Применение шаблонов проектирования

Дополнительные штрихи

ДЖОН ВЛИССИДЕС
Исследовательский центр IBM им. Т.Дж. Уотсона

Москва • Санкт-Петербург • Киев 2003

ББК 32.973.26-018.2.75
В58 УДК
681.3.07

Издательский дом "Вильямс"

Зав. редакцией *А.В. Слепцов*

Перевод с английского и редакция *И.А. Ореховой*

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

Влиссидес, Джон.

В58 Применение шаблонов проектирования. Дополнительные штрихи.: Пер. с англ. — М.: Издательский дом "Вильямс", 2003. — 144 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0393-9 (рус.)

Данная книга предназначена для разработчиков программного обеспечения, использующих в своей работе шаблоны проектирования. Шаблоны по праву считаются сегодня одной из наиболее популярных концепций объектно-ориентированного программирования. В этой книге на многочисленных примерах рассматриваются важные аспекты применения шаблонов проектирования, которые не были должным образом освещены в знаменитой книге "Design Patterns" (Джон Влиссидес является одним из ее соавторов). Здесь представлены вариации уже известных шаблонов, а также новые шаблоны. Кроме того автор изложил свое понимание процесса разработки шаблонов и предложил ряд рекомендаций начинающим разработчикам. Книга предназначена для специалистов и предполагает определенный уровень знакомства с шаблонами проектирования и языком C++.

ББК 32.973.26-018.2.75

ISBN 5-8459-0393-9 (рус.)
ISBN 0-2014-3293-5 (англ.)
1998

Оглавление

Вступление 9
Предисловие 11
Глава 1. Введение 13
Глава 2. Проектирование при помощи шаблонов 21
Глава 3. Темы и вариации 57
Глава 4. Муки творчества 105
Глава 5. Разработка шаблонов: семь правил успеха 125
Библиография 131
Предметный указатель 134

Содержание

Вступление 9
Предисловие 11
Глава 1. Введение 13
Десять основных заблуждений 14
Наблюдения 20
Глава 2. Проектирование при помощи шаблонов 21
Основные положения 21
Прикрепление и открепление объектов 25
Заменители 29
Правила посещения объектов 33
Проблемы, связанные с применением шаблона VISITOR 39
Защита в однопользовательской системе 40
Защита в многопользовательской среде 45
Краткий итог 54
Глава 3. Темы и вариации 57
Ликвидация объекта Singleton 57
Проблемы, связанные с шаблоном OBSERVER 65
Еще раз о VISITOR 71
Шаблон Generation Gap 75
Стирание типов 89
Борьба с утечками памяти 95
Pull- и push-модели 99
Глава 4. Муки творчества 105
Глава 5. Разработка шаблонов: семь правил успеха 125
Правило 1. Выделять время для размышлений 125
Правило 2. Приверженность структуре 126
Правило 3. Как можно раньше и чаще рассматривать конкретные примеры 127
Правило 4. Шаблоны должны быть разными и взаимно дополняющими 128
Правило 5. Удачное представление 128
Правило 6. Неустанные итерации 129
Правило 7. Собирать и учитывать отклики 129
Универсального рецепта нет 130
Библиография 131
Предметный указатель 134

Вступление

Наше сотрудничество началось с того, что однажды Джон (John Vlissides) попросил меня подготовить материалы для его рубрики, посвященной шаблонам проектирования, в журнале C++ *Report.* После того как я подготовил материалы для этой рубрики в пяти последующих выпусках журнала, Стэн Липпман (Stan Lippman) предложил мне сменить Джона на посту ее ведущего. Джон занялся исключительно шаблонами проектирования, а я стал уделять больше внимания своей рубрике. Таким образом я оказался частью своеобразного дуэта, призванного информировать программистов, работающих на C++, о новинках в области шаблонов проектирования. Мне импонирует подход к данной теме, который исповедует Джон. Однажды я написал ему:

> "Аналогия с выведением птенцов очень точно отражает суть процесса создания шаблонов[1]. Я только что перечитывал предисловие ко втором)' изданию книги Александера (Alexander) *Notes on Synthesis.* Очевидно, он также пришел к выводу, что речь идет об обнаружении имеющихся естественных закономерностей, которые ждут своего открытия, а не о "методах", позволяющих эти закономерности создавать."

Для меня большая честь сотрудничать с одним из членов "банды четырех"[2]. Возможно, вы бы так ничего и не узнали о шаблонах проектирования, если бы не книга этих авторов *Design Patterns (Шаблоны проектирования),* в которой изложены основополагающие понятия данной дисциплины. Предложенные ими шаблоны проектирования являются тем фундаментом, на котором основаны все современные разработки специалистов в этой области. Книга Джона Влиссидеса позволит непосредственно ознакомиться с ходом мыслей одного из членов "банды четырех", а также получить представления о процессе разработки шаблонов в целом.

Прежде чем удается получить хороший шаблон, приходится преодолеть немало трудностей. В своей книге Джон рассматривает некоторые сложные вопросы, которые не вошли в знаменитую книгу "банды четырех". Например, он показывает, что возможности шаблона VISITOR весьма ограничены, если исходная структура классов постоянно меняется. Некоторые уже разработанные шаблоны (например, GENERATION GAP) в свое время не вошли в книгу *Design Patterns,* но они вполне заслуживают, чтобы их опубликовали (шаблон GENERATION GAP рассматривается в главе 3). Вы узнаете, как в "банде четырех" происходило обсуждение шаблона MULTICAST. По этом)' поводу Джон заметил: "Те, кто

---

[1] *Оригинальное название книги - Pattern Hatching- в буквальном переводе означает "вынашивание, выведение шаблонов".* -Прим. перев.

"Банда четырех" (Gang of Four- GoF) - так называют четверку авторов основополагающей книги по шаблонам проектирования Design Patterns: Elements of Reusable Object-Oriented Software, *членами которой являются Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides).* - Прим. перев.

склонен приписывать нашей четверке экстраординарные возможности, будут шокированы совершенной хаотичностью нашего процесса разработки шаблонов." Данная книга содержит очень важную *мысль,* которой в более академичной и строгой книге *Design Patterns* не уделено должное внимание: шаблоны создаются обыкновенными программистами; не всегда удается сделать все безупречно с первой попытки, поэтому разработчики вынуждены бороться с прозой практики рекуррентного проектирования. Я надеюсь, что данная книга поможет пользователям шаблонов лучше оценить усилия, затраченные на разработку шаблонов "бандой четырех", а разработчикам шаблонов— подойти к пониманию и написанию шаблонов с большим смирением и усердием, чем прежде.

Проблема *порядка или хаоса* всегда присутствует в естественных науках и проектирование не является исключением. Создание шаблонов — это совместная работа людей над открытием и документированием конструкций, позволяющих облегчить жизнь другим разработчикам. Данная книга позволит проникнуть в естественный процесс, лежащий в основе создания шаблонов проектирования, понять ход мыслей обычных (но очень опытных и ответственных) разработчиков программного обеспечения, каждый из которых имеет собственное представление о проектировании. Книга *Design Patterns* представляет собой рафинированное изложение достигнутого авторами коллективного понимания шаблонов. В работе *Pattern Hatching* описывается процесс, с помощью которого это понимание было достигнуто, и значение данной книги в интерпретации позиции GoF не следует недооценивать. Я хотел бы привести здесь выдержку из письма, полученного мною от Ричарда Хелма (Richard Helm) в конце 1997 г., которая подтверждает мою точку зрения:

> "Разработанные GoF шаблоны проектирования учитывают только аспекты микроархитектуры. Необходимо правильно выбрать макроархитектуру: разбиение на уровни, распределение, изоляцию функций..., а также нано-архитектуру: инкапсуляцию, принцип подстановки Лисков (Barbara Liskov). Возможно, кое-где удастся использовать некий шаблон проектирования, во всяком случае, весьма маловероятно, что такой шаблон уже разработан и описан в какой-нибудь книге."

Предлагаемая книга поможет понять, как использовать книгу GoF (а также любой набор шаблонов) в качестве ценного руководства, а не обременительного предписания. Она поможет определить, какое место занимают шаблоны среди основных принципов объектно-ориентированного проектирования. В этой книге описан итеративный процесс, а также неформальные, но весьма действенные правила, которые привели к созданию 23-х шаблонов, описанных в книге GoF *Design Patterns.* Полезно знать, как именно происходит разработка шаблонов, поскольку это позволяет соотнести шаблоны с реальностью, где действуют прагматичные правила повседневной жизни. Я надеюсь, что читатель осознает важность привязки шаблонов к решаемой проблеме и постарается воспользоваться своим собственным интеллектом, а не будет слепо следовать формализму, описанному в какой-то книге. Не думаю, что книга Влиссидеса понравится теоретикам, но настоящие программисты несомненно ее оценят.

Джеймс О. Коплин
(James O. Coplien)
Lucent Technologies
Bell Labs Innovations

Предисловие

Я никогда не забуду чувства, которые испытал в то осеннее утро 1994 г., когда получил от главного редактора журнала C++ *Report* Стена Липпмена (Stan Lippman) электронное письмо с предложением вести в его журнале рубрику, которая будет выходить каждые два месяца.

Мы были просто знакомыми, причем познакомились в том же году, когда Стен посетил лабораторию Watson Lab. Тогда мы кратко поговорили о его работе над автоматическими средствами разработки и о работе GoF над шаблонами проектирования. В отличие от большинства программистов того времени Стен был уже знаком с понятием шаблонов — он прочитал серию препринтов о шаблонах проектирования *{Design Patterns)* и высказался о них с одобрением. Наш краткий диалог достаточно быстро перешел к теме искусства написания программ в целом; при этом я испытал настоящее чувство гордости: Стен, известный публицист и автор двух очень удачных книг, делился своими мыслями со мной, безвестным аматором, Я не знал, понравились ли ему мои суждения или его реакция была обусловлена терпимостью, пока не получил упомянутое приглашение к сотрудничеству. (Я слышал, что с терпимостью Стена может сравниться только его искренность!)

Мои впечатление от нашего знакомства не могут идти ни в какое сравнение с реакцией на полученное спустя несколько месяцев электронное сообщение. Я испытывал странное чувство приподнятости и страха при мысли о том, что мне предстоит регулярно писать для интернациональной аудитории. Удастся ли мне продержаться хотя бы пару отдельных выпусков? Будет ли людям интересно то, о чем я напишу? Что я должен сказать? И если я это скажу, помогут ли мои рекомендации кому-нибудь?

Около часа я предавался своим страхам, но затем вспомнил предостережения своего отца: не следует слишком увлекаться самоанализом, это парализует. Необходимо сосредоточиться на главном, а остальное приложится.

Итак, я решился,

Выбрать тему было достаточно просто. К этому моменту я уже около трех лет достаточно серьезно занимался исследованиями в сфере шаблонов проектирования. Мы только что завершили работу над книгой *Design Patterns,* но все понимали, что это еще далеко не последнее слово в данной области. Рубрика предоставляла большие возможности для разъяснения содержащегося в книге материала, дальнейшего развития этой темы и решения возникающих вопросов. Кроме того, поддержка в рубрике высказанных в книге идей способствовала бы *росту* продаж, что также совсем неплохо.

Теперь, когда рубрика *Pattern Hatching* просуществовала уже более двух лет, можно сказать, что мои сомнения были напрасны. Я ни разу не испытывал затруднения, выбирая, о чем писать, и всегда получал удовольствие от этой работы. В редакцию поступило множество заинтересованных откликов от читателей со всего мира, в том числе запросы на старые выпуски журнала. Это натолкнуло меня на мысль собрать

воедино материалы моей рубрики, а также другую полезную информацию, которая все еще оставалась неопубликованной.

Именно эту цель я преследовал при написании книги "Применению шаблонов проектирования: дополнительные штрихи". Здесь вы найдете размышления и идеи первых трех лет моей карьеры в качестве ведущего рубрики (все, что я опубликовал в журналах *C++ Report и Object Magazine,* а также новые интересные идеи). Изложение организовано в логическом, а не временном порядке. Работу по систематизации облегчало то, что многие статьи входили в различные серии, тем не менее, это потребовало определенных усилий. Надеюсь, что вы будете удовлетворены результатом.

Благодарности

Многие люди оказывали мне самую разнообразную помощь, и я должен поблагодарить их за это. Наиболее существенную поддержку я получил от моих друзей, членов "банды четырех" — Эрика Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm) и Ральфа Джонсона (Ralph Johnson). Каждый из них на определенных этапах работы вносил дельные предложения, что заметно повлияло на данную книгу (и сделало ее гораздо лучше). Редкая удача работать с людьми, которые настолько дополняют друг друга, и я всем им очень признателен.

Однако они были не единственными помощниками. Целая группа людей просматривала мои сырые наброски в поиске нестыковок и разных ляпов. В эту группу входили Брюс Андерсон (Bruce Anderson), Бард Блюм (Bard Bloom), Фрэнк Бушман (Frank Buschmann), Джим Коплин (Jim Coplien), Рэй Кризостомо (Rey Crisostomo), Уим Де Пау (Wim De Pauw), Кирк Кноерншилд (Kirk Knoernschild), Джон Лакос (John Lakos), Даг Ли (Doug Lea), Боб Мартин (Bob Martin), Дирк Рилль (Dirk Riehle), Даг Шмидт (Doug Schmidt) и Пери Тарр (Peri Tarr). Особую благодарность я хочу выразить Джиму (Jim Coplien), моему второму "я" в *Report,* за его доброжелательное предисловие к этой книге, а также за поддержку в работе.

Выражаю свою признательность также тем, кто присылал мне по электронной почте вопросы, комментарии, исправления и аргументированные возражения. Таких людей очень много, назову только тех из них, кого я цитирую или чьи комментарии вошли в эту книгу: Марк Бетц (Mark Betz), Лорион Барчел (Laurion Burchall), Крис Кларк (Chris Clark), Ричард Джайгер (Richard Gyger), Майкл Хиттесдорф (Michael Hittesdorf), Майкл МакКоскер (Michael McCosker), Скотт Мейерз (Scott Meyers), Тим Пирлз (Tim Pierls), Пол Пелетье (Paul Pelletier), Ранджив Шарма (Ranjiv Sharma), Дэвид Ван Кэмп (David Van Camp), Джеролф Уэндленд (Gerolf Wendland) и Барбара Зино (Barbara Zino). Хотя я и не назвал остальных помощников, я очень признателен им за их вклад.

Наконец, я хочу поблагодарить всех своих сотрудников, а также близких мне людей за оказанную поддержку. Я в большом долгу перед вами.

Хоторн, Нью-Йорк — Джон Влиссидес (John Vlissides)
Январь 1998 — vlis@watson.ibm.com

Глава 1

Введение

Если вы не знакомы с книгой *Design Patterns: Elements of Reusable Object-Oriented Software* [GoF95], то ее следует прочесть, прежде чем продолжить чтение данной книги. Аналогично нужно поступить и в том случае, если вы в свое время читали эту книгу, но недостаточно внимательно *изучили* ее.

Итак, далее я предполагаю, что вы не относитесь ни к одной из упомянутых выше категорий читателей, т.е. вам кое-что известно о шаблонах вообще и о нашем наборе из 23-х шаблонов проектирования в частности. Только в этом случае можно извлечь пользу из данной книги, так как она дополняет, обновляет и уточняет материалы, изложенные в *Design Patterns.* Если же вы не знакомы с вышеупомянутыми 23 шаблонами так называемой "банды четырех" *(Gang of Four - GoF),* то вам будет крайне сложно понять предлагаемый здесь материал. Лучше всего, если *Design Patterns* все время будет у вас под рукой при чтении данной книги. Наконец, я предполагаю, что вы знакомы с языком C++, тем более, что аналогичное предположение содержится и в *Design Patterns.*

А теперь в качестве проверки попробуйте с помощью 25 или меньшего количества слов описать назначение шаблона COMPOSITE.

Если вам без труда удалось выполнить это упражнение, то при чтении книги у вас не возникнет проблем. Если вы знаете о назначении данного шаблона, но затрудняетесь описать его словами, не стоит беспокоиться — эта книга предназначена и для вас.

Если же вы исписали целый лист, то вам стоит воспользоваться советом в самом начале параграфа: отложите эту книгу, возьмите *Design Patterns* и прочитайте до конца раздел, посвященный реализации данного шаблона. Необходимо также прочитать соответствующие разделы, касающиеся других шаблонов. После этого вы будете достаточно подготовлены, чтобы извлечь пользу из чтения данной книги.

