

В модели вытягивания подчеркивается 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* предполагают о классах *Observer*, которые не всегда могут быть верны. С другой стороны, модель вытягивания может оказаться неэффективной, ибо наблюдателям без помощи субъекта необходимо выяснить, что изменилось;

а явное спецификация представляемых интересов модификаций. Эффективность обновления можно повысить, расширив интерфейс регистрации субъекта, то есть предоставив возможность при регистрации наблюдателя указать, какие 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*, Aspects interest);
```

где *interest* определяет представляющее интерес событие. В момент посылки уведомления субъект передает своим наблюдателям изменившийся аспект в виде параметра операции *Update*. Например:

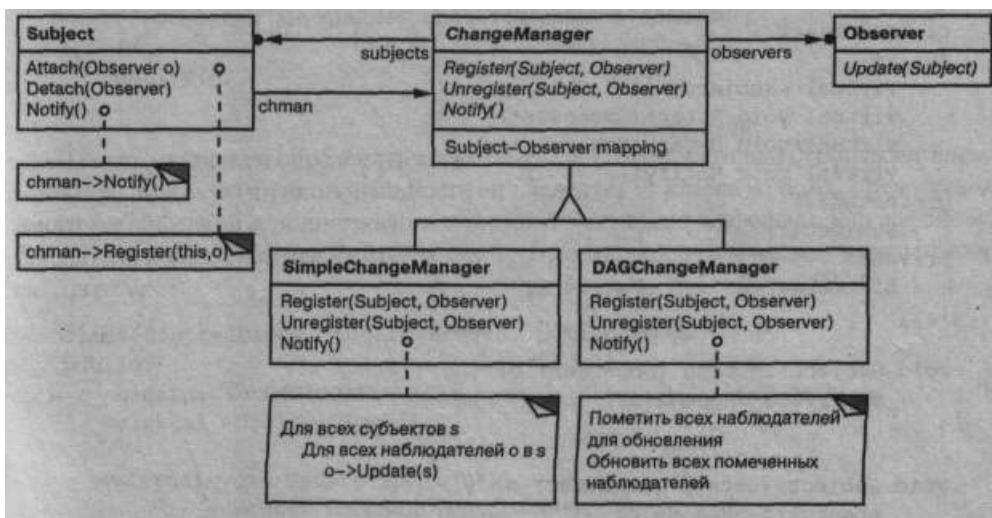
```
void Observer::Update(Subject*, Aspects interest);
```

а 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 служит для минимизации объема 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 представлять 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 диаграмме 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. В общем случае есть только один объект ChangeManager, известный всем участникам. Поэтому полезен будет также и паттерн одиночка;

а 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 () ;  
};
```

При такой реализации поддерживается несколько субъектов для одного наблюдателя. Передача субъекта параметром операции `Update` позволяет наблюдателю определить, какой из наблюдаемых им субъектов изменился.

Таким же 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 извещает 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 о времени  
    // ...  
    Notify();  
}
```

Теперь мы можем определить класс DigitalClock, который отображает время. Свою графическую функциональность он наследует от класса Widget, предоставляемого библиотекой для построения пользовательских интерфейсов. Интерфейс наблюдателя подмешивается 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 изображения 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();
    // и т.д.

    // нарисовать цифровые часы
}

```

Аналогично можно определить 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 перерисовываются себя.

Известные применения

Первый и, возможно, самый известный пример паттерна наблюдатель появился в схеме модель/вид/контроллер (МУС) языка Smalltalk, которая представляет собой каркас для построения пользовательских интерфейсов в среде

Smalltalk [KP88]. Класс Model в MVC - это 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 *видом* (view) и *объектом данных* (data object) соответственно. Unidraw делит объекты графического редактора на части View (для наблюдателей) и Subject.

Родственные паттерны

Посредник: класс ChangeManager действует как посредник между субъектами и наблюдателями, инкапсулируя сложную семантику обновления.

Одиночка: класс ChangeManager может воспользоваться паттерном одиночки, чтобы гарантировать уникальность и глобальную доступность менеджера изменений.

Паттерн State

Название и классификация паттерна

Состояние - паттерн поведения объектов.

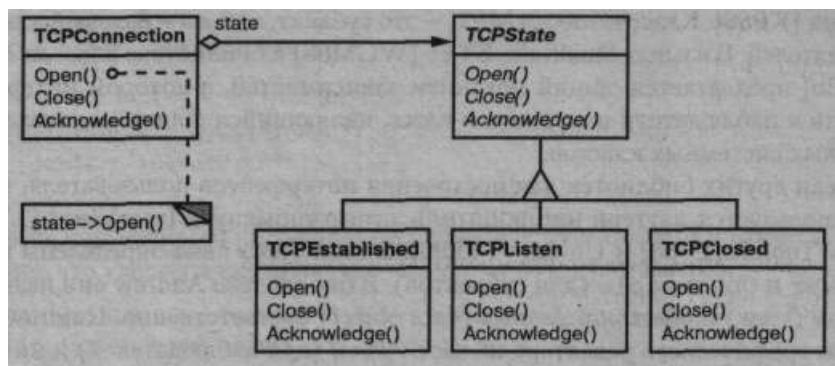
Назначение

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Мотивация

Рассмотрим класс TCPConnection, с помощью которого представлено 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 TCPConnect ion 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 поведение, 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**.

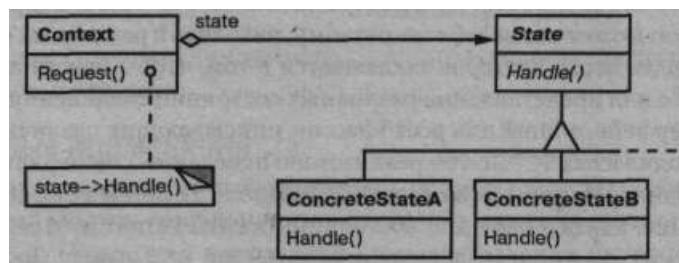
Применимость

Используйте паттерн состояния в следующих случаях:

Q когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;

O когда в коде операций встречаются состояния 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 состояния предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Структура



Участники

О Context (TCPConnection) - контекст:

- определяет интерфейс, представляющий интерес для клиентов;
- хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние;

Q State (TCPState) - состояние:

- определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context;

Q Подклассы ConcreteState (TCPEstablished, TCPListen, TCPClosed) - конкретное состояние:

- каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста Context.

Отношения

Q класс Context делегирует зависящие от состояния запросы текущему объекту ConcreteState;

Q контекст может передать себя в качестве аргумента объекту State, который будет обрабатывать запрос. Это дает возможность объекту-состоянию при необходимости получить доступ к контексту;

Q 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;

О либо Context, либо подклассы ConcreteState могут решить, при каких условиях и в каком порядке происходит смена состояний.

Результаты

Результаты использования паттерна состояния:

Q локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям. Паттерн состояния 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 ветвлений были бы разбросаны по всему коду класса Context. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение.

Паттерн состояния 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 наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. Паттерн состояния 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 состояния становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение;

Q *делает явными 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];

Q *объекты состояния можно разделять.* 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), у которых нет внутреннего состояния, а есть только поведение.

Реализация

С паттерном состояния связан целый ряд вопросов реализации:

Q *что определяет переходы между состояниями.* Паттерн состояния 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;

Q *табличная альтернатива*. Том Каргилл (Tom Cargill) в книге *C++ Programming Style* [Car92] описывает другой способ структурирования кода, управляемого сменой состояний. 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, более сложными для понимания;
- 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 состояний 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;

Q *создание и уничтожение объектов состояния*. В процессе разработки обычно приходится выбирать между:

- созданием объектов состояния, когда в них возникает необходимость, и уничтожением сразу после использования;
- созданием их заранее и навсегда.

Первый вариант предпочтителен, когда заранее неизвестно, в какие состояния будет попадать система, и контекст изменяет состояние сравнительно редко. При этом мы не создаем объектов, которые никогда не будут использованы, что существенно, если в объектах состояния хранится много информации. Когда изменения состояния происходят часто, поэтому не хотелось бы уничтожать представляющие их объекты (ибо они могут очень скоро понадобиться вновь), следует воспользоваться вторым подходом. Время на создание объектов затрачивается только один раз, в самом начале, а на уничтожение - не затрачивается вовсе. Правда, этот подход может оказаться неудобным, так как в контексте должны храниться ссылки на все состояния, в которые система теоретически может попасть;

Q *использование динамического наследования*. Варыровать поведение по запросу можно, меняя класс объекта во время выполнения, но в большинстве объектно-ориентированных языков это не поддерживается. Исключение составляет *Self* [US87] и другие основанные на делегировании языки, которые предоставляют такой механизм и, следовательно, поддерживают паттерн состояние напрямую. Объекты в языке *Self* могут делегировать операции

другим объектам, обеспечивая тем самым некую форму динамического наследования. С изменением целевого объекта делегирования во время выполнения, по существу, изменяется и структура 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

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-члене `_state` класса `TCPConnection` хранится экземпляр класса `TCPState`. Этот класс дублирует интерфейс изменения состояния, определенный в классе `TCPConnection`. Каждая операция `TCPState` принимает экземпляры `TCPConnection` как параметр, тем самым позволяя 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*);
```

¹ Пример основан на описании протокола установления TCP-соединений, приведенном в книге Линча и Роуза [LR93].

```
virtual void Synchronize (TCPConnection*) ;
virtual void Acknowledge (TCPConnection*) ;
virtual void Send (TCPConnection*) ;
protected:
    void ChangeState (TCPConnection*, TCPState*) ;
};
```

TCPConnection делегирует все зависящие от состояния запросы 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 (мы определим его 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 delegированных 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* Instanced;

    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* Instanced;

    virtual void ActiveOpen(TCPConnection*) ;
    virtual void PassiveOpen(TCPConnection*) ;
    // ...
};

```

В подклассах TCPState нет никакого локального состояния, поэтому их можно разделять, 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::Instanced);
}

```

Таким образом, каждый подкласс TCPState - это одиночка.

```
void TCPClosed::PassiveOpen (TCPConnection* t) {  
    ChangeState(t, TCPListen::Instance));  
}  
  
void TCPEstablished::Close (TCPConnection* t) {  
    // послать FIN, получить ACK для FIN  
  
    ChangeStateft, 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.  
  
    ChangeStateft, TCPEstablished::Instanced);  
}
```

После выполнения специфичных для своего состояния действий 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

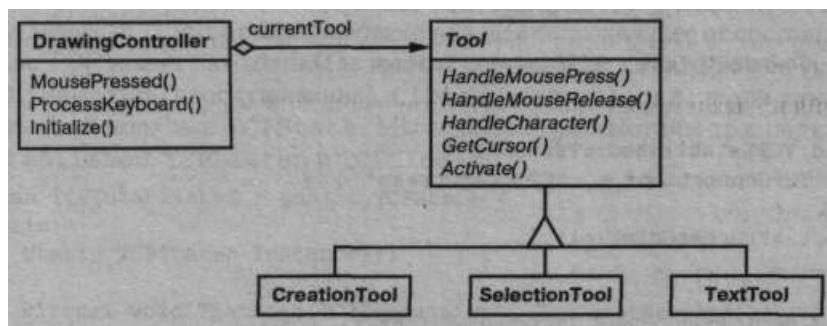
Ральф Джонсон и Джонатан Цвейг [JZ91] характеризуют паттерн состояния и описывают его применительно к протоколу TCP.

Наиболее популярные интерактивные программы рисования предоставляют «инструменты» для выполнения операций прямым манипулированием. Например, инструмент для рисования линий позволяет пользователю щелкнуть в произвольной точке мышью, а затем, перемещая мышь, провести из этой точки линию. Инструмент для выбора позволяет выбирать некоторые фигуры. Обычно все имеющиеся инструменты размещаются в палитре. Работа пользователя заключается в том, чтобы выбрать и применить инструмент, но на самом деле поведение редактора варьируется при смене инструмента: посредством 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 состояния.

Можно определить абстрактный класс Tool, подклассы которого реализуют зависящее от инструмента поведение. 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.

Данная техника используется в каркасах графических редакторов HotDraw [Joh92] и Unidraw [VL90]. Она 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 диаграмме классов схематично представлены интерфейсы классов Tool и DrawingController.



Описанный Джеймсом Коплиеном [Cop92] прием конверт-письмо (Envelope-Letter) также относится к паттерну состояния. Техника конверт-письмо - это способ изменить класс объекта во время выполнения. Паттерн состояния является частным случаем, в нем акцент делается на работу с объектами, поведение которых зависит от состояния.

Родственные паттерны

Паттерн приспособленец подсказывает, как и когда можно разделять объекты класса State.

Объекты класса state часто бывают одиничками.

Паттерн Strategy

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.

Известен также под именем

Policy (политика).

Мотивация

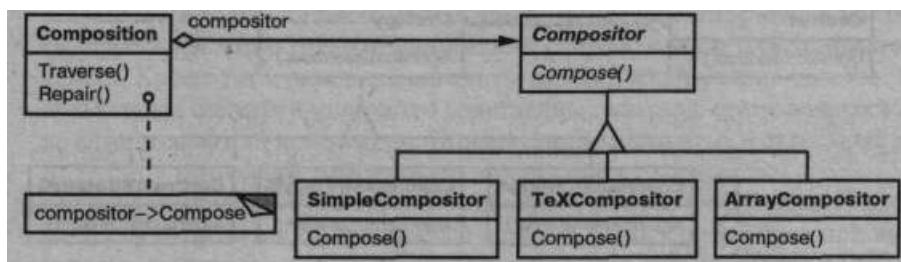
Существует много алгоритмов для разбиения текста на строки. Жестко «зашивать» все подобные алгоритмы в классы, которые в них нуждаются, нежелательно по нескольким причинам:

Q клиенту требуется алгоритм разбиения на строки, усложняется при включении в него соответствующего кода. Таким образом, клиенты становятся более громоздкими, а сопровождать их труднее, особенно если нужно поддерживать сразу несколько алгоритмов;

Q в зависимости от обстоятельств стоит применять тот или иной алгоритм. Не хотелось бы поддерживать несколько алгоритмов разбиения на строки, если мы не будем ими пользоваться;

О если разбиение на строки - неотъемлемая часть клиента, то задача добавления новых и модификации существующих алгоритмов усложняется.

Всех этих проблем можно избежать, если определить классы, инкапсулирующие различные алгоритмы разбиения на строки. Инкапсулированный таким образом алгоритм называется *стратегией*.



Предположим, что класс **Composition** отвечает за разбиение на строки текста, отображаемого в окне программы просмотра, и его своевременное обновление. Стrатегии разбиения на строки определяются не в классе **Composition**, а в подклассах абстрактного класса **Compositor**. Это могут быть, например, такие стратегии:

- Q **SimpleCompositor** реализует простую стратегию, выделяющую по одной строке за раз;
- Q **TeXCompositor** реализует алгоритм поиска точек разбиения на строки, принятый в редакторе ТХ. Эта стратегия пытается выполнить глобальную оптимизацию разбиения на строки, рассматривая сразу целый параграф;
- Q **ArrayCompositor** реализует стратегию расстановки переходов на новую строку таким образом, что в каждой строке оказывается одно и то же число элементов. Это полезно, например, 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**. Клиент указывает, какой объект **Compositor** следует использовать, параметризуя им объект **Composition**.

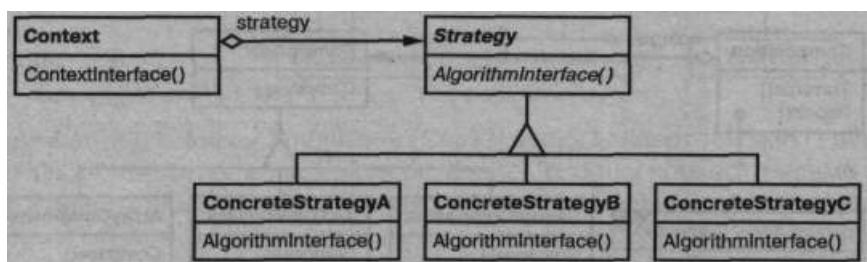
Применимость

Используйте паттерн стратегия, когда:

- Q имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;

- Q вам нужно иметь несколько разных вариантов 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];
- О в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
- Q в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Структура



Участники

Q Strategy (Compositor) - стратегия:

- объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс **Context** пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе **ConcreteStrategy**;

Q ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor) - конкретная стратегия:

- реализует алгоритм, использующий интерфейс, объявленный в классе **Strategy**;

Q Context (Composition) - контекст:

- конфигурируется объектом класса **ConcreteStrategy**;
- хранит ссылку на объект класса **Strategy**;
- может определять интерфейс, который позволяет объекту **Strategy** получить доступ к данным контекста.

Отношения

Q 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, передав ссылку на самого себя операциям класса **Strategy**;

Q контекст переадресует запросы своих клиентов объекту-стратегии. Обычно клиент создает объект **ConcreteStrategy** и передает его контексту, после

чего клиент «общается» исключительно с контекстом. Часто в распоряжении клиента находится несколько классов ConcreteStrategy, которые он может выбирать.

Результаты

У паттерна стратегия есть следующие достоинства и недостатки:

О семейства родственных алгоритмов. Иерархия классов **Strategy** определяет семейство алгоритмов или поведений, которые можно повторно использовать в разных контекстах. Наследование позволяет выделить общую для всех алгоритмов функциональность;

Q 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**. Вот почему реализации алгоритма и контекста смешиваются, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. В результате вы получите множество родственных классов, отличающихся только алгоритмом или поведением. Инкапсуляции алгоритма в отдельный класс **Strategy** позволяют изменять его независимо от контекста;

Q с помощью стратегий можно избавиться от условных операторов. Благодаря паттерну 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;
        // ...
    }
    // если необходимо, объединить результаты с имеющейся
    // композицией
}
```