Почему я выбрал такое название для своей книги? Все удачные названия уже были задействованы, и первоначально я просто написал *Pattern Hatching* (в книге рассматриваются дополнительные аспекты использования шаблонов, а одно из значений слова hatching — штриховка). Однако вскоре я обнаружил, что это название достаточно точно отражает мое понимание темы. Другое значение слова hatching (выведение птенцов) подразумевает, что речь идет не о создании чего-либо нового, а скорее о развитии имеющихся зачатков. Можно сказать, что наша книга *Design Patterns* — это коробка с яйцами, из которых, возможно, возникнет новая жизнь[1].

В своей книге я не буду просто пересказывать *Design Patterns.* Я хочу оживить ее концепции, показать, как они применяются на практике, и сделать их наиболее полезными

---

[1] *В дальнейшем мы не будем злоупотреблять этой аналогией.*

для читателя. Вы научитесь определять, какие шаблоны следует, а какие не следует применять в зависимости от конкретных обстоятельств, получите более глубокое представление о некоторых из наших исходных шаблонов и ознакомитесь с процессом создания новых шаблонов. Книга изобилует примерами. Некоторые из них — это проверенные временем конструкции, другие — так называемые "полуфабрикаты". Еще одна категория примеров — чисто иллюстративные — представляет собой полностью "бумажные" проекты, которые, скорее всего, не вынесут столкновения с жестокой реальностью, но могут содержать зерна более устойчивых конструкций.

Предлагаю все это вашему вниманию, искренне желая повысить ваши способности в сфере проектирования, уровень знаний о шаблонах, а также расширить взгляд на разработку программного обеспечения в целом. В книге собран мой опыт использования шаблонов. Надеюсь, что он пригодится и вам.

Десять основных заблуждений

Вместе с тем шумом, который поднялся вокруг шаблонов проектирования в настоящее время, возникло немало недоразумений, опасений и неверной информации. Отчасти это явилось отражением новизны данного направления для большинства разработчиков программного обеспечения (хотя, строго говоря, это направление не такое уж новое). Оно быстро развивается, вследствие чего ощущается недостаток фактов. Наконец, несмотря на то, что о шаблонах написано немало книг [BMR+96, Coplien96, CS95, GoF95, MRB98, VCK96], создатели шаблонов действительно заслужили определенные обвинения со стороны недостаточно подготовленной аудитории.

Поэтому я чувствую себя обязанным развеять наиболее распространенные заблуждения, которые мне приходилось слышать настолько часто, что их можно считать своего рода шаблонами. Я даже забавлялся, используя форму шаблона для их описания... пока меня не осенила мысль, что сводить все к шаблонам — тоже заблуждение!

Внимательно изучив мнения различных людей о шаблонах на протяжении ряда лет, я пришел к выводу, что основные заблуждения можно отнести к одной из трех категорий: заблуждения относительно того, *что* собой представляют шаблоны; заблуждения относительно того, что они *могут делать,* и заблуждения относительно тех, кто эти шаблоны предлагает. Каждое заблуждение из моей "лучшей десятки" также попадает в одну из этих категорий. Начнем с рассмотрения заблуждений, относящихся к первой категории.

Заблуждение 1:
"Шаблон — это решение определенной проблемы в определенных обстоятельствах."

Данное определение принадлежит Кристоферу Александеру (Christopher Alexander, [AIS+77]), поэтому критические высказывания о нем могут некоторым показаться ересью. Однако простой пример поможет прояснить, что я имею ввиду.

Проблема: Как получить выигрыш по лотерейному билету до истечения срока его предъявления?
Ситуация: Собака съела билет за час до истечения указанного срока.
Решение: Вскрыть собаку, извлечь билет и бежать к ближайшему пункту выплаты выигрышей.

Это и есть решение проблемы в определенных обстоятельствах. Но это не шаблон. Чего же не хватает? По крайней мере трех составляющих:

1. Повторяемости, которая делает решение применимым в ситуациях, отличных от рассматриваемой.
2. Обучения, которое дает понимание, как данное решение связано с конкретным вариантом проблемы. (В реальных шаблонах обучающая часть состоит главным образом в описании и выделении движущих сил и/или последствий его применения.)
3. Имени, с помощью которого можно ссылаться на данный шаблон.

Как свидетельствует продолжающаяся дискуссия в Интернет-форуме *{patterns-discussion@cs.uiuc.edu*[2]*,* до сих пор не удалось дать исчерпывающее определение шаблона. Дело усложняется тем, что шаблон — это и конкретная сущность, и описание аналогичных сущностей. Один из способов различать их состоит в том, чтобы использовать термин *шаблон* по отношению к описанию, а термин *экземпляр шаблона* — по отношению к конкретной реализации этого шаблона.

Разработка определений для терминов — неблагодарный труд, поскольку определение, понятное одной аудитории (скажем, программистам), может показаться полностью бессмысленным другим специалистам. Я не ставлю перед собой задачу дать исчерпывающее определение шаблона. Достаточно сказать, что в любом определении, перечисляющем составные части шаблона, наряду с проблемой, решением и контекстом необходимо сказать о повторяемости, обучении и именовании.

Заблуждение 2:
"Шаблоны — это просто жаргон, правила, приемы программирования, структуры данных..."

Я называю это "отторжением вследствие недооценки". Стремление свести нечто непознанное к известному является совершенно естественным, тем более, если вы не заинтересованы в его исследовании. Кроме того, люди слишком часто помещают старое вино в новую упаковку и называют это новшеством, поэтому настороженное отношение ко всяким новинкам вполне оправдано.

В данном случае отторжение не является следствием опыта. Зачастую оно основано на поверхностном знакомстве и изрядной доле цинизма. В действительности, нет ничего полностью нового; человек создавал шаблоны в своей голове с тех пор, как у него появилась голова. Новое — в том, что мы предложили именовать шаблоны и описывать PIX.

Относительно приведенных высказываний. Действительно, существует определенный жаргон, связанный с шаблонами, в который входят термины "шаблон", "движущие силы", используемое Александером (Alexander) "качество без названия" и т.д. Но было бы неверно сводить шаблоны проектирования к жаргону! По сравнению с остальными областями информатики в данной области не так уж много новых терминов. И это весьма симптоматично. Хорошему шаблону внутренне присуща доступность для тех, кому он предназначен, в нем могут использоваться термины целевой области, но вряд ли возникнет необходимость в специальной терминологии из области шаблонов.

---

[2] *Чтобы подписаться, нужно послать электронное сообщение по адресу* patterns-discussion-request@cs.uiuc.edu, *указав единственное слово "subscribe "(без кавычек!) в качестве темы сообщения.*

Шаблоны не являются правилами, которые можно бездумно применять (об этом свидетельствует наличие обучающего компонента), они также не сводятся к приемам программирования, несмотря на то, что "идиоматическое" направление данной дисциплины занимается зависящими от языка программирования шаблонами. "Приемы" — также слишком узкое определение, поскольку в нем преувеличивается значение решения за счет недооценки проблемы, контекста, обучения и именования.

Вы, без сомнения, слышали о трех стадиях принятия любого новшества: сначала оно отрицается, как не содержащее ничего ценного, затем— считается нежизнеспособным и, наконец, становится очевидным и тривиальным — "Мы всегда так делали". Шаблоны к настоящему времени еще находятся на первой стадии своего развития.

Заблуждение 3:
"Достаточно посмотреть на один из них, чтобы получить представление об остальных."

Стричь всех под одну гребенку неправильно, однако по отношению к шаблонам это неправильно вдвойне. Существует широкий диапазон всевозможных шаблонов, которые различаются областью применения, содержимым, масштабом, стилем и качеством — достаточно пролистать одну из книг серии *Pattern Languages of Program Design* [CS95, MRB98, VCK96], чтобы убедиться в этом. Шаблоны *столь* же различны, как и люди, которые их пишут, а может быть даже больше. Такие авторы, как Элистер Кок-борн (Alistair Cockburn), Джим Коплин (Jim Coplien), Нэйл Харрисон (Neil Harrison) и Ральф Джонсон (Ralph Johnson) смогли выйти за рамки написания шаблонов определенного стиля для отдельных предметных областей, поэтому было бы неправильно судить о шаблонах в целом, рассмотрев лишь несколько примеров.

Заблуждение 4:
"Для успешного применения шаблонов необходимы соответствующие инструментальные средства или методологическая поддержка."

Последние пять лет я занимался созданием и применением шаблонов проектирования, помогал использовать их другим, а также участвовал в подготовке по меньшей мере одного основанного на шаблонах инструментального средства [BFV+96], поэтому могу со всей ответственностью заявить, что выигрыш при применении шаблонов получается в основном за счет самих шаблонов, т.е. они не нуждаются ни в какой специальной поддержке.

Обычно я указываю четыре основных преимущества шаблонов:

1. Они позволяют суммировать опыт экспертов и сделать его доступным рядовым разработчикам.
2. Имена шаблонов образуют своего рода словарь, который позволяет разработ чикам лучше понимать друг друга.
3. Если в документации системы указано, какие шаблоны в ней используются, это позволяет читателю быстрее понять систему.
4. Шаблоны упрощают реструктуризацию системы независимо от того, использо вались ли шаблоны при ее проектировании.

Долгое время я считал главным первое из названных преимуществ. Теперь я понимаю, что второе не менее важно. Представьте, сколькими байтами информации обмениваются разработчики (не важно, в вербальной или электронной форме) в процессе работы над проектом! Я думаю, что это мегабайты, если не гигабайты. (У меня хранится несколько десятков мегабайт электронной корреспонденции, которой обменивались

участники "банды четырех" в процессе написания книги *Design Patterns.* По приблизительным оценкам, такая работа эквивалентна среднему проекту разработки программного обеспечения.) При такой интенсивности обмена информацией все, что может сделать этот обмен хоть немного более эффективным, приведет к существенной экономии времени. Таким образом, шаблоны проектирования способствуют повышению интенсивности человеческого общения. Моя оценка преимуществ 3 и 4 также повышается со временем, особенно по мере роста масштабов проектов и увеличения продолжительности жизни их программного обеспечения.

Короче говоря, шаблоны проектирования — это главным образом пища для ума, а не материал для инструментального средства. Выгода от методологической или инструментальной поддержки также возможна, но это только розочки на торте, а не сам торт, и даже не слой крема на нем.

Рассмотренные до сих пор заблуждения касались того, что представляют собой шаблоны. Теперь перейдем к заблуждениям относительно возможностей шаблонов. Речь может идти как о недооценке, так и о переоценке этих возможностей.

Заблуждение 5:
"Шаблоны гарантируют возможность повторного использования программного обеспечения, повышение производительности, отсутствие разногласий и т.п."

Это заблуждение очевидно, поскольку шаблоны *вообще ничего не гарантируют.* Можно говорить только о вероятности получения неких преимуществ. Шаблоны никоим образом не могут заменить человека в творческом процессе. Они просто предоставляют дополнительные возможности недостаточно опытному или малоинициативному, но в целом толковому разработчику.

Хороший шаблон вызывает восторженную реакцию у пользователей, но это может произойти только тогда, когда данный шаблон встречает отклик читателя. Если же этого не происходит, шаблон напоминает говорящее дерево в лесу, где его никто не слышит. Как бы хорошо ни был написан шаблон, от него в этом случае не будет толку.

Шаблоны существенно отличаются от других элементов арсенала разработчика. Не стоит слишком полагаться на них. Меньше пообещать и больше сделать — вот лучшая защита от неоправданного оптимизма и последующего разочарования.

Заблуждение 6:

"Шаблоны порождают готовые архитектурные решения."

Это заблуждение аналогично предыдущему, только менее категорично по форме.

Порождающие аспекты шаблонов периодически обсуждаются в соответствующих Internet-форумах. Насколько я понимаю, под порождающей способностью подразумевается способность шаблона создавать результирующее поведение, т.е. некий шаблон помогает решать проблемы, для которых он изначально не был предназначен. В некоторых работах утверждается, что истинная порождающая способность носит объективный характер, и нужное поведение формируется практически независимо от желания пользователя шаблона.

Порождающая способность в основном заключена в частях шаблона, предназначенных для обучения — там, где рассматриваются движущие силы или последствия. Это особенно полезно при определении и уточнении архитектуры. Но сами по себе шаблоны ничего не создают, это делают люди и только в том случае, если они достаточно квалифицированы, а используемые ими шаблоны достаточно эффективны. Более того,

вряд ли шаблоны покрывают все аспекты архитектуры. Любой нетривиальный проект содержит множество аспектов проектирования, для которых не существует шаблонов. Возможно, эти аспекты не являются стандартными или повторяющимися, или же шаблоны для них еще только предстоит написать. В любом случае, вам придется проявить свои творческие способности и заполнить пробелы между шаблонами самостоятельно.

Заблуждение 7:
"Шаблоны предназначены для объектно-ориентированного проектирования и реализации."

Это заблуждение относится к разряду тех, где возможности шаблонов чересчур ограничиваются.

Шаблоны не представляют собой никакой ценности, если они не содержат практического опыта. Какой опыт будет заключен в шаблоне, зависит от его разработчика. Безусловно, в объектно-ориентированном проектировании существует огромный опыт, но не менее данные наработки имеются в проектировании, не основанном на объектно-ориентированной парадигме, и не только в проектировании, но и в анализе, сопровождении, тестировании, документировании, разработке организационной структуры и т.д. В настоящее время появляются шаблоны и в этих областях. Уже вышло по крайней мере две книги, посвященные шаблонам анализа [Fowler97, Hay96], а каждая конференция по языкам шаблонов программ (Pattern Languages of Programs — PLoP) открывает все новые виды шаблонов. (Например, на конференции 1996 года рассматривались шаблоны для создания музыкальных композиций!)

В каждом заблуждении содержится рациональное зерно. В данном случае, если проанализировать используемые формы шаблонов, можно выделить вариации двух основных стилей: строго структурированного стиля, предложенного "бандой четырех" в книге *Design Patterns,* и почти беллетристического стиля Кристофера Алексан-дера (Christopher Alexander), практически лишенного структуры и напоминающего простой комментарий. Занявшись написанием шаблонов в сферах, отличных от объектно-ориентированного проектирования, я понял, насколько предложенный "бандой четырех" стиль привязан к своей области. Эта схема совершенно не срабатывает в других областях, где я пытался ее применить. Что такое структурная диаграмма для идиомы C++? Реализационные компромиссы в шаблоне музыкальной композиции? Взаимодействия в шаблоне для составления описаний?

Очевидно, что определенный формат не может охватить все разнообразие шаблонов. Действительно всеобъемлющим является понятие шаблона как средства сбора и передачи опыта в любой сфере деятельности.

Заблуждение 8:
"Нет доказательств того, что шаблоны хоть кому-нибудь помогают."

В прошлом это заблуждение имело под собой основания, но сейчас ситуация изменилась. О преимуществах, полученных при использовании шаблонов, сообщается в журналах (в частности, в журнале *Software-Practice and Experience* [Kotula96]) и на конференциях (OOPSLA [HJE95, Schmid95] и ICSE [BCC+96]). Даг Шмидт (DougSchmidt) указал на определенные преимущества от применения шаблонов при обучении информатике новичков и старшекурсников [PD96]. Хотя большинство доказательств носит качественный характер, мне известна по крайней мере одна группа, которая проводит эксперименты с целью получить количественные результаты [Pretcheit97, PUS97].

С течением времени станут более понятны преимущества и опасности, возникающие при использовании шаблонов. Хотя первые результаты весьма обнадеживающие, для более тщательной оценки необходим дополнительный опыт. Тем не менее, было бы неправильно отказываться от шаблонов на том основании, что выгоды от их использования не измерены количественно.

От ошибочных суждений относительно возможностей шаблонов перейдем к рассмотрению двух последних заблуждений, которые касаются не самих шаблонов, а тех, кто эти шаблоны создает и поддерживает идеи их использования.

Заблуждение 9:
"Сообщество, поддерживающее идеи применения шаблонов, представляет собой узкий элитарный круг."

Интересно узнать, откуда возникло это заблуждение, поскольку на самом деле для сообщества приверженцев идеи использования шаблонов характерна максимальная открытость. Достаточно посмотреть на данные об участниках конференций PLoP: это люди со всего мира, из больших корпораций и крохотных фирм; аналитики, проектировщики и разработчики; студенты и профессора; признанные авторитеты и неоперившиеся юнцы. Я был крайне удивлен, когда узнал, что некоторые регулярные участники даже не являются специалистами в области информатики! Сообщество постоянно находится в движении, состав участников конференции год от года заметно меняется.

При таком обилии публикаций некоторые могут удивиться определенному недостатку академичности. Дело в том, что большинство участников конференций PLoP — практики. По-видимому, так и должно быть. Никто из известных разработчиков первых шаблонов программного обеспечения— включая Кента Бека (Kent Beck), Питера Коуда (Peter Coad) и Уорда Каннингэма (Ward Cunningham) — не был выходцем из академических кругов. Только один из "банды четырех"— Ральф (Ralph Johnson) — академический ученый, и он самый лучший прикладник из всех ученых, которых я знаю. Таким образом, сама природа сообщества приверженцев шаблонов опровергает все подозрения относительно его однородности и элитарности.