Паттерн же стратегия позволяет обойтись без оператора переключения за счет делегирования задачи разбиения на строки объекту **Strategy**:

```
void Composition::Repair () {
    _compositor->Compose();
    // если необходимо, объединить результаты
    // с имеющейся композицией
}
```

Если код содержит много условных операторов, то часто это признак того, что нужно применить паттерн стратегия;

О 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;

Q 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 имеют значение для клиента;

Q обмен информацией между стратегией и контекстом. Интерфейс класса **Strategy** разделяется всеми подклассами **ConcreteStrategy** — неважно, сложна или тривиальна 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;

О 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 разделяться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Разделяемые стратегии не должны сохранять состояние между вызовами. В описании паттерна приспособленец этот подход обсуждается более подробно.

Реализация

Рассмотрим следующие вопросы реализации:

Q определение интерфейсов классов Strategy и Context. Интерфейсы классов **Strategy** и **Context** могут обеспечить объекту класса **ConcreteStrategy** эффективный доступ к любым данным контекста, и наоборот.

Например, **Context** передает данные в виде параметров операциям класса **Strategy**. Это разрывает тесную связь между контекстом и стратегией. При этом не исключено, что контекст будет передавать данные, которые стратегии не нужны.

Другой метод — передать контекст в качестве аргумента, в таком случае стратегия будет запрашивать у него данные, или, например, сохранить ссылку на свой контекст, так что передавать вообще ничего не придется. И в том, и в другом случаях стратегия может запрашивать только 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**.

Какой подход лучше, зависит от конкретного алгоритма и требований, которые он предъявляет к данным;

а стратегии как параметры шаблона. В C++ для конфигурирования класса стратегией можно использовать шаблоны. Этот способ хорош, только если стратегия определяется на этапе компиляции и ее не нужно менять во время выполнения. Тогда конфигурируемый 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 конфигурируется классом Strategy в момент инстанцирования:

```
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 виде параметра шаблона позволяет статически связать стратегию с контекстом, вследствие чего повышается эффективность программы;

и объекты-стратегии можно не задавать. Класс Context разрешается упростить, если для него отсутствие какой бы то ни было стратегии является нормой. 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. Достоинство такого подхода в том, что клиентам вообще не нужно иметь дело со стратегиями, если их устраивает поведение по умолчанию.

Пример кода

Из раздела «Мотивация» мы приведем фрагмент высокоуровневого кода, 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 можно увеличивать объект по сравнению с его естественным размером, а сжимаемость — насколько можно этот размер уменьшать. Композиция передает эти значения компоновщику, который использует их, чтобы найти оптимальное место для разрыва строки.

```

class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components; // список компонентов
    int _componentCount; // число компонентов
    int _lineWidth; // ширина строки в композиции Composition
    int* _lineBreaks; // позиции точек разрыва строки
                      // (измеренные в компонентах)
    int _lineCount; // число строк
};

```

Когда возникает необходимость изменить расположение элементов, композиция запрашивает у компоновщика позиции точек разрыва строк. При этом она передает компоновщику три массива, в которых описаны естественные размеры, растягиваемость и сжимаемость компонентов. Кроме того, передается число компонентов, ширина строки и массив, в который компоновщик должен поместить позиции точек разрыва. Компоновщик возвращает число рассчитанных им точек разрыва.

Интерфейс класса Compositor позволяет композиции передать компоновщику всю необходимую ему информацию. Приведем 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 определены различные стратегии разбиения на строки.

Композиция обращается к своему компоновщику посредством 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 разрыва и, наконец, отображает документ (этот код также опущен):

```

void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;
}

```

```
// подготовить массивы с характеристиками компонентов
// ...

// определить, где должны быть точки разрыва
int breakCount;
breakCount = _compositor->Compose(
    natural, stretchability, shrinkability,
    componentCount, _lineWidth, breaks
);

// разместить компоненты с учетом точек разрыва
// ...
}
```

Теперь рассмотрим подклассы класса Compositor. Класс SimpleCompositor для определения позиций точек разрыва исследует 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 равномерно «раскрасить» абзац, минимизируя 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 операции 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 вы передаете ему 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]⁶ 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 (кризис 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 Грейди Буча [BV90] стратегии используются как аргументы шаблонов. В классах коллекций поддерживаются три разновидности

стратегий распределения памяти: управляемая (распределение из пула), контролируемая (распределение и освобождение защищены замками) и неуправляемая (стандартное распределение памяти). Стратегия передается классу коллекции в виде аргумента шаблона в момент его инстанцирования. Например, коллекция `UnboundedCollection`, в которой используется неуправляемая стратегия, инстанцируется как `UnboundedCollection<MyItemType*, Unmanaged>`.

RApp - это система для проектирования топологии интегральных схем [GA89, AG90]. Задача RApp - проложить провода 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 при проверке корректности введенной строки потребуется поиск данных в 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: объекты-стратегии в большинстве случаев подходят как приспособленцы.

Паттерн Template Method

Название и классификация паттерна

Шаблонный метод — паттерн поведения классов.

Назначение

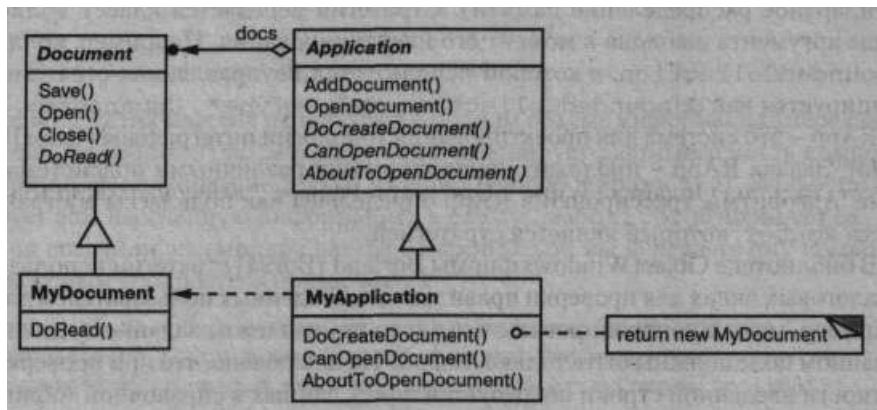
Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

Мотивация

Рассмотрим каркас приложения, в котором имеются классы `Application` и `Document`. Класс `Application` отвечает за открытие существующих документов, хранящихся во внешнем формате, например в виде файла. 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 считывания документа в операции OpenDocument:

```

void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // работа с этим документом невозможна
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}

```

Операция OpenDocument определяет все шаги открытия документа. Она проверяет, можно ли открыть документ, создает объект класса Document, добавляет его к набору документов и считывает документ из файла.

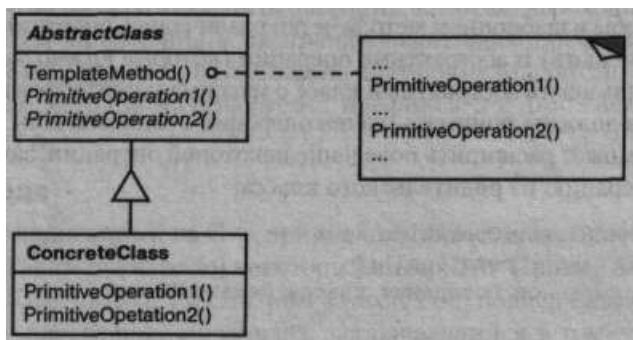
Операцию вида OpenDocument мы будем называть *шаблонным методом*, описывающим алгоритм в терминах абстрактных операций, которые замещены в подклассах для получения нужного поведения. Подклассы класса Application выполняют проверку возможности открытия (CanOpenDocument) и создания документа (DoCreateDocument). 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

Паттерн шаблонный метод следует использовать:

- а чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
- а когда нужно выделить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения», описанной в работе Уильяма Опдейка (William Opdyke) и Ральфа Джонсона (Ralph Johnson) [OJ93]. Сначала идентифицируются различия в существующем коде, а затем они выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;
- а для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зажечки (hooks) – см. раздел «Результаты» – в определенных точках, разрешив тем самым расширение только в этих точках.

Структура



Участники

- а **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;
- а **ConcreteClass** (MyApplication) - конкретный класс:
 - реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

Отношения

ConcreteClass предполагает, что инвариантные шаги алгоритма будут выполнены в **AbstractClass**.

Результаты

Шаблонные методы - один из фундаментальных приемов повторного использования кода. Они особенно важны 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 позвоним» [Swe85]. В данном случае это означает, что родительский класс вызывает операции подкласса, а не наоборот.

Шаблонные методы вызывают операции следующих видов:

- а конкретные операции (либо из класса `ConcreteClass`, либо из классов клиента);
- а конкретные операции из класса `AbstractClass` (то есть операции, полезные всем подклассам);
- а примитивные операции (то есть абстрактные операции);
- а фабричные методы (см. паттерн фабричный метод);
- а 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, автотипы 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 () {
    ParentClass::Operation();
    // Расширенное поведение класса Periveddass
}
```

К сожалению, очень легко забыть о необходимости вызывать унаследованную операцию. У нас есть возможность трансформировать такую операцию в шаблонный метод с целью 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 родительского класса ParentClass
    HookOperation();
}
```

В родительском классе `ParentClass` операция `HookOperation` не делает ничего:

```
void ParentClass::HookOperation () { }
```

Но она замещена в подклассах, которые расширяют поведение:

```
void DerivedClass::HookOperation () {
    // расширение в производном классе
}
```

Реализация

Стоит рассказать о трех аспектах, касающихся реализации:

а *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;

а *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;

а *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 написанный на C++ пример показывает, как родительский класс может навязывать своим подклассам некоторый инвариант. Пример взят из библиотеки NeXT AppKit [Add94]. Рассмотрим класс View, поддерживающий рисование на экране, - своего рода инвариант, заключающийся 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).

Чтобы установить состояние, можно использовать шаблонный метод Display. В классе View определены две конкретные операции (SetFocus и ResetFocus), которые соответственно устанавливают и сбрасывают контекст рисования. Операция-зачепка 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 () { }
```