Заблуждение 10:
"Сообщество приверженцев шаблонов преследует корыстные цели."

Мне неоднократно приходилось слышать обвинения в том, что шаблоны служат источником наживы для тех, кто пишет о них книги. Иногда даже утверждают, что "движение" приверженцев шаблонов имеет низменные цели.

Какая чушь!

Все члены "банды четырех" не менее других были удивлены реакцией читателей на появление книги *Design Patterns*. Никто из нас не был готов к ажиотажу, вызванному ее представлением на OOPSLA '94, даже издатели были поражены таким высоким спросом. При работе над книгой мы старались сделать ее как можно лучше. Мы были слишком заняты содержанием, чтобы задумываться о том, как эта книга будет продаваться.

Теперь, когда термин "шаблон" стал популярным, кое-кто пытается использовать это слово не совсем в альтруистических целях. Но если внимательно читать работы ведущих специалистов в области шаблонов, можно заметить, что всех авторов объединяет стремление собрать и передать другим приобретенный опыт, лучшие

практические наработки, имеющиеся преимущества — результат многих лет работы — и не просто описать их, но и *приобщить к* ним всех читателей.

Именно желание облегчить жизнь разработчиков движет настоящими авторами работ по шаблонам. Другие мотивы пагубны и ведут к неправильным представлениям о шаблонах.

Наблюдения

Как правило, люди реагируют на шаблоны проектирования одним из двух способов, которые я попытаюсь проиллюстрировать с помощью аналогии.

Представьте себе электронщика-любителя, который, не имея формального образования, сумел за несколько лет разработать и сделать уйму полезных, приспособлений: любительское радио, счетчик Гейгера, домашнюю сигнализацию и т.д. И вот однажды любитель решает, что пора добиться официального признания своего таланта, пройти курс обучения и получить специальность электронщика. Приступив к учебе, он поражается тому, насколько ему знаком материал. Конечно, ему знакома не терминология или способ изложения, а лежащие в основе концепции. Электронщик-любитель видит названия и усовершенствования тех вещей, которыми он пользовался годами. Для него это просто одно откровение за другим!

Теперь представим себе новичка, который посещает те же занятия и изучает тот же материал. Он много знает о роликовых коньках, но не имеет представления об электронике. Ему будет очень сложно усваивать предлагаемый материал не потому,, что он тупой, а потому, что этот предмет ему совершенно незнаком. Новичку потребуется гораздо больше времени, чтобы понять и оценить материал. Но он все-таки может это сделать, если проявит настойчивость и упорно поработает.

Если вы можете сравнить себя с любителем шаблонов — тем лучше для вас. Но если вы ощущаете себя скорее новичком, не стоит расстраиваться: те усилия, которые вы потратите на изучение хороших шаблонов, будут окупаться всякий раз, когда вы будете применять эти шаблоны в своих проектах. Я вам это обещаю.

Возможно, электроника с ее чисто техническим содержанием — не лучшая аналогия. Альфред Норт Уайтхед (Alfred North Whitehead) сказал в 1943 году:

> "Искусство — это наложение шаблона на опыт и наше эстетическое наслаждение от узнавания этого шаблона..."

Хотя эта фраза была сказана совсем по другому поводу, она может иметь к нашему случаю самое непосредственное отношение.

Глава 2

Проектирование при помощи шаблонов

Лучший способ получить представление о применении шаблонов — это попытаться их использовать. Трудность состоит в выборе понятного всем примера. Люди всегда больше заняты своими собственными проблемами, но пример, способный заинтересовать определенную аудиторию, скорее всего окажется весьма специфическим и его будет сложно понять неспециалистам.

Поэтому давайте рассмотрим, как проектируется то, с чем знаком каждый пользователь компьютера, а именно —- иерархическая файловая система. Мы не будем заниматься вопросами низкоуровневой реализации, такими как буферизация ввода-вывода И управление секторами диска, а разработаем модель, которую используют создатели приложений — интерфейс прикладных программ (API) файловой системы. В большинстве операционных систем интерфейс прикладных программ состоит из нескольких десятков обращений к процедурам и определенных структур данных, причем возможности его расширения отсутствуют или весьма ограничены. Наш проект будет полностью объектно-ориентированным и расширяемым.

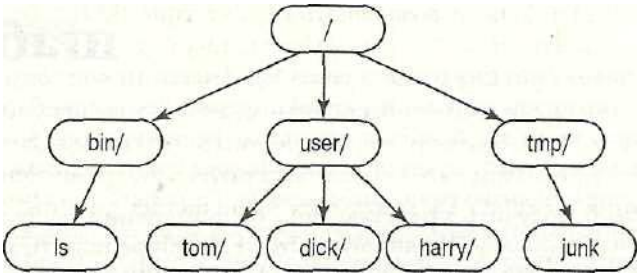
Первым делом остановимся на двух исключительно важных задачах проектирования и шаблонах, с помощью которых эти задачи решаются, а затем рассмотрим на примере, как применяются при проектировании другие шаблоны. Цель состоит не в том, чтобы предложить жестко регламентированный процесс применения шаблонов или продемонстрировать наилучший способ проектирования файловой системы, а в том, чтобы помочь вам научиться самостоятельно применять шаблоны. Работая с шаблонами и наблюдая за их использованием, вы научитесь свободно оперировать ими и найдете свой собственный стиль применения.

Основные положения

С точки зрения пользователя, файловая система должна обрабатывать файловые структуры произвольной сложности и размера, не налагая ограничения на их ширину и глубину. С точки зрения программиста, представление файловой структуры должно быть таким, чтобы его можно было легко обрабатывать и расширять.

Рассмотрим реализацию команды, которая выводит перечень файлов некоторого каталога. Программный код для извлечения имени каталога не должен отличаться от кода для извлечения имени файла — в обоих случаях работает один и тот же код. Другими словами, нужно иметь возможность одинаково трактовать каталоги и файлы при извлечении их имен — тогда будет проще написать код и осуществлять его поддержку. Кроме того, при желании можно будет ввести новые типы файлов (например, символические связи), не переделывая половину системы.

Из сказанного следует два вывода: основными элементами проблемной области являются файлы и каталоги; необходимо иметь возможность конкретизировать эти элементы после окончания проектирования. Очевидный подход к проектированию состоит в том, чтобы представить данные элементы в виде объектов:



Как же реализовать такую структуру? Имеется два вида объектов, следовательно, должно быть два класса— класс файлов (File) и класс каталогов (Directory). Если нужно в определенных случаях трактовать файлы и каталоги одинаковым образом, необходимо, чтобы они имели одинаковый интерфейс. Это означает, что соответствующие классы являются производными классами некоего общего (абстрактного) базового класса, который мы назовем *Node.* Наконец, известно, что файлы содержатся в каталогах.

Все эти ограничения задают следующую иерархию классов:

```
class Node { public:
    //здесь декларируется общий интерфейс
    protected: Node () ;
    Node (const Node&) ;

class File : public Node {
public:
    File();
    //здесь переопределяется общий интерфейс
};
Class Directory : public Node {
public:
    Directory();
    //здесь   переопределяется общий интерфейс private:
    list<Node*> nodes;
};
```

Следующий вопрос касается структуры общего интерфейса: какие операции одинаково применимы к файлам и каталогам?

Существуют всевозможные общие атрибуты, такие как имя, размер, режим доступа и т.д. Для каждого атрибута имеются операции доступа и модификации его значения(-й). Операции, имеющие понятный смысл как для файлов, так и для каталогов, легко трактовать единообразно. Сложности возникают, когда применимость операций к обоим случаям не так прозрачна.

Например, пользователь достаточно часто запрашивает список файлов определенного каталога. Это означает, что класс Directory должен иметь интерфейс, который позволит ему выводить перечень своих дочерних файлов. Ниже приводится простой интерфейс, возвращающий n-й дочерний файл:

```
virtual  Node*  getChild(int n) ;
```

Оператор getChild должен возвращать объект типа Node* (поскольку каталог может содержать объекты типа File и объекты типа Directory), вследствие этого приходится определять getChild не только для класса Directory, но и для всего класса Node, так как необходимо обеспечить возможность перечислить дочерние файлы подкаталога. Пользователю часто требуется спускаться вниз по файловой структуре. Невозможно делать это, сохраняя статическую типизацию, если нельзя применять операцию getChild к результату выполнения getChiid. Поэтому хотелось бы применять getChild единообразно, как и операции с атрибутами.

Операция getChild также предоставляет возможность рекурсивно определять операции класса Directory. Предположим, в классе Node декларирована операция size, которая возвращает общее число байтов, занятых деревом (поддеревом) каталога. В классе Directory можно определить свою версию этой операции как сумму значений, возвращенных дочерними файлами каталога при вызове операции size:

```
long Directory::size () {
    long total = 0;
    Node* child;

    for (int i = 0; child = getChild(i); ++i) {
        total += child->size();
    }
    return total;
}
```

Каталоги и файлы иллюстрируют ключевые аспекты шаблона COMPOSITE. Он позволяет задавать структурное дерево произвольной сложности и описывает, как единообразно трактовать объекты таких структур. Назначение шаблона COMPOSITE выглядит следующим образом:

> "Объединяет объекты в древовидные структуры с тем, чтобы представить иерархические отношения часть— целое, и дает клиентам универсальный способ работы с этими объектами независимо от их местоположения в иерархии."

В разделе "Применимость" данного шаблона указывается, что он используется в следующих случаях:

- чтобы представить иерархии объектов, связанных отношениями часть — целое;
- чтобы предоставить клиентам возможность игнорировать различия между
  простыми и составными объектами и позволить им трактовать все объекты по
  лученной структуры универсальным образом.

Структурный раздел данного шаблона представляет собой модифицированную OMT-диаграмму канонической (наиболее часто встречающейся) структуры классов шаблона COMPOSITE. Представить *исчерпывающее* определение множества классов и

связей невозможно, поскольку интерфейсы могут меняться в зависимости от конкретных проектов или реализационных компромиссов. (В шаблоне также содержатся аналогичные разъяснения.)

На рис. 2.1 показаны классы-участники шаблона COMPOSITE и их статические связи. Компонент (Component) — это абстрактный базовый класс, которому соответствует наш класс Node. Подклассы-участники могут быть листовыми объектами (Leaf), которым соответствуют файлы, и составными объектами (Composite), которым соответствуют каталоги. Стрелка, идущая от Composite к Component, показывает, что составной объект содержит экземпляры типа Component. Шарик на конце стрелки показывает, что этих экземпляров может быть несколько; если же шарика нет, то экземпляр должен быть один и только один. Ромб в начале стрелки означает, что Composite *агрегирует* свои дочерние экземпляры (т.е. при удалении составного объекта удаляются и его дочерние объекты), кроме того, компонент не может входить в несколько составных объектов, тем самым обеспечиваются строго древовидные структуры. Разделы шаблона "Участники" и "Взаимодействия" описывают соответственно статические и динамические связи между участниками.

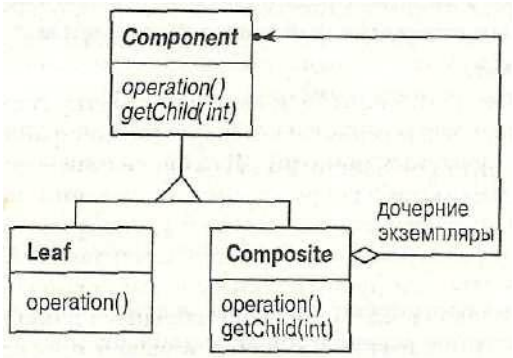


РИС. 2.1. *Структура шаблона COMPOSITE*

В разделе "Следствия" указываются преимущества и недостатки данного шаблона. К преимуществам следует отнести то, что шаблон COMPOSITE поддерживает древовидные структуры произвольной сложности. В результате сложность узла оказывается скрытой от клиентов: они не знают и не должны знать, с каким компонентом (листовым или составным) работают, благодаря этому код клиента становится более независимым от кода компонентов. Кроме того, клиент упрощается, поскольку он может универсальным образом трактовать листовые и составные объекты, ему больше не приходится решать, какой из множества путей кода выбрать в зависимости от типа компонента. Но самое главное — теперь можно добавлять новые типы компонентов, не затрагивая исходный код.

Недостаток шаблона COMPOSITE заключается в том, что его использование может приводить к созданию системы, в которой все классы объектов похожи друг на друга. Значительные различия всплывают только на стадии выполнения, в результате код будет трудно восприниматься даже теми программистами, кто хорошо разбирается

в реализациях классов. Более того, если шаблон применяется на низком уровне или при очень мелком разбиении, многие объекты могут оказаться запрещенными.

Из сказанного очевидно, что при реализации шаблона COMPOSITE необходимо рассмотреть множество вопросов:

- когда и где кэшировать информацию для повышения производительности;
- сколько памяти необходимо выделить для класса Component;
- какие структуры данных использовать для хранения дочерних объектов;
- следует ли в классе Component декларировать операции добавления и удаления дочерних объектов;
- и др.

При дальнейшей разработке файловой системы нам придется столкнуться с этими, а также со многими другими вопросами.

Прикрепление и открепление объектов

Проанализируем следствия применения шаблона COMPOSITE в нашем приложении. Сначала рассмотрим важный компромисс в проектировании интерфейса класса Node, а затем попытаемся добавить новые функциональные возможности к пока еще несовершенному проекту.

Использование шаблона COMPOSITE позволило создать костяк данного приложения и выразить основные характеристики иерархической файловой системы с помощью объектно-ориентированных понятий. Шаблон связывает свои классы-участники (Component, Composite и Leaf) посредством наследования и компоновки, причем таким способом, который позволяет осуществлять поддержку файловых систем произвольной сложности и размеров. Он также дает клиентам возможность единообразно трактовать файлы и каталоги (и другие объекты файловой системы).

Как уже отмечалось, ключ к единообразию находится в общем интерфейсе объектов файловой системы. К настоящему моменту в нашем проекте имеется три класса объектов: Node, File и Directory. Мы пришли к выводу, что операции, имеющие очевидный смысл для файлов и каталогов, необходимо декларировать в базовом классе Node. К этой категории относятся операции извлечения и задания имени узла, а также степени его защищенности. В общий интерфейс необходимо включить и операцию доступа к дочерним узлам (getChild), несмотря на то, что на первый взгляд эта операция не применима к объектам типа File. Теперь рассмотрим другие операции, включение которых в общий интерфейс еще менее очевидно.

Прежде всего, нужно ответить на вопрос: откуда берутся дочерние объекты? Ведь до того, как объект-каталог сможет перечислить свои дочерние объекты, они должны в нем откуда-то появиться. Откуда же?

Каталог не отвечает за создание своих дочерних объектов — это прерогатива пользователя файловой системы. Разумно предположить, что клиенты файловой системы создают файлы и каталоги, а затем помещают их туда, куда хотят. Это означает, в частности, что объекты Directory будут *принимать* дочерние объекты, а не создавать

их. Следовательно, объектам Directory нужен интерфейс, позволяющий прикреплять дочерние записи. Он может выглядеть примерно так:

```
virtual  void adopt (Node*   child) ;
```

Когда клиент вызывает операцию adopt некоторого каталога, он явно передает ответственность за указанный дочерний объект этому каталогу. В данном случае ответственность означает владение: при удалении каталога дочерний объект также удаляется. Именно в этом состоит сущность связи агрегирования между классами Directory и Node (она обозначена ромбом на рис. 2.1).

Если клиент может заставить каталог принять ответственность за дочерний объект, то нужно предусмотреть возможность *освободить* его от этой ответственности.

```
virtual  void orphan (Node*   child);
```

В данном случае "orphan" (открепить) не означает, что родительский каталог прекращает свое существование (удаляется). Он просто перестает быть родителем данного дочернего объекта. Дочерний объект также продолжает существовать; впоследствии он может быть прикреплен к другому узлу или удален.

Как все это отражается на единообразии? Почему нельзя определить данные операции непосредственно в классе Directory?

Предположим, что мы так и сделали. А теперь рассмотрим, как клиент реализует операции, изменяющие структуру файловой системы. Пример такого клиента— команда пользовательского уровня "создать новый каталог". Пользовательский интерфейс для этой команды не имеет значения; предположим, что это просто командная строка, например, команда mkdir системы UNIX. Аргументом этой команды является имя создаваемого каталога, например: mkdir newsubdir

В общем случае пользователь может указать любой допустимый путь к данному имени: mkdir subdirA/subdirB/newsubdir

Команда нормально выполняется, если subdirA и subdirB существуют и являются каталогами (а не файлами). Более формально, subdirA и subdirB должны быть экземплярами подклассов Node, способных иметь дочерние объекты. Если это не так, пользователь должен получить сообщение об ошибке.