Чтобы она что-то рисовала, подклассы переопределяют ее:

```
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. [WBW90, WBJ90] подробно обсуждаются шаблонные методы.

Родственные паттерны

Фабричные методы часто вызываются из шаблонных. В примере из раздела «Мотивация» шаблонный метод OpenDocument вызывал фабричный метод DoCreateDocument.

Стратегия: шаблонные методы применяют наследование для модификации части алгоритма. Стратегии используют делегирование для модификации алгоритма в целом.

Паттерн Visitor

Название и классификация паттерна

Посетитель - паттерн поведения объектов.

Назначение

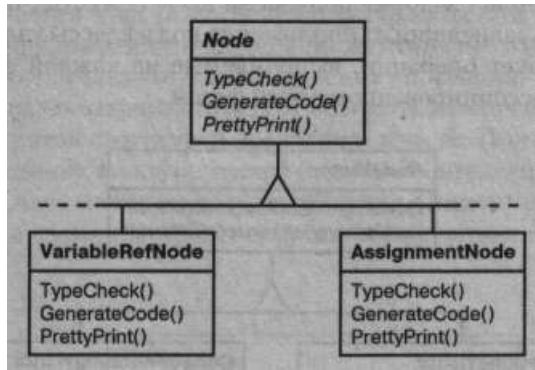
Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

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 красивой печати программы, реструктурирования кода и вычисления различных метрик программы.

В большинстве таких операций узлы дерева, представляющие операторы присваивания, следует рассматривать 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.

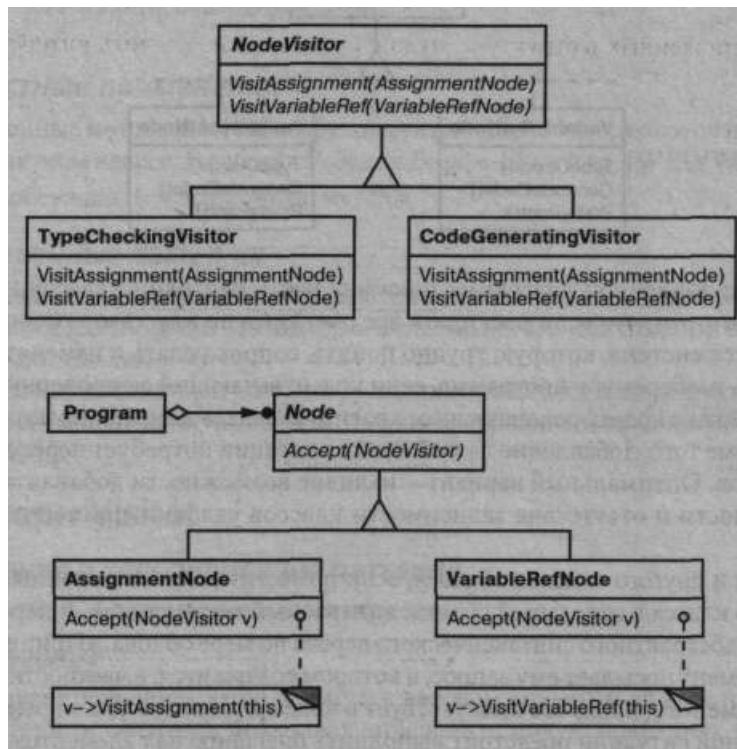


На представленной диаграмме показана часть иерархии классов `Node`. Проблема здесь в том, что если раскидать все операции по классам различных узлов, то получится система, которую трудно понять, сопровождать и изменять. Вряд ли кто-нибудь разберется в программе, если код, отвечающий за проверку типов, 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.

И того, и другого можно добиться, если поместить взаимосвязанные операции из каждого класса в отдельный объект, называемый *посетителем*, и передавать его элементам абстрактного синтаксического дерева по мере обхода. «Принимая» посетителя, элемент посыпает 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 диаграмму 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 качестве аргумента. 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 необходим 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`.

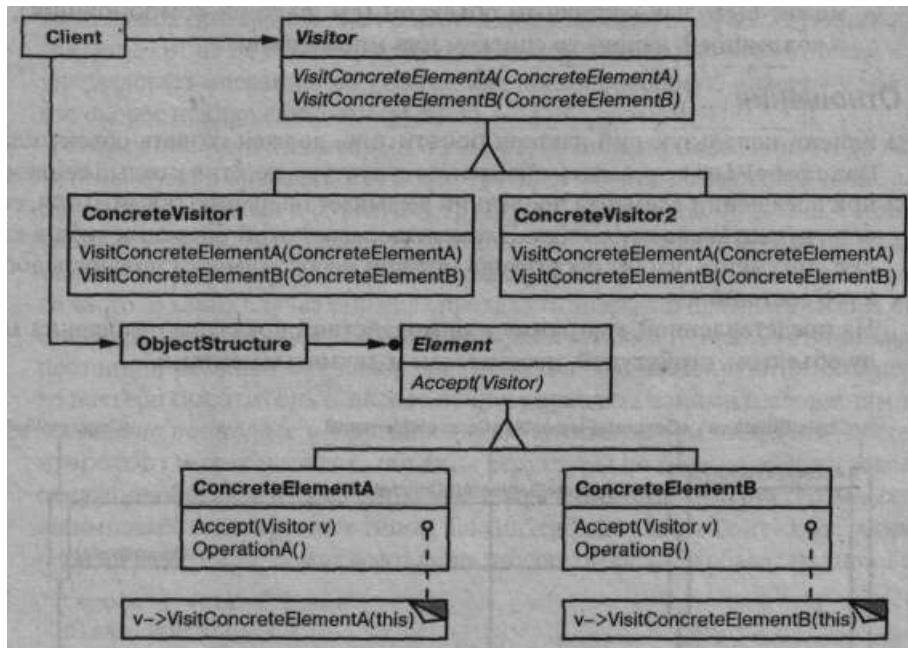
Применимость

Используйте паттерн посетитель, когда:

- а в структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;

- а над объектами, входящими в состав структуры, надо выполнять 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;
- а классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, нужно будет 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

- а **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;
- а **ConcreteVisitor** (TypeCheckingVisitor) - конкретный посетитель:
- реализует все операции, объявленные в классе **Visitor**. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего

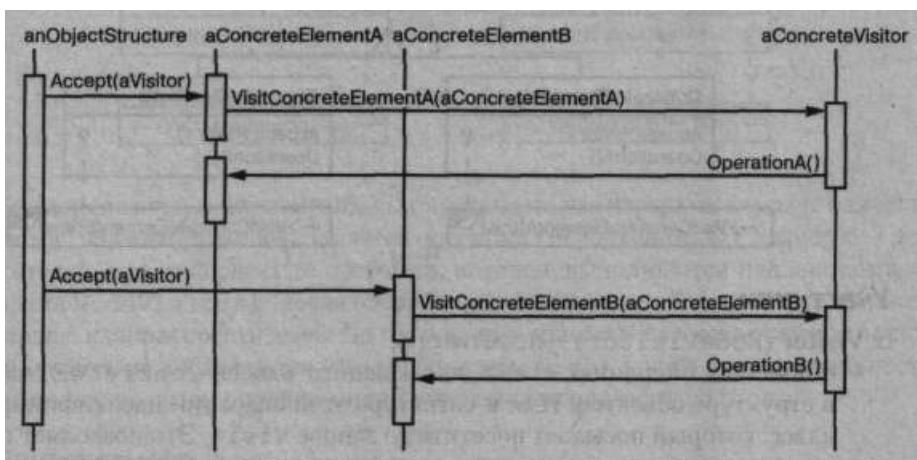
объекта в структуре. Класс ConcreteVisitor предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры;

- а **Element (Node)** - элемент:
 - определяет операцию Accept, которая принимает посетителя в качестве аргумента;
- а **ConcreteElement (AssignmentNode, VariableRefNode)** - конкретный элемент:
 - реализует операцию Accept, принимающую посетителя как аргумент;
- а **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), так и коллекцией, например списком или множеством.

Отношения

- а клиент, использующий паттерн посетитель, должен создать объект класса ConcreteVisitor, а затем обойти всю структуру, посетив каждый ее элемент.
- а при посещении элемента последний вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в качестве аргумента, чтобы посетитель мог при необходимости получить доступ к его состоянию.

На представленной диаграмме взаимодействий показаны отношения между объектом, структурой, посетителем и двумя элементами.



Результаты

Некоторые достоинства и недостатки паттерна посетитель:

Q *упрощает добавление 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;

а *объединяет родственные операции и отсекает те, 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;

а *добавление новых классов ConcreteElement затруднено.* 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 сопровождать иерархию классов **Visitor** будет нелегко, 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;

а *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 стр. 255, может всего лишь получить доступ к объектам типа **Item**:

```
template <class Item>
class Iterator {
    // ...
    Item CurrentItemO 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;

- а *аккумулирование состояния*. Посетители могут аккумулировать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, состояние придется передавать в виде дополнительных аргументов операций, выполняющих обход, или хранить в глобальных переменных;
- а *нарушение инкапсуляции*. Применение посетителей подразумевает, что у класса ConcreteElement 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 ассоциирован 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++ класс `Visitor` следовало бы объявить приблизительно так:

```
class Visitor {
public:
    virtual void VisitElementA(ElementA* );
    virtual void VisitElementB(ElementB* );

    // и так далее для остальных 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

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 за, так и против подобной перегрузки. 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.

Конкретные элементы объявляются так:

```
class Element {  
public:  
    virtual ~Element();  
    virtual void Accept(Visitors) = 0;  
protected:  
    Element();  
};  
  
class ElementA : public Element {  
public:  
    ElementA();  
    virtual void Accept(Visitors v) { v.VisitElementA(this); }  
};  
  
class ElementB : public Element {  
public:  
    ElementB();  
    virtual void Accept(Visitors v) { v.VisitElementB(this); }  
};
```

Класс CompositeElement мог бы реализовать операцию Accept следующим образом:

```
class CompositeElement : public Element {  
public:  
    virtual void Accept(Visitors);  
private:  
    List<Element*>* _children;  
};  
  
void CompositeElement::Accept (Visitors 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:

а *двойная диспетчеризация*. По своей сути паттерн посетитель позволяет, не изменяя классы, добавлять в них новые операции. Достигает он этого 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`:

а *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). 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*

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 выбиралась 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.

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 поместить алгоритм обхода в посетитель, хотя закончится это дублированием кода обхода в каждом классе `ConcreteVisitor` для 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. Для определения операций, создающих инвентарную описание 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. 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(EquipmentVisitors*) ;
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, соответствующую тому классу, который получил 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<Equipment*> i(_parts);
        !i.IsDone();
        i.Next()
    ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}

```

Подклассы EquipmentVisitor определяют конкретные алгоритмы, применимые 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-член. Особенноважното, чтодляоценки конструкции можно выбрать другую стратегию, просто поменяв класс **PricingVisitor**.

Определить посетитель для составления инвентарной описи можно следующим образом:

```
class InventoryVisitor : public EquipmentVisitor {  
public:  
    InventoryVisitor();  
  
    InventoryS 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);  
}
```

InventoryVisitor к структуре объектов можно применить следующим образом:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Инвентарная опись "
<< component->Name()
<< visitor.GetInventory();
```