Как реализовать mkdir? Предположим, что у нас есть возможность получить ссылку на объект Directory, соответствующий выбранному пользователем текущему каталогу[1]. Чтобы добавить к текущему каталогу новый подкаталог, создается новый экземпляр объекта Directory, а затем вызывается операция adopt текущего каталога, параметром которой является новый каталог:

```
Directory*  current;
current->adopt(new  Directory("newsubdir"));
```

Все просто. Но что будет в общем случае, когда процедуре mkdir задается нетривиальный путь?

Тогда mkdir должна выполнить следующие действия.

---

[1] *Например, с помощью статической операции класса Node. Организация доступа к подобным ресурсам, задача шаблона проектирования SINGLETON. Его применение будет рассмотрено позднее.*

1. Найти объект subdirA (и сообщить об ошибке, если такого объекта не существует).
2. Найти объект subdirB (и сообщить об ошибке, если такого объекта не существует).
3. Прикрепить объект newsubdir к объекту subdirB.

Для выполнения пунктов 1 и 2 необходимо осуществлять итерации по дочерним каталогам текущего каталога и дочерним каталогам subdirA (если он существует) в поиске узла, представляющего subdirB.

В основе реализации mkdir должна лежать рекурсивная функция, аргументом которой является заданный путь:

```
void Client::mkdir (
    Directory* current, const string& path
) {
    string subpath = subpath(path);
    if (subpath.empty()) {
        current->adopt(new Directory (path));
    } else {
        string name = head(path);
        Node* child = findfname(current);
        if (child) {
        mkdir(child, subpath);
        } else {
          cerr <<name << "nonexistent. "<<endl;
        }
    }
}
```

Здесь head и subpath— это стандартные процедуры работы со строками, head возвращает первое имя пути, а subpath — оставшуюся часть.

Операция find находит в каталоге дочерний объект с заданным именем:

```
Node* Client::find (
    const string& name, Directory* current
) {
    Node* child = 0;
    for (int i = 0; child = current->getChild(i);++i {
        if (name == child->getName () ) {
            return child;
        }

    }
    return 0;
}
```

Отметим, что операция find должна возвращать объект типа Node*, поскольку именно его возвращает операция getChild, и это совершенно разумно, ведь дочерний объект может оказаться как каталогом, так и файлом. Но из-за этой маленькой детали Client: :mkdir не будет компилироваться.

Посмотрим еще раз на рекурсивный вызов mkdir. Ему передается некий объект Node*, а не Directory*, как следовало бы. Дело в том, что при спуске по иерархии невозможно сказать, является ли дочерний объект файлом или каталогом. Если для клиента это различие не имеет значения, то все хорошо. Но в данном случае, похоже, различие существенно, поскольку только для объектов Directory определен интерфейс для прикрепления и открепления дочерних объектов.

Однако так ли это на самом деле? Действительно ли клиента (оператор

mkdir) должно заботить различие между файлами и каталогами? На самом деле нет. Его задача — создать новый каталог или сообщить пользователю о неудаче. Поэтому давайте предположим (только предположим!), что adopt и orphan трактуются одинаковым образом для всех классов Node.

Читатель может подумать: "Эти операции не имеют никакого смысла применительно к листовым компонентам типа файлов." Но на чем основано это мнение? Можно определить листовые компоненты и другого типа, например, типа мусорного ящика (или, точнее, повторно используемого бункера), который уничтожает все, что принимает. Прикрепление к листовому компоненту может означать "генерировать сообщение об ошибке". Так что вряд ли следует считать, что операция adopt не может иметь смысла для листовых компонентов. То же самое относится и к операции orphan.

Может быть, не следовало выделять классы File и Directory, пусть бы *все объекты* считались объектами типа Directory? Но реализационные аспекты свидетельствуют об обратном. Объекты каталогов должны иметь определенные свойства, в которых большинство файлов не нуждается: структуры данных для хранения дочерних объектов; кэшированная информация о дочерних объектах, позволяющая повысить производительность и т.д. Опыт показывает, что во многих приложениях листовых объектов значительно больше, чем внутренних узлов. Это одна из причин, по которой шаблон COMPOSITE предписывает рассматривать классы Leaf и Composite отдельно.

Посмотрим, что получится, если определить операции adopt и orphan для всех объектов Node, а не только для класса Directory. По умолчанию эти операции будут; генерировать сообщения об ошибках:

```
virtual void Node::adopt (Node*) {
    cerr << getName() << " is not a directory." << endl;
}
virtual void Node::orphan (Node* child) {
    cerr << child->getName () << " not found." <<endl;
}
```

Это не лучшее решение, но идея понятна: операции могут генерировать исключительные ситуации или ничего не делать — возможностей много. В любом случае, теперь метод Client::mkdir работает удовлетворительно[2]. Заметим, что при таком подходе не нужно вносить изменения в класс File, необходимо только изменить метод Client: mkdir, сделав его параметром Node*, а не Directory*:

```
void Client::mkdir (Node* current, const string& path) {
// …
}
```

Несмотря на то, что операции adopt и orphan на первый взгляд не относятся к тем, которые следует трактовать единообразно, такой подход приносит ощутимые преимущества, по крайней мере в данном приложении. Альтернативный подход может состоять во введении механизма приведения типов, который позволит клиенту определять тип узла:

---

[2] *Точнее,* почти *удовлетворительно. В этом примере игнорируются вопросы управления памятью. В частности, возможна утечка памяти при вызове операции adopt листового объекта, так как клиент передает владение узлу, который не может быть владельцем. Проблема возможной утечки памяти, связанной с применением adopt, носит достаточно общий характер, поскольку данная операция может закончиться неудачей даже для объектов Directory (например, в том случае, когда клиент не имеет необходимых полномочий). Проблема снимается, если объекты Node снабжены счетчиками ссылок, а операция adopt уменьшает (либо неувеличивает) значение счетчика при неудаче.*

```
void Client: mkdir   (
     Directory* current, const string& path
) {
     string subpath = subpath(path);
     if (subpath.empty()) {
          current->adopt(new Directory(path));
     } else {
          string name = head(path);
          Node* node = find(name, current);
          if (node) {
               Directory* child =
                    dynamic_cast<Directory*> (node) ; if(child) {
                    mkdir(child, subpath);
               } else {
                    cerr << getName()
                   <<"is not a directory."
                   <<endl;
              }
          } else {
          cerr << name << "nonexistent." <<endl;
          }
    }
}
```

Обратите внимание, что в связи с введением dynamic_cast возникла дополнительная ветвь программы, которая обрабатывает ситуацию**,** когда пользователь где-либо в path указал неправильное имя каталога. Этот пример демонстрирует, что отсутствие единообразия приводит к усложнению клиента.

Не следует думать, что веских причин для различного представления не бывает. В некоторых приложениях требуется, чтобы компилятор мог выявить любую попытку вызвать операцию с дочерними объектами для листового узла. В таких случаях операции adopt, orphan и др. не могут декларироваться в базовом классе. Однако в тех ситуациях, когда нет оснований опасаться, что единообразие приведет к неприятным последствиям, оно обычно дает ощутимые преимущества в простоте и расширяемости (в чем мы сможем убедиться далее).

Заменители

Рассмотрим введение новой характеристики — а именно символических связей (которые у Мака Файндера (Mac Finder) называются "псевдонимами", а в Windows 95 — "ярлыками"). *Символическая связь*— это ссылка на другой узел файловой системы, т.е. "заменитель" этого узла, а не собственно узел. Удаление символической связи не влияет на узел, на который она ссылается.

Существуют отдельные права доступа к символическим связям, которые могут отличаться от прав по отношению к узлам. Однако в большинстве случаев символические связи ведут себя практически так же, как и сами узлы. Если связь ссылается на файл, то клиент может трактовать ее как файл. Клиент может редактировать файл и, возможно, даже сохранять его с помощью данной связи. Если же связь ссылается на каталог, клиент может добавлять и удалять узлы из каталога, выполняя операции с данной связью, заменяющей каталог.

Символические связи позволяют получать доступ к различным файлам и каталогам, не перемещая и не копируя их, что очень важно, когда узлы

должны находиться в одном месте, а использоваться в другом. Наш проект много потеряет, если он не будет поддерживать символические связи.

Возникает вопрос: "Существует ли шаблон, который поможет спроектировать и реализовать символические связи?" (или более обобщенно: "Как найти подходящий шаблон проектирования для имеющейся задачи?")**.**

В разделе 1.7 книги *Design Patterns* предлагаются следующие шесть подходов:

1. Рассмотреть, как задачи проектирования решаются при помощи шаблонов.
   (Иными словами, изучить раздел 1.6. Ясно, что вряд ли кто-то будет этим зани
   маться, когда процесс разработки в разгаре.)
2. Просмотреть разделы "Назначение" тех шаблонов, которые имеют обнадежи
   вающее название. (Совсем грубое решение.)
3. Изучить взаимосвязи шаблонов. (Слишком сложно для нас на данном этапе, но
   уже ближе к цели.)
4. Найти шаблоны, цель которых (порождающая, структурная, поведенческая)
   соответствует стоящей перед разработчиком задаче. (Добавление символиче
   ских связей в файловую систем) — это структурная задача.)
5. Проанализировать причину перепроектирования (в книге *Design Patterns* при
   водится список таких причин) и применить те шаблоны, которые помогут ее
   избежать. (О перепроектировании в данном случае говорить рано, поскольку
   еще не закончено проектирование.)
6. Выяснить, что в нашем проекте должно быть переменным. Для каждого шаблона проектирования в табл. 1.2 *(Design Patterns,* с. 30) перечислены те аспекты, которые шаблон позволяет изменять.

Воспользуемся последней рекомендацией. Структурные шаблоны (см. табл. 1.2) позволяют изменять:

- ADAPTER — интерфейс объекта;
- BRIDGE — реализацию объекта;
- COMPOS I TE — структуру и состав объекта;
- DECORATOR — обязанности объекта без порождения подкласса;
- FACADE — интерфейс подсистемы;
- FLYWEIGHT — накладные расходы на хранение объектов;
- PROXY — способ доступа к объекту и/или его местоположение.

Попробуем применить PROXY (заменитель). Его назначение формулируется так:

> "Предоставляет заменитель другого объекта для управления доступом к нему."

В разделе "Мотивация" описано применение данного шаблона для решения проблемы задержки загрузки изображений (которая возникает, например, в Web-броузере).

Однако из раздела "Применимость" *следует,* что этот шаблон нам подходит. В данном разделе говорится, что шаблон PROXY можно применять во всех случаях, когда

необходима более гибкая или сложная ссылка на объект, чем простой указатель. Далее описывается несколько наиболее распространенных случаев применения данного шаблона, в том числе "защищающий заменитель", который управляет доступом к другому объекту — именно то, что нам нужно.

Попробуем применить PROXY в нашем проекте файловой системы. Рассмотрим структурную диаграмму данного шаблона (рис. 2.2). На ней представлено три основных класса: абстрактный класс Subject и конкретные подклассы RealSubject и Proxy. Из этого можно заключить, что Subject, RealSubject и Proxy имеют совместимые интерфейсы. Подкласс Proxy содержит также ссылку на RealSubject. В разделе шаблона "Участники" поясняется, что класс Proxy предлагает интерфейс, аналогичный интерфейсу Subject, что позволяет объекту Proxy заменять любой объект Subject. А RealSubject — это конкретный тип объекта, который представляет данный заменитель.

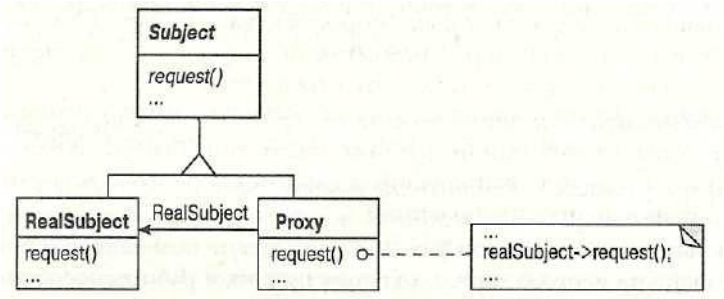


РИС. 2.2. *Структура шаблона PROXY*

Сопоставив эти связи с классами нашей файловой системы, нетрудно понять, что общий интерфейс необходимо привязать к классу Node (так советует поступить и шаблон COMPOSITE). Следовательно, класс Node будет выполнять роль класса Subject.

Теперь нужно определить подкласс Node, соответствующий классу Proxy шаблона. Назовем его Link:

```
class Link : public Node   {
public:
    Link(Node*);
    //здесь повторно объявляется общий интерфейс Node
private:
    Node* _subject;
};
```

Член subject обеспечивает ссылку на реальный субъект. Однако мы, похоже, несколько отклонились от структурной диаграммы, согласно которой ссылка должна быть типа RealSubject. В данном случае этому соответствует ссылка типа File или Directory, но нам необходимо, чтобы символические связи работали для всех видов Node. Как же быть? Обратимся к описанию участников шаблона PROXY:

> "Proxy поддерживает ссылку, которая позволяет ему получать доступ к реальному субъекту. Он может ссылаться на Subject, если RealSubject и Subj ect имеют одинаковый интерфейс."

В нашем случае это действительно так: File и Directory имеют такой же интерфейс, как и Node. Следовательно, _subject может указывать на Node. Если бы интерфейс не был общим, оказалось бы намного сложнее определить единый класс символических связей для файлов и каталогов. Скорее всего, пришлось бы определять два класса символических связей с идентичными обязанностями, один — для файлов, а второй — для каталогов.

Осталось рассмотреть, как в классе Link реализуется интерфейс Node. В первом приближении каждая операция просто сводится к соответствующей операции _subject, например, следующим образом:

```
Node* Link::getChild (ind n) {
    return _subject->getChild(n);
}
```

В некоторых случаях объект Link может демонстрировать поведение, независимое от своего субъекта. Например, класс Link может определять операции своей защиты, которые реализуются в нем точно так же, как и в классе File.

Лорион Барчел (Laurion Burchall) сделал несколько интересных наблюдений по поводу применения шаблона PROXY в данном приложении [Burchall95]:

> "Если файл удаляется, заменители, указывающие на него, будут иметь висящие указатели. Чтобы уведомить все заменители об удалении файла, можно использовать шаблон OBSERVER, но это не позволит нам поместить новый файл на место старого, сохранив при этом работоспособность символических связей.
> В системах UNIX и Mac символическая связь хранит только имя файла, на который она ссылается. Заменитель может содержать имя файла и ссылку на корневой каталог файловой системы. Однако тогда доступ к файлу посредством заменителя может стать весьма дорогостоящим, так как придется каждый раз выполнять поиск имени."

Все правильно, за исключением высказывания относительно OBSERVER. Ничто не мешает уведомить заменитель и вновь осуществить его привязку при замене файла, на который он указывает. В этом смысле замена не отличается от удаления.

В остальном замечание Лорина справедливо: хранение указателя на субъект хотя и эффективно, но неудовлетворительно без дополнительного механизма. Замена субъекта без аннулирования ссылок на него требует такого уровня косвенности, которого у нас в настоящий момент нет. Хранение имени файла вместо указателя может решить проблему, но потребуется организовать ассоциативное хранение, чтобы эффективно отображать имена в объекты. Даже в этом случае произойдет возрастание накладных расходов по сравнению с хранением указателей; с этим можно смириться, пока количество файлов или уровней ссылок не слишком велико. Кроме того, ассоциативное хранилище также необходимо обновлять при удалении или замене файлов.

Если доступ к файлу посредством ссылки — более часто выполняемая операция, чем замена или удаление файла (как правило, так оно и есть), то основанный на применении шаблона OBSERVER подход предпочтительнее, чем поиск имен.

По мере развития проекта возникает тенденция трактовать базовый класс как универсальную свалку: его интерфейс постоянно растет, со временем к нему добавляются все новые и новые операции. С каждой новой характеристикой файловой системы к интерфейсу добавляется одна или две операции. Сегодня он поддерживает расширяемые атрибуты, завтра вычисляет новый вид статистики, затем добавляется операция, которая возвращает пиктограмму графического интерфейса пользователя. Вскоре класс Node разрастается до таких размеров, что его сложно понимать, поддерживать и делить на подклассы.

Эту проблему мы рассмотрим ниже и найдем такой способ добавления новых опе раций, при котором не требуется модифицировать существующие классы.

Правила посещения объектов

К настоящему моменту мы применили два шаблона проектирования: COMPOSITE использовался для определения структуры файловой системы, а PROXY — для организации поддержки символических связей. В результате получилась иерархия классов, представленная на рис. 2.3.

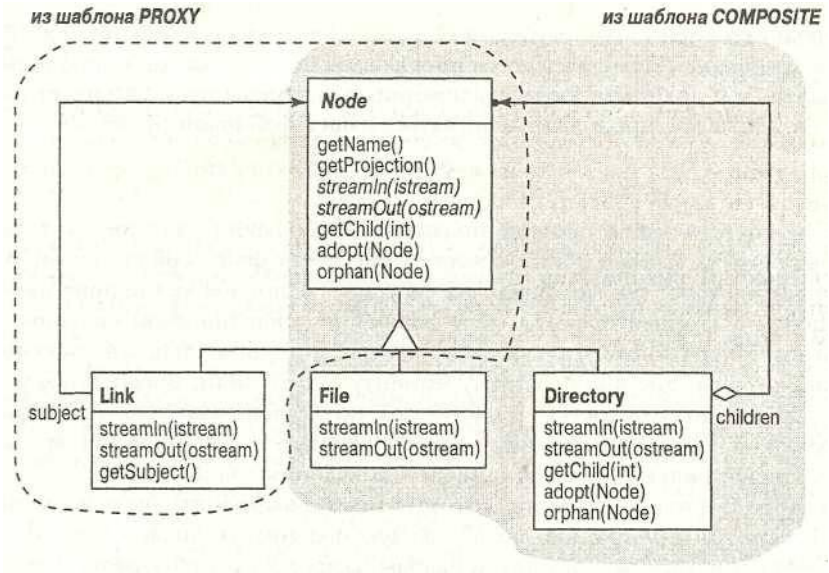


РИС. 2.3. *Структура классов после применения шаблонов COMPOSITE и PROXY*

Операторы getName и getProtection возвращают соответствующие атрибуты узла. Базовый класс Node определяет реализацию этих операторов по умолчанию. Операции streamIn и streamOut предназначены для ввода и вывода содержимого узлов файловой системы. (Предполагается, что файлы моделируются в виде простых потоков байтов, как в системе UNIX.) streamIn и streamOut — *абстрактные* операции, это означает, что базовый класс декларирует, но не обязательно реализует их. Поэтому названия этих операций на рисунке набраны курсивом. Для операций getChild, adopt и orphan задана реализация по умолчанию, чтобы несколько упростить определение листовых компонентов.

Классы Node, File и Directory получены в результате применения шаблона COMPOSITE. Использование шаблона PROXY привело к созданию класса Link, также предписывалось создать класс Node, но он к этому моменту уже существовал. В результате класс Node сочетает в себе свойства базовых классов этих двух шаблонов: он представляет класс Component в шаблоне COMPOSITE и класс Subject в шаблоне PROXY. Такое "двойное гражданство" является признаком того, что Александер (Alexander) называет "плотной" композицией шаблонов, при которой два или более шаблонов занимают одно и то же "место" в классах системы.

Плотность имеет свои плюсы и минусы. Когда несколько шаблонов размещаются в относительно небольшом количестве классов, это дает проекту определенную глубину; в меньшем пространстве заключено больше смысла, как в хорошей поэзии. С другой стороны, такая плотность может быть напоминанием о не столь удачных попытках.

Ричард Гебриэл (Richard Gabriel) в этой связи заметил [Gabriel95]:

> "Определенной Александером (Alexander) плотности в программном обеспечении соответствует, по крайней мере частично, код, в котором каждая часть выполняет несколько задач. Такому коду требуется в два-три раза больше места в оперативной памяти; он напоминает код, который мы писали на ассемблере в 60-е — 70-е годы."

Правильно замечено, что "глубокий код— не обязательно хороший код". В действительности замечание Ричарда касается проявлений более серьезной проблемы: шаблон может оказаться утерянным после реализации. Это замечание заслуживает отдельного обсуждения, а пока что продолжим заниматься нашей файловой системой.

В любой операционной системе подавляющее большинство команд пользовательского уровня определенным образом взаимодействует с файловой системой. В этом нет ничего удивительного, так как файловая система— основное хранилище информации, содержащейся в компьютере. По мере эволюции операционной системы этот центральный компонент должен предоставлять новые функциональные возможности.

Классы, которые мы к настоящем)' моменту определили, обеспечивают минимум функциональных возможностей. В частности, интерфейс класса Node содержит только несколько основных операций, поддерживаемых всеми подклассами Node. Эти операции предоставляют доступ к данным и поведению узлов.

Существуют и другие операции, которые можно выполнять по отношению к этим классам. Рассмотрим операцию, позволяющую подсчитать число слов в файле. Если добавить операцию getWordCount в базовый класс Node, то в результате придется модифицировать как минимум класс File, а вероятно— и все остальные классы. Во избежание модификации существующего кода (и внесения ошибок в него) поищем другие решения. В базовом классе есть операции работы с потоками, которые клиент файловой системы может использовать для исследования текста файла. Нет необходимости изменять существующий код, так как клиенты могут реализовать подсчет слов с помощью имеющихся операций.

Фактически, главной задачей при проектировании интерфейса Node является обеспечение минимального набора операций, которые позволят клиентам создавать новые функциональные возможности. В противном случае придется при добавлении каждой новой возможности изменять класс Node и его подклассы, что требует немалых усилий

и повышает вероятность возникновения, ошибок. Кроме того, интерфейс Node превратится в набор разнородных операций, и это неизбежно будет затенять основные свойства объектов Node, классы будет сложно понимать, расширять и использовать. Поэтому необходимо сконцентрировать усилия на создании достаточного набора стандартных процедур, чтобы определить простой, однородный интерфейс класса Node.

Но как быть с операциями, которые должны по-разному работать в узлах различных типов, как сделать их внешними по отношению к подклассам Node? Рассмотрим в качестве примера оператор cat системы UNIX, который просто печатает содержимое файла в стандартном выводе, а при попытке применить его к каталогу сообщает, что данный узел нельзя распечатать (возможно потому, что текстовое представление каталога выглядит не слишком красиво).

Поскольку поведение оператора cat зависит от типа узла, необходимо определить операцию базового класса, которую классы File и Directory реализуют по-разному. Таким образом, нам придется менять существующие классы.

Есть ли альтернатива? Предположим, что мы уберем эти функциональные возможности из классов Node и поместим их в клиент. Тогда придется ввести некий механизм нисходящего приведения типов, чтобы клиент мог определить, с каким узлом он работает:

```
void Client::cat (Node* node) {
    Link* l;
    if (dynamic_cast<File*> {node) ) {
        node->streamOut (cout); //выводит содержимое
    }else if (dynamic_cast<Directory*>(node)) {
        cerr<< "Can't cat a directory. "<< endl;
    }else if {l = dynamic_cast<Link*>(node)) {
        cat(l->getSubject()); //печатает субъект ссылки
}
```

Вновь кажется, что приведения типов избежать невозможно. И снова это приводит к усложнению клиента. Мы сознательно шли па его усложнение, когда решили поместить функциональные возможности в клиент, а не в классы Node. Но помимо самих функциональных возможностей добавились проверки типа и условные разветвления, в результате быстрота выполнения метода существенно снизилась.

Если размещение функциональных возможностей непосредственно в узлах вызывает неудобства, то применение проверок типа— немыслимо! Однако не будем спешить помещать операцию cat () в класс Node и его подклассы во избежание этих проверок, а рассмотрим шаблон проектирования VISITOR (посетитель), который предлагает третью альтернативу. Его назначение выглядит следующим образом.

> "Шаблон представляет операцию, которую нужно выполнить с объектами некой структуры. Шаблон VISITOR позволяет определить новую операцию, не внося изменения в классы, элементы которых затрагивает данная операция."

В разделе "Мотивация" рассматривается компилятор, который представляет программы в виде абстрактных синтаксических деревьев. Проблема состоит в обеспечении поддержки открытого множества аналитических действий, таких как проверка типов**,** красивая печать и генерирование кода без изменения классов, реализующих абстрактные синтаксические деревья. Проблема компилятора аналогична нашей, за

исключением того, что мы оперируем структурами файловой системы, а не абстрактными синтаксическими деревьями и хотим производить с нашими структурами другие операции. Однако конкретные операции не имеют особого значения. Важно отделить их от операций класса Node, не прибегая к приведению типов и дополнительному ветвлению программы.

В шаблоне VISITOR это достигается путем добавления единственной операции к участнику под названием "элемент" (Element), которому в данном случае соответствует класс Node.

```
virtual  void  accept(Visitors]  =  0;
```

Операция accept позволяет объекту "Visitor" посещать указанный узел. Объект-посетитель инкапсулирует операции, выполняемые с данным узлом. Все конкретные подклассы класса Element реализуют accept столь же просто.

```
void File::accept (Visitor& v) {v.visit(this);}
void Directory::accept (Visitor& v) {v.visit(this);}
void Link::accept (Visitor& v)      {v.visit(this);}
```

Все эти реализации выглядят одинаково, но в действительности они, конечно же, отличаются, — тип this в каждом случае различен. Реализация шаблона предлагает примерно такой интерфейс класса Visitor:

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

     void visit(File*);
     void visit(Directory*);
     void visit(Link*);
};
```

Интересное свойство шаблона заключается в том, что когда операция accept в узле вызывает операцию visit объекта Visitor, узел фактически сообщает свой тип этому объекту. В свою очередь вызванная операция объекта Visitor может делать все, что нужно, с узлом этого типа:

```
void Visitor::visit (File* f) {
     f->streamOut(cout);
}
void visitor::visit (Directory* d) {
     cerr<<"Can't cat a directory. "<<endl;
}
void Visitor::visit (Link* I) {
     l->getSubject()->accept(*this);
}
```

Последняя операция требует пояснений. Она вызывает операцию getSubject (), возвращающую узел, на который указывает данная связь, т.е. субъект данной связи[3]. Просто вывести субъект в выходной поток нельзя, поскольку он может оказаться каталогом, а не файлом. Поэтому мы вновь подключаем посетителя, как и в случае с

---

[3] *Операция getSubject () - собственная операция класса Link, она декларируется и реализуется только для данного класса, поэтому невозможно подучить доступ к этой операции, трактуя связи как узлы. При. использовании Visitor данная проблема снимается, так как посетитель фактически восстанавливает информацию о типеузла во время его посещения.*

самой связью. Посетитель действует согласно типу субъекта— он следует вдоль связей, пока не дойдет до файла или каталога, где, наконец, может сделать нечто полезное.

Теперь, чтобы распечатать любой узел, создается соответствующий объект-посетитель и узлу дается указание принять его:

```
Visitor  cat; node->accept(cat) ;
```

Обращение узла к посетителю разрешается в операцию visit, которая соответствует типу данного узла (File, Directory или Link), тем самым обеспечивается правильный ответ. Из этого следует, что Visitor позволяет упаковать функциональные возможности типа команды cat в единый класс**,** не прибегая к проверкам типов.

Инкапсуляция в объекте Visitor операции cat— вещь хорошая, но как быть с другими командами, неужели придется изменять существующий код, если понадобится сделать с файлом что-нибудь отличное от cat? Пусть требуется реализовать команду, которая перечисляет имена дочерних узлов определенного каталога, как это делает команда Is системы UNIX. Выводимое имя должно сопровождаться знаком /, если данный узел является каталогом, и знаком *@*, если это символическая связь.

Необходимо предоставить другому классу-посетителю "право посещения" объектов Node, но не хотелось бы добавлять еще одну операцию accept в базовый класс Node. Оказывается, что это и не нужно: любой объект Node может принимать объект-посетитель любого вида, просто к данному моменту у нас был только один вид посетителя. Б действительности же в шаблоне VISITOR класс Visitor — это абстрактный класс.

```
class Visitor   {
public:
     virtual ~Visitor() {}

virtual void Visit(File*) = 0;
virtual void Visit(Directory*) = 0;
virtual void Visit(Link*) = 0;
protected:
     Visitor();
Visitor(const  Visitor&);
};
```

Для каждой новой возможности создается подкласс Visitor, в котором реализуются операции visit для всех типов посещаемых узлов. Например, подкласс CatVisitor реализует описанные выше операции. Можно также определить подкласс SuffixPrinterVisitor, который печатает имена узлов с соответствующими знаками.

```
class  SuffixPrinterVisitor  :  public  Visitor   {
public:
     SuffixPrinterVisitor()   {}
     virtual   ~SuffixPrinterVisitor()   {}

     virtual   void  visit(File*)   {}
     virtual   void  visit (Directory* )   { cout   <<"/" ; }
     virtual   void  visit(Link*)   {cout   <<"@";}
};
```

SuffixPrinterVisitor можно использовать в клиенте, реализующем команду ls:

```
void  Client::ls   (Node*   n)   {
     SuffixPrinterVisitor suffixPrinter; Node*
     child;

     for (int i = 0; child = n->getChild(i); ++i) {
```

```
cout <<child->getName();
child->accept(suffixPrinter) ;
cout <<endl;
```

После того как в классы Nodes была добавлена операция accept (Visitor&) и таким образом были установлены правила посещения, нет необходимости дополнительно модифицировать эти классы при появлении новых подклассов Visitor.

Мы использовали перегрузку имени функции, чтобы дать операциям посетителя одно и то же имя. Альтернативный подход может заключаться в кодировании типа узла в имени операции visit:

```
class Visitor {

public:
    virtual ~Visitor() {}
    virtual void , visitFile(File*) = 0
    virtual void visitDirectory(Directory*) = 0
    virtual void visitLink(Link*) = 0

protected:
  Visitor();
     Visitor(const Visitors);
};
```

Тогда вызовы операций принимают более понятную форму:

```
void File::accept (Visitor& v)  {v.visitFile (this) ;  }

void Directory::accept(Visitor& v)  {
   v.visitDirectory(this) ; } void Link::accept(Visitor& v)
{v.visitLink(this);  }
```

Более существенное преимущество данного подхода проявляется в случае, когда имеется определенное поведение по умолчанию или в подклассах замещается только небольшая часть имеющихся операций. При перегрузке подклассы должны замещать *все* функции; в противном случае C++-компилятор может решить, что ваши выборочные замещения направлены на сокрытие одной или нескольких операций базового класса. Мы обошли эту проблему, когда дали операциям посетителя различные имена. Теперь подклассы могут переопределять любое подмножество операций.

Операции базового класса реализуют поведение по умолчанию для всех типов узлов. Когда поведение по умолчанию одинаково для двух или нескольких типов, можно поместить эти общие функциональные возможности во "всеулавливающую" операцию visitNode (Node*), которую другие операции будут вызывать по умолчанию:

```
void Visitor::visitNode  (Node* n)  {
    //общее поведение по умолчанию
}
void Visitor::visitFile  (File*  f)  {
    Visitor::visitNode(f);
}
void Visitor::visitDirectory  (Directory* d)  {
    Visitor::visitNode(d);
}
void Visitor::visitLink  (Link*  1)  {
    Visitor::visitNode(1);