Далее мы покажем, как на языке Smalltalk реализовать пример из описания паттерна интерпретатор с помощью паттерна посетитель. Как и в предыдущем случае, этот пример настолько мал, что паттерн посетитель практически бесполезен, но служит неплохой иллюстрацией основных принципов. Кроме того, демонстрируется ситуация, в которой обход выполняет 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 `expression!` и `expr2!`; для `AlternationExpression` - `alternative!` 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 много-кратно. 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 выражение должно сопоставиться, чтобы получить текущее состояние:

```
visitSequence: sequenceExp
inputstate := sequenceExp expression1 accept: self.
^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
I finalState I
finalState := inputstate copy,
[inputstate isEmpty]
```

```

whileFalse:
    [inputState := repeatExp repetition accept: self.
     finalState addAll: inputState].
    ^ finalState

VisitAlternation: alternateExp
    I finalState originalState I
    originalState := inputState.
    finalState := alternateExp alternative1 accept: self.
    inputState := originalState.
    finalState addAll: (alternateExp alternative2 accept: self).
    ^ finalState

visitLiteral: literalExp
    I finalState tStream I
    finalState := Set new.
    inputState
    do:
        [:stream I 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 генерации 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: посетители могут использоваться для выполнения операции над всеми объектами структуры, определенной с помощью паттерна компоновщик.

Интерпретатор: 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. Своим названием паттерн как раз и обязан этому объекту.¹ Например:

- а объект стратегия инкапсулирует алгоритм;
- а объект состояние инкапсулирует поведение, зависящее от состояния;
- а объект посредник инкапсулирует протокол общения между объектами;
- а объект итератор инкапсулирует способ 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.

Но не все паттерны поведения разбивают функциональность таким образом. Например, паттерн цепочка обязанностей 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.

Объекты как аргументы

В нескольких паттернах участвует объект, *всегда* используемый только как аргумент. Одним из них является посетитель. Объект-посетитель - это аргумент

¹ Эта тема красной нитью проходит и через другие паттерны. Абстрактная фабрика, построитель 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.

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 палочек, которые передаются от одного владельца к другому и активизируются в будущем. К этой категории относятся команда и хранитель. В паттерне команда такой «палочкой» является запрос, а в хранителе она представляет внутреннее состояние объекта в определенный момент. И-там, и там «йалочка» 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 и наблюдатель 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.

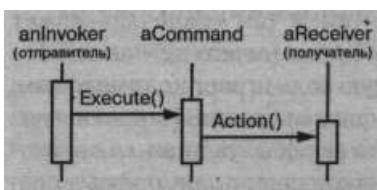
Нам кажется, что повторно использовать наблюдатели и субъекты проще, чем посредники. Паттерн наблюдатель способствует разделению и ослаблению связей между наблюдателем и субъектом, что приводит к появлению сравнительно мелких классов, более приспособленных для повторного использования.

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 из-за присущей наблюдателю косвенности разобраться в системе все же нелегко.

В языке Smalltalk наблюдатели можно параметризовать сообщениями, приемлемыми 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 наблюдатель там, где программист на C++ применил бы 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.

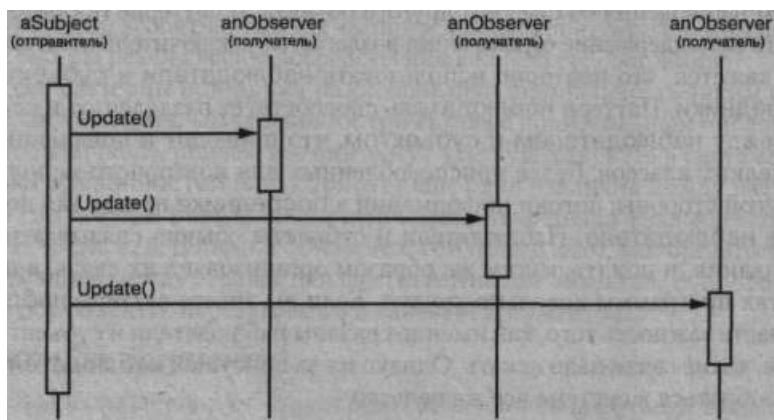


Паттерн команда поддерживает разделение за счет объекта-команды, который определяет привязку отправителя к получателю.

Паттерн команда предоставляет простой интерфейс для выдачи запроса (операцию Execute). Заключение связи между отправителем и получателем в самостоятельный объект позволяет отправителю работать с разными получателями.

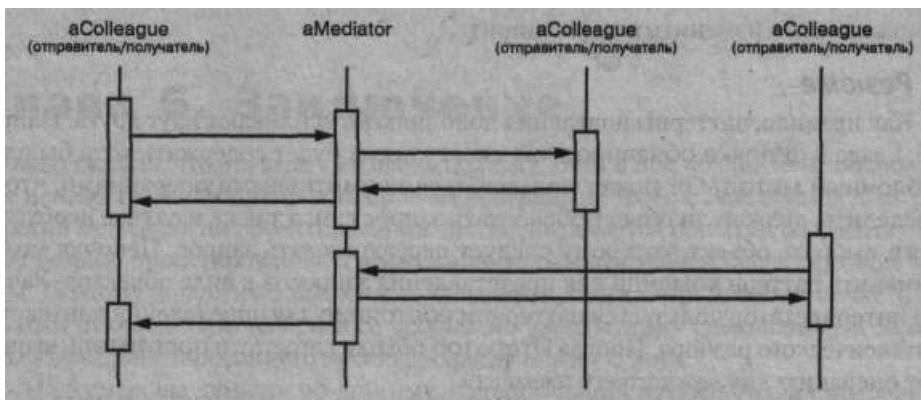
Он отделяет отправителя от получателей, облегчая тем самым повторное использование. Кроме того, объект-команду можно повторно использовать для параметризации получателя различными отправителями. Номинально паттерн команда требует определения подкласса для каждой связи отправитель-получатель, хотя имеются способы реализации, при которых удается избежать порождения подклассов.

Паттерн наблюдатель отделяет отправителей (субъектов) от получателей (наблюдателей) путем определения интерфейса для извещения о произошедших 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.

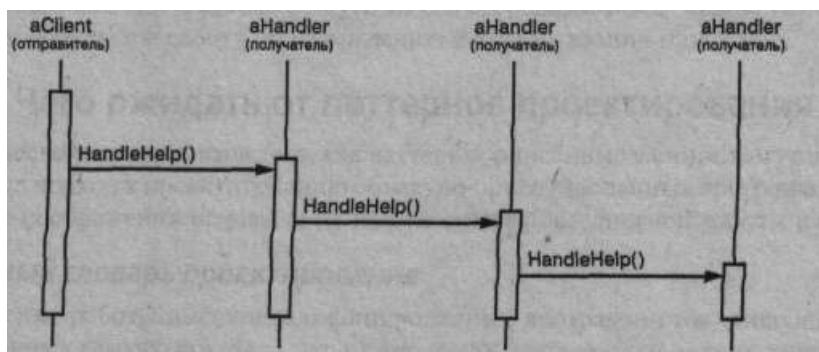
Паттерн посредник разделяет объекты, заставляя их ссылаться друг на друга косвенно, через объект-посредник.



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.

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 - это 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

Как правило, паттерны поведения дополняют и усиливают друг друга. Например, класс в цепочке обязанностей, скорее всего, будет содержать хотя бы один шаблонный метод. 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 команды для представления запросов в виде объектов. Зачастую интерпретатор пользуется паттерном состояния для определения контекстов синтаксического разбора. Иногда Итератор обходит агрегат, а посетитель выполняет операцию для каждого его элемента.

Паттерны поведения хорошо сочетаются и с другими паттернами. Например, система, в которой применяется паттерн компоновщик, время от времени использует посетитель 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 для переопределения некоторых свойств частей композиции. А паттерн наблюдатель может связать структуры разных объектов, тогда как паттерн состояние позволит компонентам варьировать свое поведение при изменении состояния. Сама композиция может быть создана с применением строителя и рассматриваться как прототип какой-то другой частью системы.

Хорошо продуманные объектно-ориентированные системы внешне 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.



Приложение А. Глоссарий

Абстрактная операция - операция, которая объявляет сигнатуру, но не реализует ее. В C++ абстрактные операции соответствуют *исключительно виртуальным, функциям-членам*.

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.

Диаграмма взаимодействий - диаграмма, на которой показан поток запросов между объектами.

Диаграмма классов - диаграмма, на которой изображены классы, их внутренняя структура и операции, а также статические связи между ними.

Диаграмма объекта - диаграмма, на которой изображена структура конкретного объекта во время выполнения.

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-членами*, в Smalltalk — *методами*.

Операция класса - операция, определенная для класса в целом, а не для индивидуального объекта. В C++ операции класса называются *статическими функциями-членами*.

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 *данные-член*.

Подкласс – класс, наследующий другому классу. В C++ подкласс называется *производным классом*.

Подмешанный класс – класс, спроектированный так, чтобы сочетаться с другими классами путем наследования. Подмешанные классы обычно абстрактны.

Подсистема – независимая группа классов, функционирующих совместно для выполнения набора обязанностей.

Подтип – один тип называется подтипов 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.

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.

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 В. Объяснение нотации

На протяжении всей книги мы пользуемся диаграммами для иллюстрации важных идей. Некоторые диаграммы 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 диаграмм:

- а на *диаграмме классов* отображены классы, их структура и статические отношения между ними;
- а на *диаграмме объектов* показана структура объектов во время выполнения;
- а на *диаграмме взаимодействий* изображен поток запросов между объектами.

В описании каждого паттерна проектирования есть хотя бы одна диаграмма классов. Остальные используются, если в них возникает необходимость. Диаграммы классов и объектов основаны на методологии OMT (Object Modeling Technique - методика моделирования объектов) [RBP+91, Rum94].¹ Диаграммы взаимодействий заимствованы из методологии Objectory [JCJO92] и метода Буча [Boo94].

B.1. Диаграмма классов

На рисунке B.1a представлена нотация ОМТ 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 находятся 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.

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,

¹ В ОМТ для обозначения диаграмм классов используется термин «диаграмма объектов». Мы же зарезервировали термин «диаграмма объекта» исключительно для описания структуры объекта.

например, обстоит дело в паттерне приспособленец. 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 рисунке B.1b. Примером может служить паттерн заместитель. Бледный шрифт клиента напоминает также о том, что мы специально не включили клиента в состав участников.

На рисунке B.1c показаны отношения между классами. В нотации ОМТ 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 ОМТ такого отношения не поддерживает. Мы называем такое отношение «создает». Стрелка направлена в сторону класса, экземпляр которого инстанцируется. На рисунке B.1c класс CreationTool создает объекты класса LineShape.

В ОМТ определен также символ залитого круга, обозначающий «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. На рисунке B.1c показано, что класс Drawing агрегирует несколько объектов типа Shape.

Наконец, мы дополнили ОМТ аннотациями на псевдокоде, которые позволяют коротко описать реализацию операций. На рисунке B.1d приведена такая аннотация для операции Draw в классе Drawing.

B.2. Диаграмма объектов

На диаграмме объектов представлены только экземпляры. На ней показан мгновенный снимок объектов в паттерне проектирования. Объекты именуются «aSomething», где Something - это класс объекта. Для обозначения объекта

¹ В ОМТ 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.

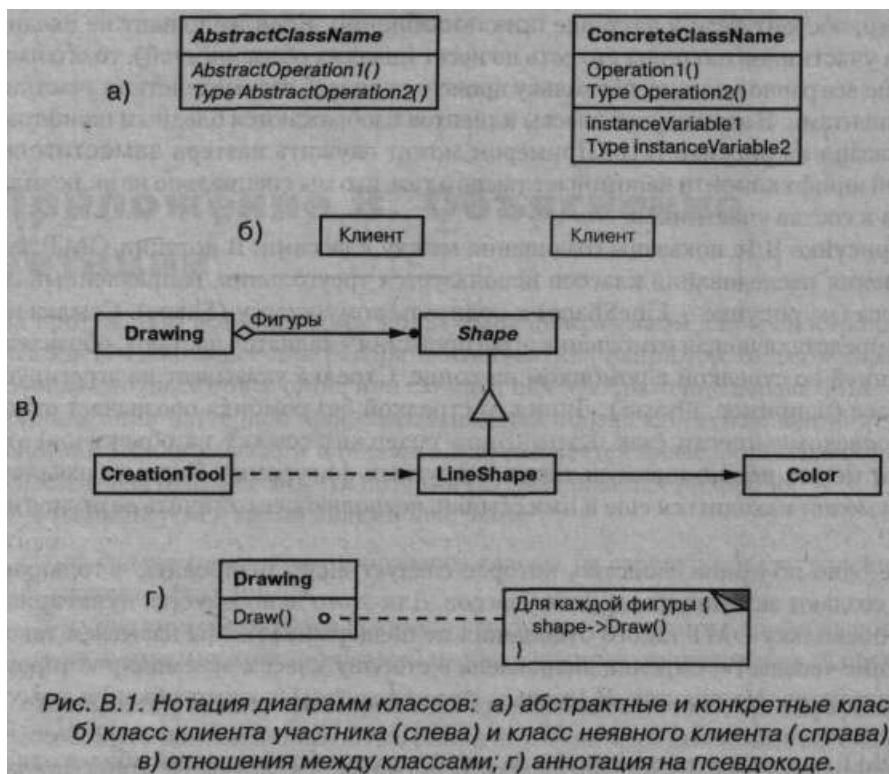


Рис. В.1. Нотация диаграмм классов: а) абстрактные и конкретные классы; б) класс клиента участника (слева) и класс неявного клиента (справа); в) отношения между классами; г) аннотация на псевдокоде.

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. На рисунке В.2 приведен соответствующий пример.

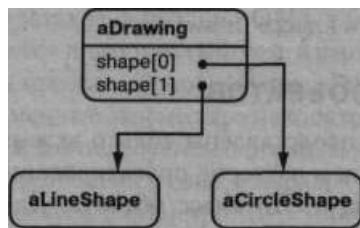


Рис. В.2. Нотация диаграмм объектов

В.3. Диаграмма взаимодействий

Порядок исполнения запросов, которые объекты посылают друг другу, показан на диаграмме взаимодействий. Так, на рисунке В.3 представлено, как фигура добавляется к рисунку.

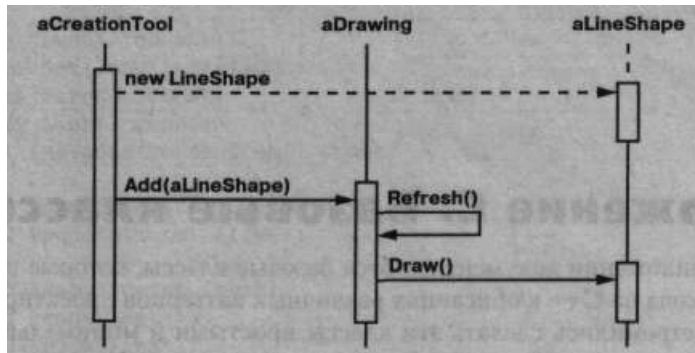


Рис. В.3. Нотация диаграмм взаимодействия

На диаграмме взаимодействий время откладывается сверху вниз. Сплошная вертикальная линия обозначает время жизни объекта. Соглашение об именовании объектов такое же, как на диаграммах объектов: имени класса предшествует буква «*a*» (например, *aShape*). Если объект еще не создан к начальному моменту времени, представленному на диаграмме, 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 рисунке В.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 посылка объектом *aDrawing* запроса к *aLineShape*.

Приложение С. Базовые классы

В данном приложении документируются базовые классы, которые применялись в примерах кода на C++ в описаниях различных паттернов проектирования. Мы специально стремились сделать эти классы простыми и минимальными. Будут описаны следующие классы:

- а **List** – упорядоченный список объектов;
- а **Iterator** – интерфейс для последовательного доступа к объектам в агрегате;
- а **ListIterator** – итератор для обхода списка;
- а **Point** – точка с двумя координатами;
- а **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;
```