    }
```

Проблемы, связанные с применением шаблона VISITOR

Прежде чем применять шаблон VISITOR, необходимо рассмотреть ряд вопросов.

Во-первых, следует ответить на вопрос, стабильна ли иерархия посещаемых классов, т.е. будут ли постоянно определяться новые подклассы Node или это достаточно редкое явление. При определении нового типа узлов понадобится вносить изменения во все классы иерархии посетителей, чтобы добавить соответствующую операцию visit.

Если ни один из существующих посетителей никак не связан с новым подклассом и определен некий эквивалент операции visitNode, обеспечивающий определенное поведение по умолчанию, то проблем не возникнет. Но если хотя бы один класс существующих посетителей должен работать с новым подклассом, то придется изменять, как минимум, сам этот класс и базовый класс Visitor. В таком случае множественные изменения неизбежны. Если же отказаться от использования шаблона VISITOR и попытаться разместить новые функциональные возможности в иерархии Node, это, скорее всего, также приведет к изменениям в иерархии.

Во-вторых, необходимо понимать, что шаблон VISITOR создает циклическую зависимость между иерархиями классов Node и Visitor. Следовательно, изменение интерфейса любого из базовых классов приведет к перекомпиляции обеих иерархий. Конечно, это не хуже, чем размещать функции в едином базовом классе Node, но хотелось бы в принципе избежать таких зависимостей.

Ниже приводится актуальное замечание Келвина Хеннея [Henney96]:

> "Перегрузка в C++ не требует обязательной перегрузки всех версий visit, также не нужно отказываться от перегрузки конкретного члена visit. Наряду с поддержкой концепций пространства имен, декларация using позволяет вводить имена из базового класса в текущий класс для перегрузки:
>
> ```
> class NewVisitor : public Visitor {
> public:
>     using Visitor::visit;  //вводит все функции
>                            //visit для перегрузки
>     virtual void visit(Node*);  //замещает вариант Node*
> };
> ```
>
> Перегрузка обеспечивает определенный порядок. Она достаточно удобна, так как пользователям не нужно помнить, какие имена или соглашения использовать для данной функции [visit], и позволяет новой версии объекта Visitor вобрать в себя изменения, не затрагивая код клиента."

Мы рассмотрели шаблоны COMPOSITE и PROXY, с помощью которых удалось определить структуру файловой системы, и шаблон VISITOR, позволяющий безболезненно вводить новые возможности, *добавляя* код, а не *изменяя* его. В этом состоит еще одна черта хорошего объектно-ориентированного проекта: если систему можно модифицировать, не затрагивая существующий код, это повышает ее гибкость и возможности сопровождения.

Однако вернемся к нашей файловой системе. Еще один важный вопрос касается обеспечения безопасности. В этой связи нужно рассмотреть по меньшей мере два аспекта.

1. Защита системы от случайных и злонамеренных повреждений.
2. Поддержка целостности системы при сбоях аппаратного и программного обеспечения.

Остановимся подробно на первом из них, а второй аспект читателям предлагается рассмотреть самостоятельно в качестве упражнения.

Защита в однопользовательской системе

Каждый, кто активно пользуется компьютером, может поведать ужасную историю о потере жизненно важных данных вследствие случайной синтаксической ошибки, щелчка мыши и т.д. Типичные проблемы — ошибочное удаление или случайное изменение файла. Современная файловая система должна предоставлять возможность отменить несколько последних действий, чтобы обеспечить в таких случаях восстановление данных, однако всегда лучше предотвратить нежелательные действия, чем потом бороться с их последствиями. Тем более, что большая часть файловых систем предлагает другой выбор: предотвращение ошибки или сожаление о случившемся.

Рассмотрим защиту объектов (узлов) файловой системы от удаления и модификации. Речь пойдет о защите, связанной с программным интерфейсом, а не с интерфейсом пользователя. Это различие не должно нас беспокоить, так как наши программные абстракции вполне соответствуют абстракциям пользовательского уровня. Кроме того, предполагается, что файловая система однопользовательская, т.е. как в обычном, не соединенном с сетью персональном компьютере (а не многопользовательская, типа системы UNIX). Это предположение позволит упростить рассуждения. Защита в многопользовательской системе рассматривается позже.

Все элементы файловой системы (файлы, каталоги и символические связи) придерживаются Node-интерфейса, который к настоящему моменту включает в себя следующие операции:[4]

```
const string& getName();
const Protection& getProtection();

void setName(const string&);
void setProtection(const Protection&);

void streamIn(instream&);
void streamOut(ostream&);

Node* getChild(int);

void adopt(Node*);
void orphan(Node*);
```

Все операции за исключением get Protection уже обсуждались. Данная операция извлекает информацию о защищенности узла, но что это значит, пока неясно. О какой защите идет речь?

Чтобы защитить узлы от случайного изменения или удаления, нужна защита от записи, т.е. узел может быть перезаписываемым или неперезаписываемым. Чтобы узел был защищен от любопытных глаз, следует иметь возможность сделать его несчитываемым.

---

[4] *Добавлены соответствующие операции set... для getName и getProtection.*

Понятно, что узел будет защищен только от *неосведомленных* любопытствующих, т.е. от тех, кто не знает, как изменять защиту узла. Защита от чтения в однопользовательской системе не так уж существенна, но в многопользовательской системе она играет очень важную роль.

Итак, узлы могут быть считываемыми и несчитываемыми, перезаписываемыми и не-перезаписываемыми. Большинство файловых систем имеет дополнительные режимы защиты, которые регулируют выполнение, автоматическое архивирование и т.п. Разновидности защиты можно трактовать более-менее аналогично считыванию и перезаписи, поэтому ограничимся обсуждением этих двух режимов.

Какое влияние оказывают невозможность считывания и перезаписи на поведение узла? Очевидно, что несчитываемый файл не должен показывать свое содержимое, т.е. он не должен отвечать на запросы streamOut. Кроме того, клиент не должен иметь доступ к дочерним узлам несчитываемого узла, таким образом, по отношению к несчитываемым узлам операция getChild не работает. Что касается запрещения перезаписи, такой узел не должен разрешать изменять ни свои атрибуты, ни свою структуру; поэтому по отношению к нему нужно нейтрализовать операции setName, streamln, adopt, orphan. (К операции setProtecton в данном случае надо отнестись с осторожностью. Более подробно о ней будет сказано при обсуждении защиты в многопользовательской системе.)

Предотвращение удаления неперезаписываемого узла ставит интересные задачи языкового уровня. Например, клиент не имеет права явно удалять такой узел в отличие от других объектов. Сделаем так, что компилятор C++ будет отслеживать подобные попытки, но не путем объявления узла const, как некоторые читатели могли подумать, ведь защита узла может меняться во время выполнения. Вместо этого *защитим деструктор.* В отличие от обычного деструктора защищенный запрещает классам, внешним по отношению к иерархии классов Node, явно удалять узел[5]. Еще одно положительное свойство защиты деструктора— она не допускает образования локальных объектов Node (т.е. узлы создаются в стеке) и препятствует автоматическому удалению неперезаписываемого узла при его неправильном поведении (которое может свидетельствовать об ошибке).

Но как удалить узел, деструктор которого защищен? Очевидно одно: нужно использовать некую операцию, в которой удаляемый узел выступает в качестве параметра. Где определить эту операцию? Есть три возможных решения.

1. Класс Node (с возможным последующим переопределением в подклассах).
2. Класс, не входящий в иерархию Node.
3. Глобальная функция.

От третьего решения следует отказаться сразу, потому что это обычная статическая функция, определенная на существующем классе. Операция удаления, поставляемая извне иерархии Node, также не слишком привлекательна, поскольку требует, чтобы класс, в котором она определена, был дружественным классу Node. Почему? Потому что если узел оказывается перезаписываемым и, следовательно, удаляемым, то кто-то должен вызвать его защищенный деструктор. Единственная возможность

---

[5] *Вариант объявления деструктора private не рассматривается, поскольку это не позволит создавать подклассы.*

осуществить это извне иерархии классов Node — сделать удаляющий класс дружественным Node, но возникает неприятный побочный эффект: видимым становится не только Node-деструктор, но и все содержимое класса Node.

Рассмотрим первый вариант; определим операцию destroy в базовом классе Node. Если данная операция статическая, она должна использовать в качестве параметра некий экземпляр Node; в противном случае она может быть беспараметрической, что подразумевает использование параметра this. Выбор между статической, виртуальной и невиртуальной функциями-членами - это выбор между расширяемостью и эстетикой.

Виртуальная функция-член расширяется подклассами. Но с эстетической точки зрения выражение

```
node->desstroy() ;
```

выглядит слишком мрачно. Тоже можно сказать и о выражении, использующем невиртуальную функцию-член:

```
delete  this;
```

Статическая функция позволяет избежать этой проблемы,

```
Node::destroy(node);
```

но ее нельзя модифицировать в подклассах. Вариант с невиртуальной функцией членом — наихудший с обеих точек зрения.

Попробуем воспользоваться синтаксическими преимуществами статической функции-члена и одновременно добиться возможности расширения в подклассах.

Какие задачи операции destroy не зависят от вариантов ее расширения в псд классах? Инвариантными представляются две задачи: операция destroy должна про верить, является ли переданный ей узел перезаписываемым, и если да — удалить его. Подклассы могут изменить критерии удаления или порядок его осуществления, инварианты останутся инвариантами. Чтобы реализовать эти инварианты способе допускающим последующие расширения, воспользуемся шаблоном TEMPLATE METHOD. В его назначении написано:

> "Шаблон определяет основу алгоритма операции, оставляя некоторые шаги на усмотрение подклассов. Позволяет подклассам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом."

Согласно первому пункту раздела "Применимость" шаблон используется в тех случаях, когда нужно однократно реализовать инвариантные части алгоритма и оставить на усмотрение подклассов реализацию поведения, которое может изменяться. Реализация шаблона в общем случае выглядит следующим образом:

```
void BaseClass::templateMethod() {
    //далее идет инвариантная часть
    doSomething();   //часть, которую подклассы могут менять
    //другая инвариантная часть
    doSomethingElse(); //другая меняющаяся часть
    //и т.д.
}
```

Определенные в базовом классе BaseClass операций doSomething и doSomethingElse реализуют поведение по умолчанию, в подклассах они конкретизируются, чтобы выполнять различные действия. В шаблоне такие операции

называются *элементарными операциями (primitive operations),* поскольку из них фактически составляется операция более высокого порядка.

Элементарные операции следует объявлять виртуальными (virtual), чтобы подклассы имели возможность полиморфным образом переопределять их. Шаблон предлагает явно идентифицировать элементарные операции, предваряя их имена приставкой "do-". Следует также декларировать их как защищенные (protected), чтобы клиенты не могли обращаться к ним непосредственно, поскольку эти элементарные операции могут быть лишены смысла вне контекста данного шаблонного метода.

Что касается самого шаблонного метода, шаблон рекомендует декларировать его как невиртуальный (final в Java), чтобы гарантировать, что инвариантная часть останется инвариантной. В рассматриваемом случае мы пошли еще дальше — наш кандидат на шаблонный метод, операция destroy, не только невиртуальная, но даже статическая. Это не означает, что применить данный шаблон проектирования не удастся, но нам придется повозиться с его реализацией.

Однако прежде чем окончательно сформировать операцию destroy, следует разработать элементарные операции. Как уже отмечалось, инвариантная часть операции состоит в том, чтобы определить, является ли узел перезаписываемым, и, если да, удалить его. Таким образом, можно предложить следующую структуру:

```
void Node::destroy  (Node*  node)   {
     if  (node->isWritable()){
         delete  node;
     } else  {
         cerr  <<  node->getName()  <<" cannot  be  deleted."
              <<endl;

     }
 }
```

isWritable — элементарная операция[6], которую подклассы могут переопределять, чтобы варьировать критерии защиты. Базовый класс может обеспечивать реализацию isWritable по умолчанию или оставлять реализацию этой операции подклассам, объявив ее полностью виртуальной:

```
class  Node  {
public:
     static void destroy(Node*);

protected:
     virtual  ~Node();
     virtual bool  isWritable()=0;
     //...
     };
```

Такое объявление позволяет избежать сохранения связанного с защитой состояния в абстрактном базовом классе, но также препятствует повторному использованию этого состояния в подклассах.

Хотя операция destroy статическая (а не просто невиртуальная), в данном случае она может работать как шаблонный метод: в ссылке на this нет необходимости; операция просто передается соответствующему экземпляру Node. А поскольку операция

---

[6] *При ее именовании мы отошли от правила, рекомендованного шаблоном, но "doIsWritable" — слишком длинно и неэстетично.*

destroy — член базового класса Node, она может вызывать защищенные операции, такие как isWritable и delete экземпляров Node, не нарушая инкапсуляцию.

К настоящему моменту операция destroy использует только одну элементарную операцию, не считая деструктор. Следует добавить другую элементарную операцию, чтобы позволить подклассам варьировать сообщение об ошибке, а не жестко задавать его в базовом классе:

```
void Node::destroy (Node* node) {
    if (node->isWritable()) {
        delete node;
    } else {
        node->doWarning(undeletableWarning);
    }
}
```

Операция doWarning определяет, как узел предупреждает пользователя о *любой* проблеме, а не только о невозможности своего удаления. Ее можно произвольным образом усложнить, заставив выполнять любые действия: от печати строки до генерирования исключительной ситуации. При этом не нужно определять примитивные операции для всех мыслимых ситуаций (doUndeletableWarning, doUnwritableWarning, doThisThatOrTheOtherWarning и т.п.).

Можно применить шаблон TEMPLATE METHOD к другим операциям класса Node, которые не являются статическими. При этом вводятся новые элементарные операции:

```
void Node::streamOut (ostream& out) {
    if (isReadable()) {
        doStreamOut(out);
    } else {
        doWarning(unreadableWarning) ;
    }
}
```

Основное отличие между шаблонными методами streamOut и destroy состоит в том, что streamOut может непосредственно вызывать операции класса Node. Метод destroy не может этого делать, поскольку он статический и не ссылается на this. Следовательно, destroy должен передаваться удаляемому узлу, которому он делегирует примитивные операции. Необходимо также помнить, что метод streamOut становится *невиртуальным,* если его статус повышается до шаблонного метода.

Шаблон TEMPLATE METHOD приводит к инверсии управления, получившей название *"принцип Голливуда"* ("Не звоните нам, мы вам сами позвоним"). Подклассы могут расширять или изменять реализацию различных частей алгоритма, но они не могут изменить поток управления шаблонного метода и другие инвариантные части. Сдедовательно, определяя новый подкласс класса Node, нужно думать не об управляющей логике программы, а об *обязанностях* подкласса— о том, какие операции *необходимо* переопределить, какие *молено* переопределить, а какие *НЕЛЬЗЯ* переопределять. Шаблонные методы делают эти обязанности более явными, что способствует более четкому структурированию операций.

"Принцип Голливуда" представляет особый интерес, поскольку он является ключом к пониманию каркасов. Каркас содержит неизменные архитектурные и реализационные атрефакты, а варьирующиеся части остаются в подклассах конкретных приложений.

Именно из-за инверсии управления некоторым программистам сложно использовать в своей работе каркасы. При процедурном подходе к программированию главное внимание уделяется управляющей логике программы. Не зная обо всех поворотах и разветвлениях, процедурную программу невозможно понять даже при самой совершенной функциональной декомпозиции. Хороший каркас позволяет абстрагироваться от деталей управляющей логики. Основное внимание следует уделить объектам, которые по сравнению с управляющей логикой могут оказаться более или менее материальными. Нужно оперировать такими понятиями, как обязанности и взаимодействия объектов. Это более высокоуровневый и более декларативный взгляд на мир, обладающий большими потенциальными возможностями и гибкостью. Шаблон TEMPLATE METHOD реализует данные преимущества в меньшей степени, чем некий каркас — на операционном уровне, а не на уровне объектов.

Защита в многопользовательской среде

Мы добавили к разрабатываемому проекту файловой системы простую защиту в однопользовательском режиме. Рассмотрим теперь среду, в которой пользователи работают с файловой системой совместно. Возможность работы в многопользовательском режиме — обязательное современное требование независимо от того, идет ли речь о традиционной системе разделения времени с централизованной файловой системой или о более современной сетевой файловой системе. Даже операционные системы персональных компьютеров, которые изначально разрабатывались для однопользовательских сред (такие как OS/2 и Windows NT), сейчас допускают существование нескольких пользователей. В любом случае, поддержка многопользовательского режима добавляет новые аспекты к проблеме защиты.

И вновь предлагается пойти по пути наименьшего сопротивления и моделировать нашу схему многопользовательской защиты, исходя из уже существующей схемы, а именно — используемой в системе UNIX. В системе UNIX с каждым узлом файловой системы связан некий "user" (пользователь). Как правило, пользователь узла— это тот, кто создал данный узел. Таким образом, для некоторого узла множество всех пользователей делится на два лагеря: пользователь-создатель и все остальные. Официальное (и не всегда удачное) название "всех остальных" в системе UNIX— *other* (другой)[7].

Отличие пользователя-создателя от "всех остальных" позволяет создать независимый уровень защиты для каждого из узлов. Например, файл может считывать только его пользователь и никто другой; в таком случае файл является "user-readable" и "other-unreadable". То же самое происходит с возможностью перезаписи и любым другим режимом защиты (выполнимостью, автоархивированием и т.д.).

Пользователи должны иметь *регистрационное имя,* под которым они входят в систему (login name). Это имя уникальным образом идентифицирует пользователя как для системы, так и для других пользователей. Для системы определенный пользователь и его регистрационное имя неразделимы, хотя в действительности одному человеку может быть разрешено иметь несколько таких имен. Важно убедиться, что никто не может связать себя

---

[7] *Читатели, которые знакомы с системой UNIX, сразу отметят, что есть еще третий лагерь — "group " (группа). Мы рассмотрим его немного позднее.*

ни с каким иным именем, кроме своего собственного (предположим, что оно одно). Поэтому при входе в систему UNIX нужно сообщать не только регистрационное имя, но и пароль, удостоверяющий личность входящего. Данный процесс называется *аутентификацией.* В системе UNIX большое внимание уделяется борьбе с маскировкой, поскольку самозванец имеет доступ ко всему, что доступно легитимному пользователю.

Перейдем от общих вопросов к конкретным. Как моделируется пользователь? Для приверженцев объектно-ориентированной методологии ответ очевиден: можно использовать объект. Объект должен относиться к определенному классу, следовательно, нужно определить класс User.

Теперь необходимо подумать об интерфейсе класса User. Нужно решить, что клиент может делать с данным объектом, хотя на данном этапе важнее, что клиент *не может* делать с ним. В частности, *клиенту* не разрешается бесконтрольно создавать объекты User.

Предположим, что между объектами User и регистрационными именами существует взаимно однозначное соответствие. (Они концептуально неразделимы и нет особой необходимости разрешать существование нескольких объектов User с одним и тем же регистрационным именем.) Предположим также, что с объектом User всегда должно быть связано правильное регистрационное имя. Это разумно, поскольку с точки зрения системы пользователь без регистрационного имени не имеет смысла. И последнее важное замечание — клиенту нельзя разрешать создавать экземпляр пользователя, если он не сообщит регистрационное имя и пароль. В противном случае незаконные приложения смогут получать доступ к файлам и каталогам просто путем создания объектов User с соответствующим регистрационным именем.

Само существование конкретного объекта User представляет собой единицу аутентификации, поэтому создание экземпляров объектов User необходимо тщательно контролировать. Попытка создания экземпляра должна отклоняться, если приложение сообщает неверное регистрационное имя или пароль. При этом не должны создаваться неполноценные объекты User, которые не работают из-за того, что созданы без необходимой информации. Все это исключает использование обычных конструкторов C++ *для* создания экземпляров объектов User.

Необходим *безопасный* способ создавать объекты User, не связанный с конструкторами интерфейса клиента (т.е. клиент не должен иметь возможность незаконно создавать экземпляры объекта User) . Попытаемся выразить данное требование с помощью объектно-ориентированных понятий.

Рассмотрим три элементарных объектно-ориентированных понятия: наследование, инкапсуляция и полиморфизм. Из них непосредственное отношение к безопасности имеет инкапсуляция, которая фактически является некой формой безопасности. Согласно определению, инкапсуляция гарантирует, что соответствующие данные и код недоступны клиентам[8]. В данном случае нужно инкапсулировать весь процесс аутентификации, который начинается с обращения пользователя к системе и приводит к созданию объекта User.

---

[8] *Как справедливо заметил Даг Шмидт (Doug Schmidt), такое определение достаточно сложно реализовать в C+ + [Schmidt96a]. Например, при желании можно превратить все закрытые члены в открытые с помощью команды #define of private to public. Один из способов избежать этого заключается в том, чтобы вообще не декларировать переменные-члены в заголовочных файлах, а объявлять их вместе с остальными важ-ными реализационными деталями в отдельных неэкспортируемых файлах заголовков. Здесь очень уместен шаблон BRIDGE; но, перефразируя Ферма, доказательство слишком велико для данного комментария.*

Итак, проблема понятна. Теперь необходимо найти решение и выразить его в виде объектов. Попробуем подыскать какой-нибудь шаблон.

В данном случае для выбора шаблона не так уж много информации. Известно только, что нужно инкапсулировать создание объекта. Чтобы направить поиск в нужном направлении, вспомним, что в книге *Design Patterns* выделены три группы шаблонов согласно их целям: порождающие, структурные и поведенческие. Наиболее подходящими представляются *порождающие* шаблоны: ABSTRACT FACTORY, BUILDER, FACTORY METHOD, PROTOTYPE и SINGLETON. Кратко проанализируем каждый из них и решим, какой шаблон можно использовать в нашем случае.

Шаблон ABSTRACT FACTORY предназначен главным образом для создания семейств объектов без указания их конкретных классов. В рассматриваемом случае понятие семейства объектов не фигурирует и у нас нет намерения избегать создания экземпляра конкретного класса, а именно — класса User. Таким образом, ABSTRACT FACTORY не подходит. Шаблон BUILDER предназначен для создания сложных объектов. Он позволяет использовать один и тот же многошаговый процесс для конструирования объектов, имеющих различные представления (этот также не подходит). Назначение шаблона FACTORY METHOD аналогично назначению ABSTRACT FACTORY за исключением акцента на семействах объектов, что также не делает его более подходящим.

Шаблон PROTOTYPE служит для параметризации различных видов объектов при создании экземпляров. Вместо того чтобы писать код, вызывающий оператор new с указанием имени класса (которое не может изменяться во время выполнения), вызывается операция copy экземпляра-прототипа (во время выполнения его разрешается замещать). Чтобы изменить класс реализуемого объекта, достаточно использовать другой экземпляр-прототип.

Но, к сожалению, и этот шаблон не подходит — не нужно изменять то, *что* создается; необходимо контролировать, *как* клиенты создают экземпляры объектов User. Поскольку все могут копировать экземпляр-прототип, контроль получается менее строгий по сравнению с обычным конструктором. Помимо этого, наличие объекта-прототипа User противоречит нашей модели аутентификации.

Остается рассмотреть шаблон SINGLETON. Его назначение— гарантировать, что у класса есть только один экземпляр, и обеспечить глобальную точку доступа к этому экземпляру. В шаблоне используется беспараметрическая статическая функция-член Instance, которая возвращает единственный экземпляр данного класса. Все конструкторы защищены, так что клиент не имеет прямого доступа к ним.

На первый взгляд этот шаблон также кажется не очень подходящим — программе может понадобиться несколько объектов User. В данном случае не нужно, чтобы эк-земпляр был единственным, необходимо, чтобы *на одного пользователя* приходилось не более одного экземпляра. Тем не менее, определенная общность целей прослеживается: в обоих случаях налагается ограничение на количество экземпляров.

Рассмотрим более внимательно раздел "Последствия" шаблона SINGLETON:

> "Шаблон SINGLETON допускает произвольное количество экземпляров. Он позволяет легко изменить первоначальное решение и разрешить появление нескольких экземпляров класса Singleton, Его можно использовать *для* управления количеством экземпляров, используемых определенным приложением, требуется только изменить операцию Instance, которая предоставляет доступ к экземпляру класса Singleton."

Эврика! В нашем случае можно применить шаблон SINGLETON, нужно только переименовать операцию Instance в login и снабдить ее списком параметров:

```
static const User*::login(
    const string& loginName, const string& password
};
```

Операция login гарантирует что для определенного регистрационного имени создается только один экземпляр объекта. Для этого класс User должен содержать закрытую статическую хеш-таблицу объектов User, проиндексированную по регистрационному имени. Операция login ищет параметр LoginName в этой хеш-таблице. Если она находит соответствующий элемент User, то возвращает его, в противном случае выполняет следующие действия.

1. Создает новый объект User и связывает его с паролем.
2. Регистрирует данный объект User в хеш-таблице для последующего доступа.
3. Возвращает созданный объект User.

Ниже кратко перечислены свойства операции User::login:

- Она глобально доступна.
- Она предотвращает создание нескольких объектов User с одним и тем же ре
  гистрационным именем.
- В отличие от конструктора эта операция может возвращать 0, если регистра
  ционное имя или пароль введены неправильно.
- Приложения не могут изменять операцию login путем порождения подклас
  сов User.

Это достаточно неортодоксальное применение шаблона SINGLETON. Факт, что клиент может создавать несколько экземпляров класса User, свидетельствует о том, что мы не следовали назначению шаблона буквально. Более того, в разделе шаблона, посвященном реализации, много времени уделяется обсуждению создания подклассов класса Singleton, а в данном приложении этого не следует допускать вовсе[9].

Тем не менее, теперь появилась возможность контролировать количество экземпляров, и шаблон SINGLETON послужил основой для нашего подхода. При решении задач проектирования шаблоны не должны доминировать. Хороший шаблон — не просто описание решения конкретной задачи; он способствует более глубокому пониманию сути проблемы и позволяет приспособить решение к конкретной ситуации.

Однако шаблон SINGLETON не решил всех проблем. Например, если у нас есть операция login, следует предусмотреть соответствующую операцию logOut для удаления пользователей из системы. В связи с операцией logOut возникают важные вопросы, касающиеся управления памятью для объектов Singleton, о которых в шаблоне SINGLETON совершенно умалчивается. Эти вопросы будут более подробно рассматриваться в главе 3.

---

[9] *Чтобы не допустить порождения подклассов класса User, достаточно декларировать его конструктор(-ы) закрытыми (private).*

Как клиент использует объект User? Для ответа на данный вопрос проанализируем различные варианты использования этого объекта.

Во-первых, рассмотрим процесс регистрации. Предположим, существует программа регистрации, которая выполняется, когда пользователь хочет войти в систему (или получить доступ к ее защищенным частям). С помощью обращения User::login программа регистрации получает объект User. Затем она каким-то образом делает этот объект доступным другим приложениям, так что пользователю не нужно регистрироваться более одного раза.

Во-вторых, проследим, как приложение получает доступ к файлу, созданному не-сколько дней назад кем-то под регистрационным именем "jonny". Предположим, регистрационное имя пользователя приложения "mom", а файл является считываемым для своего создателя и несчитываемым для остальных. Тогда "mom" не следует предостав-лягь доступ к файлу. В однопользовательской системе приложение запрашивает содержимое файла с помощью вызова операции streamOut, поставляя ей некий поток:

```
void streamOut(ostream&)
```

В идеале хотелось бы, чтобы вызов выглядел аналогичным образом и в многополь-зовательской системе, но в данном случае необходимо указать пользователя, который осуществляет доступ к файлу, так как без такой ссылки невозможно гарантировать, что пользователь имеет соответствующие полномочия доступа. Ссылка может передаваться явно в виде параметра

```
void streamOut(ostream&, const User*);
```

или задаваться неявно посредством процесса регистрации. Как правило, прило-жение работает от имени одного и только одного пользователя на протяжении своего жизненного цикла. В таком случае постоянно поставлять объект User в качестве па-раметра ни к чему. Однако совместно используемое приложение может на законных основаниях предоставлять доступ нескольким пользователям и, тогда действительно необходимо указывать пользователя для каждой операции.

Таким образом, следует добавить параметр const User* в каждую операцию интерфейса класса Node, но при этом *не обязывать* клиентов поставлять его. Это можно осуществить с помощью задания параметров по умолчанию:

```
const string& getName(const User* = 0);
const Protection& getProtection(const User* = 0);

void setName(const string&, const User* = 0};
void setProtection(const Protection&, const User* = 0);

void streamIn(instream&, const User* = 0);
void streamOut(ostream&, const User* = 0);

Node* getChild(int, const User* = 0};

void adopt(Node*, const User* = 0);
void orphan(Node, const User* = 0);
```

В типичном случае, когда пользователь явно не указывается, необходима глобально доступная операция для извлечения единственного экземпляра User. Это равносильно использованию Singleton, но для повышения гибкости лучше, чтобы экземпляр Singleton задавался приложением. Таким образом, вместо одной статической операции User::instance будут использоваться статические операции get и set:

```
static const User* User::getUser ();
static void User::setUser(const User*);
```

Операция setUser позволяет приложению сделать неявным пользователем любого полученного им (предположительно законным путем) пользователя const User*. После этого программа регистрации может вызвать setUser, чтобы создать глобальный экземпляр User, который будут использовать другие приложения[10]:

```
extern const int maxTries;
// …
const User* user = 0;
for (int i = 0; i < maxTries; ++i) {
      if (user = User::login(loginName, password)) {
            break;
      } else {
            cerr <<"Log-in invalid!" << endl;
      }
}
if (user) {
      User::setUser(user);

} else {
      //слишком много неуспешных попыток регистрации;
      //заблокировать данное регистрационное имя!
      //...
}
```

Рассмотрим теперь, как такая схема защиты повлияет на реализацию метода streamOut и других шаблонных методов интерфейса Node (точнее, как *эти* методы используют объект User).

Основное отличие многопользовательского проекта от однопользовательского состоит не в самих шаблонных методах, а в примитивных операциях, возвращающих булевы значения. Например, метод streamOut приобретает следующий вид:

```
void Node::streamOut (ostream& out, const User* u) {
      User* user = u ? u : User::getUser();

      if (isReadableBy(user)) {
            doStreamOut(out);

      } else {
            doWarning(unreadableWarning) ;
      }
}
```

Вторая строка данного определения явно отличается от соответствующей строки определения этого же метода в однопользовательской системе. Локальная переменная user инициализируется указанным пользователем или, если он не указан, объектом User класса Singleton. Но более тонкое отличие содержится в третьей строке, где операция isReadable заменена на isReadableBy. Операция isReadableBy на основании хранящейся в данном узле информации проверяет, является ли узел считываемым только для своего пользователя или и для остальных тоже:

```
bool Node::isreadableBy (const User* user) {
      bool isOwner = user->getLoginName() == getOwner();
```

[10] *Имеется ввиду, что объекты User находятся в совместно используемой памяти или могут иным способом передаваться от одной программы к другой. Это важная деталь, но ее реализация не повлияет ни на разрабатываемый нами интерфейс, ни на применяемый в данном случае подход.*

```
return
    isOwner  &&
    isUserReadable()  ||
    !isOwnerReadable ();
```

Операция isReadableBy делает очевидной необходимость в операции User::getLoginName (которая возвращает регистрационное имя, связанное с данным объектом User) и интерфейсе для извлечения регистрационного имени владельца узла:

```
const string& Node::getOwner{};
```

Класс Node также нуждается в примитивных операциях типа isUserReadable и isOtherReadable, которые предоставляют более подробную информацию о праве пользователя-создателя и "других" на считывание и перезапись. Базовый класс Node может реализовать эти операции в виде средств доступа к флагам, которые он хранит в переменных экземпляров, или может передать средства хранения подклассам.

Однако хватит о деталях, вернемся на уровень проектирования.

Итак, для объекта мир делится на два лагеря — пользователь и остальные, но при таком делении невозможно учесть реально существующие отношения. Разумно, чтобы при работе над общим проектом члены группы могли получать доступ к файлам друг друга и в то же время защищать свои файлы от любопытства посторонних. Поэтому в системе UNIX существует третий лагерь с точки зрения защиты — группа. *Группа* представляет собой именованное множество регистрационных имен. Сделав узел считываемым или перезаписываемым для группы (group-readable или -writable), можно более избирательно контролировать полномочия доступа. Именно такой контроль отвечает среде коллективной разработки.

Как добавить понятие групп к нашему проекту? О группах известно следующее.

1. В группе может состоять любое число пользователей (в том числе и 0).
2. Некий пользователь может быть членом произвольного количества групп (0 и более).

Пункт 2 подразумевает ссылку, но не агрегирование: удаление группы не приводит к удалению составляющих ее пользователей.

Исходя их сказанного, группы можно представить как объекты. Вопрос заключается в том, идет ли речь о создании новой иерархии классов или только о расширении одной из уже существующих.

Единственным кандидатом на расширение является класс User. Альтернатива— определение класса Group как некой разновидности Node — бесполезна и бессмысленна. Рассмотрим, какими будут отношения наследования между классами Group и User.

Ранее мы уже рассматривали шаблон проектирования COMPOSITE. Он описывает рекурсивную связь между листовыми (Leaf) узлами, такими как файлы, и составными (Composite) узлами, такими как каталоги. Шаблон дает этим узлам одинаковые интерфейсы, что позволяет единообразно трактовать их и объединять в иерархии. Возможно, связь между пользователями и группами также удастся описать с помощью шаблона COMPOSITE: класс User будет соответствовать классу Leaf шаблона, а класс Group — классу Composite.

Вновь обратимся к разделу "Применимость" шаблона COMPOSITE. Данный шаблон используется в следующих случаях:

- для представления иерархии объектов, связанных отношениями часть — целое;
- чтобы предоставить клиентам возможность игнорировать различия между простыми и составными объектами и позволить им трактовать все объекты по лученной структуры универсальным образом.

Исходя из этого, можно заключить, что данный шаблон не подходит, и вот почему:

- В нашем случае связь не является рекурсивной. В файловой системе UNIX не разрешается составлять группы из групп, поэтому такая возможность не нужна. То, что в шаблоне предполагается рекурсивность связи, не означает, что она необходима в данном приложении.
- Пользователь может принадлежать к нескольким различным группам. Следовательно, связь нельзя назвать строго иерархической.
- Преимущества от единообразной трактовки пользователей и групп сомнительны. Что означает, например, регистрация группы или прохождение группой процесса аутентификации?

Эти доводы указывают на отсутствие связи типа Composite между классами User и Group. И тем не менее, необходимо связать эти классы, поскольку система должна отслеживать, к каким группам относится тот или иной пользователь.

Для повышения производительности необходимо задать *двунаправленное* отображение. Скорее всего, пользователей будет существенно больше, чем групп, следовательно, должна быть возможность определить всех пользователей определенной группы, не прибегая к опросу всех пользователей системы. Нахождение всех групп, в которые входит конкретный пользователь, — также важная задача, и нужно иметь возможность решать ее достаточно быстро.

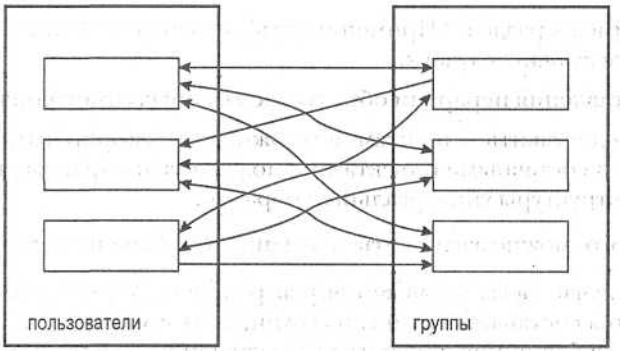
Очевидный способ реализации двунаправленного соответствия заключается в добавлении коллекции узлов к классу Group и коллекции групп к классу Node. Однако такой подход имеет два существенных недостатка:

1. Отображение сложно менять. Придется модифицировать как минимум один, а то и оба базовых класса.
2. Все объекты (включая группы, лишенные пользователей, а также пользователей, не состоящих в группах) несут затраты по поддержанию коллекции. Как минимум, каждый объект должен хранить дополнительный указатель.

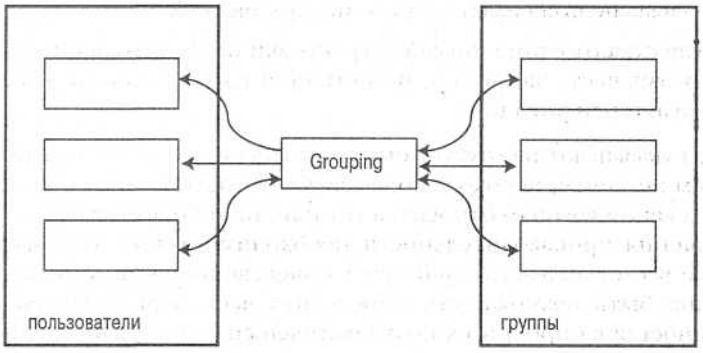
Соответствие между группами и пользователями является сложным и непостоянным. Очевидный подход распределяет ответственность за его поддержку и приводит к уже упомянутым недостаткам. Этих недостатков можно избежать, применив менее очевидный подход, который состоит в *централизации* ответственности.

Шаблон MEDIATOR дает взаимодействию объектов статус полноправного объекта. Он способствует их свободному соединению, освобождая объекты от явных ссылок друг на друга, что позволяет независимо менять их взаимосвязи.

Ниже изображена типичная ситуация перед применением шаблона: два множества взаимодействующих объектов (в шаблоне они называются *коллегами* (colleagues)), причем каждый объект одного множества содержит явные ссылки (практически) на все объекты другого множества.



После применения шаблона получается следующая картина.



Центральный объект шаблона— объект Mediator (посредник) , которому соответствует объект Grouping на второй диаграмме. Вместо явных ссылок друг на друга, коллеги ссылаются исключительно на объект-посредник.

В разрабатываемой файловой системе объект Grouping задает двунаправленное соответствие между пользователями и группами. Чтобы соответствие легче было менять, в шаблоне создан абстрактный базовый класс объектов-посредников, из которого порождаются подклассы для конкретных видов соответствий. Ниже приведен простой интерфейс посредника Grouping, который позволяет клиентам регистрировать и аннулировать связи между пользователями и группами.