C.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-членов.

- **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.

Доступ

Следующие операции обеспечивают доступ к элементам списка.

long Count () const - возвращает число объектов в списке.

Item& Get (long index) const - возвращение объекта с заданным индексом.

Item& First () const - возвращае первый объект в списке.

Item& Last () const - возвращение последнего объекта 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 сравнения на равенство **==**.

void RemoveFirst () - удаляет первый элемент из списка.

void RemoveLast () - удаление последнего элемента из списка.

void RemoveAll() - удаляет все элементы из списка.

Интерфейс стека

Item& Top () const - возвращает элемент, находящийся на вершине стека.
void Push (const Item&) - «заталкивает» элемент в стек.
Item& Pop () — «выталкивает» элемент с вершины стека.

C.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 делают следующее:

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.

C.3. ListIterator

ListIterator реализует интерфейс класса **Iterator** для обхода списка **List**. Его конструктор принимает в качестве аргумента 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;
};
```

C.4. Point

Класс Point представляет точку на плоскости с помощью декартовых координат, 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&);  
  
    Points operator+=(const Point&);  
    Points operator-=(const Point&);  
    Points operator*=(const Point&);  
    Points 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&);  
};
```

Статический член Zero представляет начало координат Point (0, 0).

C.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 Points origin, const Points 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 Points);
Points Extent() const; void Extentfconst Points);

void MoveTo(const Points);
void MoveBy(const Points);

bool IsEmpty() const;
bool Contains(const Points) const;
};
```

Статический член Zero представляет вырожденный прямоугольник:

```
Rect(Point(0, 0), Point(0, 0));
```