```
class Grouping  {
public:
    virtual void ~Grouping getGrouping ();
    static void setGrouping(
        const Grouping*, const User* = 0
    };

    virtual void register(
        const User*, const Group*, const User* = 0
    ) = 0;
    virtual void unregister(
        const User*, const Group*, const User* = 0
     ) = 0;
```

```
    virtual const Group* grtGroup(
        const string& loginName, int index = 0
    ) = 0;
    virtual const string& getUser(
        const Group*, int index = 0
    ) = 0;
    protected:
        Grouping();
        Grouping(const Grouping&);
    };
```

Заметим, что в данном интерфейсе присутствуют статические операции get и set, аналогичные операциям, определенным для класса User в результате применения шаблона SINGLETON. Данный шаблон применен по тем же причинам: необходимо обеспечить глобальный доступ к соответствию и возможность его задания.

Заменяя во время выполнения объект Grouping, можно одним движением изменить соответствие. Например, привилегированный пользователь может иметь право переопределить соответствие исходя из административных соображений. Изменение соответствия должно быть тщательно защищенной операцией, поэтому клиенту следует вызывать setGrouping, поставляя const User*. Аналогично, пользователь, выполняющий операции register и unregister, должен иметь полномочия устанавливать или аннулировать данное соответствие.

Последние две операции getGroup и getUser позволяют выявить связанные с пользователем группы и связанных с группой пользователей. Необязательные параметры index предоставляют клиентам простой способ перешагивать через несколько значений. В конкретных подклассах можно определить другую реализацию этих операций. Заметьте, что данные операции работают не с самими объектами User, а со строками для соответствующих регистрационных имен. Это позволяет любому клиенту увидеть ассоциации, не извлекая и не поставляя объекты User, к которым он не имеет отношения.

Один из недостатков шаблона MEDIATOR заключается в тенденции к монолитности классов Mediator. Поскольку класс Mediator инкапсулирует потенциально сложные взаимодействия, он может стать чрезмерно сложным, что затруднит его поддержку и расширение. Применение других шаблонов способно помочь в решении этих проблем. Например, чтобы частично изменить поведение посредника в подклассах, можно использовать шаблон TEMPLATE METHOD. Шаблон STRATEGY позволяет сделать то же самое с большей гибкостью, а с помощью шаблона COMPOSITE можно скомпоновать посредник из меньших компонентов путем рекурсивной композиции.

Краткий итог

В ходе проектирования файловой системы мы неоднократно применяли шаблоны. В основе системы лежит рекурсивная древовидная структура, полученная с помощью шаблона COMPOSITE. Шаблон PROXY дополнил эту структуру поддержкой символических связей, а шаблон VISITOR позволил нам легко и безболезненно добавлять новые функциональные возможности.

Что касается защиты, то шаблон TEMPLATE METHOD обеспечил ее на самом примитивном уровне, т.е. на уровне отдельных операций. Этого вполне хватало для обеспечения защиты в однопользовательской среде, но переход к многопользовательской среде потребовал введения дополнительных абстракций — поддержки регистрации, пользователей и групп. С помощью шаблона SINGLETON удалось обеспечить поддержку регистрации и пользователей путем инкапсуляции и контроля процедуры регистрации, а также задания неявного пользователя, к которому можно получить доступ и заменить любым объектом системы. Наконец, шаблон MEDIATOR обеспечил простой и гибкий способ связывания пользователей с группами, членами которых эти пользователи являются.

На рис. 2.4 представлены использованные шаблоны и воплощающие их классы. Систему обозначений несколько лет назад предложил Эрих (Erich Gamma) и назвал ее *pattern:role annotation.* Классы представлены затененными прямоугольниками, каждый из которых содержит названия связанных с классом шаблонов и/или участников. Для краткости иногда указывается только имя участника, если название шаблона очевидно и его пропуск не приведет к неоднозначности.

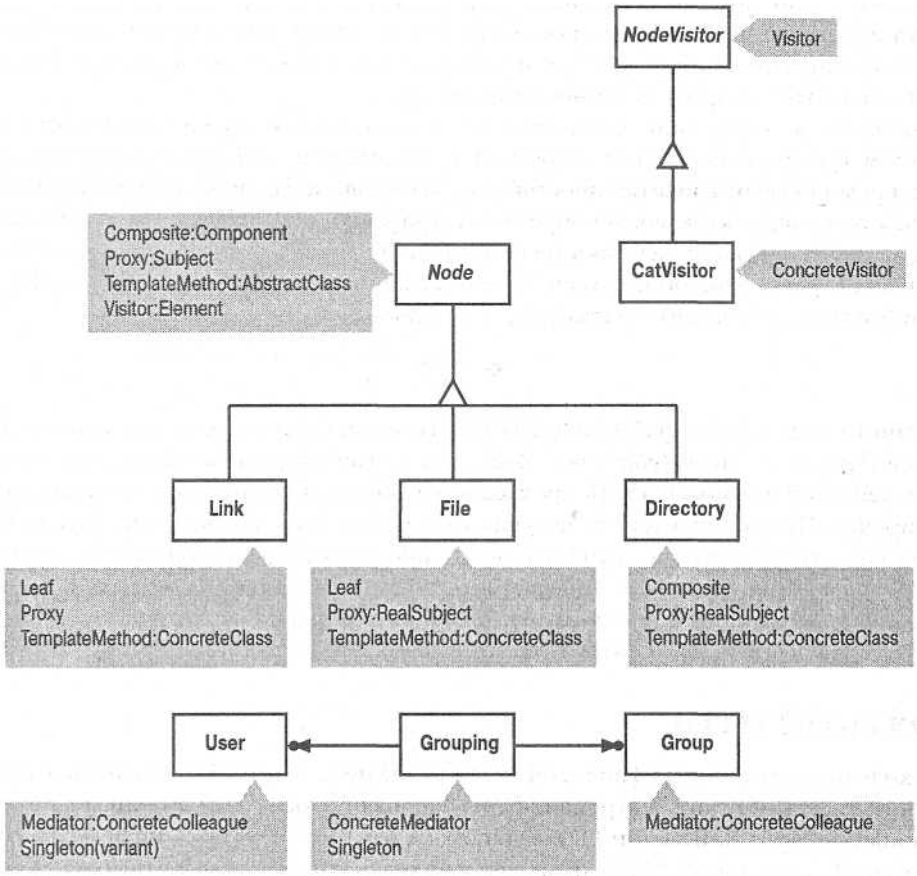


РИС. 2.4. *Проект файловой системы*

Отсутствие дополнительных линий и использование прямоугольников с контрастным фоном позволило сделать рисунок более понятным — кажется, будто описания шаблонов находятся в иной плоскости, чем структура классов. Данная нотация действительно позволяет уменьшить количество линий, поскольку некоторые артефакты внутренне присущи шаблонам и их можно опустить. Например, связь агрегирования между классами Directory и Node опущена, так как это внутренне присущая шаблону COMPOSITE связь Component-Composite и можно лишний раз не акцентировать на ней внимание.

Предложенная Эрихом нотация легко читаема, информативна и масштабируема, особенно в сравнении с представленной на рис. 2.3 нотацией, навеянной диаграммами Венна (Venn).

Глава 3

Темы и вариации

В этой главе предлагается новый углубленный взгляд на некоторые исходные шаблоны, описанные "бандой четырех", — SINGLETON, OBSERVER, VISITOR и MEMENTO, а также новый шаблон GENERATION GAP. Приводимые здесь рассуждения призваны убедить читателя, что шаблон никогда нельзя считать полностью законченным.

Ликвидация объекта Singleton

Шаблон SINGLETON чрезвычайно прост. Его назначение сформулировано так:

> "Гарантировать, что класс содержит единственный экземпляр, и обеспечить глобальную точку доступа к нему."

Тем не менее, данный шаблон обладает достаточной гибкостью. При проектировании файловой системы он помог нам инкапсулировать создание объектов User, чтобы пользователи файловой системы имели право осуществлять доступ только к своим собственным файлам. Для получения объекта User клиентская программа вызывает статическую операцию User: : login:

```
static const User* User::login(
    const string& loginName, const string& password
```

Эта операция представляет собой несколько видоизмененную версию статической операции Instance шаблона SINGLETON. В исходной версии шаблона операция Instance ограничивала количество экземпляров класса Singleton (в нашем случае— класса User) единственным экземпляром. В качестве одного из следствий шаблона указывалась возможность *контролировать* число экземпляров, а не только препятствовать появлению нескольких экземпляров в классе. Мы воспользовались этой возможностью, чтобы предотвратить появление нескольких экземпляров User *для каждого отдельного пользователя.* Таким образом, если приложение одновременно обслуживает многих пользователей, оно может создавать несколько экземпляров.

Как уже отмечалось, в описании шаблона подозрительно умалчивается о том, кто, как и когда удаляет экземпляры Singleton. Слова "delete" и "destructor" нигде в шаблоне не встречаются. Восполним данный пробел и попутно узнаем много нового об этом небольшом, но на удивление богатом шаблоне.

Как и во всяком самостоятельном классе, в классе Singleton должен быть определен деструктор. Если класс Singleton может порождать подклассы, деструктор следует объявить виртуальным. Пока все просто. Теперь перейдем к более сложным вопросам. Каким должен быть этот деструктор: открытым, закрытым или защищенным?

Можно подумать: "Что здесь сложного? Сделаем его открытым, и все." Это будет означать, что удаление экземпляра Singleton производится *явно,* и за него отвечает клиент.

Однако существует весомый аргумент против такого решения. Шаблон SINGLETON возлагает ответственность за создание объектов исключительно на класс Singleton. Чтобы получить экземпляр Singleton, клиенты обращаются к данному классу. Если клиент удаляет экземпляр Singleton без ведома класса Singleton, то с этого момента класс Singleton будет выдавать "висящие ссылки", указывающие на объект, который больше не существует. Обязанности класса Singleton означают, что он *владеет* создаваемыми экземплярами, а владение подразумевает ответственность за удаление. Этим шаблон SINGLETON отличается от других порождающих шаблонов, таких как ABSTRACT FACTORY и FACTORY METHOD, в которых владение созданными экземплярами не сохраняется.

Таким образом, избежать неприятностей с открытым деструктором можно только в том случае, если будут выполнены следующие условия.

1. Деструктор удаляет статический экземпляр и убирает ссылки на него. Тогда последующий вызов операции Instance будет происходить так, как будто он производится впервые.
2. Клиенты не сохраняют ссылки на объект Singleton. В противном случае они могут остаться с висящими ссылками.

Эти условия достаточно ограничительны, чтобы сделать открытый деструктор скорее исключением, чем правилом.

Рассмотрим, например, как и когда удаляются объекты User в проектируемой нами файловой системе. Предположим, клиентам разрешено явно удалять объекты User с помощью обычного оператора delete. Можно пойти еще дальше и определить статическую операцию logOut, обратную login (неважно где, так как интерфейс удаления не имеет большого значения). Однако в настоящий момент у класса User нет возможности узнать, какие клиенты имеют ссылки на объекты User. Поэтому при удалении некоторого объекта User клиенты могут остаться с висящими ссылками, что совершенно недопустимо.

Вероятно, нужен какой-то механизм аннулирования регистрации пользователей (скажем, для целей учета), однако возможность возникновения висящих ссылок делает удаление непригодным для данного механизма. Иными словами, не следует путать аннулирование регистрации с удалением объекта User. Независимо от того, какой интерфейс будет выбран для отмены регистрации пользователя, он не должен содержать явное удаление объекта User.

Данный пример показывает, когда следует отказаться от идеи использования открытого деструктора. Основания для отказа от *закрытого* деструктора найти гораздо проще — достаточно предположить, что нужно разрешить порождение подклассов классом Singleton. Для нашей файловой системы это предположение не подходит, но в общем случае оно вполне допустимо. Таким образом, остается один вариант — защищенный деструктор.

Рассмотрим, при каких обстоятельствах удаляется объект Singleton.

Одно из свойств объектов Singleton заключается в том, что они, как правило, долго-живущие (часто они существуют на протяжении жизни программы). Эти объекты удаляют не столько для освобождения места, сколько для того, чтобы свернуть их неким упорядоченным образом. Точно так закрывают файлы, разблокируют ресурсы, разрывают сетевые соединения и т.д., при этом не возникает никаких аварийных ситуаций. Если с объектами Singleton нужно сделать то же самое, следует подождать до окончания работы программы, тогда язык C++, возможно, позволит выполнить их удаление *неявно.*

В языке C++ при окончании программы статические объекты автоматически удаляются: гарантируется, что будут вызваны их деструкторы и освобождено пространство, хотя порядок выполнения вызовов не оговаривается. На данном этапе предположим, что порядок не имеет значения; в программе используется только один Singleton, а если их несколько, то порядок удаления не важен. Это означает, что можно определить класс Singleton следующим образом:

```
class Singleton  {
public:
     static Singleton*  instance ();

protected:
     Singleton();
     Singleton(const  Singleton&);
     friend class SingletonDestroyer;
     virtual  ~Singleton()  {  }
private:
     static Singleton*  _instance;
     static SingletonDestroyer _destroyer;
};
Singleton*  Singleton::_instance = 0;
SingletonDestroyer Singleton::_destroyer;
Singleton*  Singleton::instance  ()  {
     if  (!_instance)  {
            _instance = new Singleton;
            _destroyer.setSingleton(_instance);
      }
     return _instance;
}
```

SingletonDestroyer — это класс, единственной целью которого является разрушение определенного объекта Singleton:

```
class SingletonDestroyer  {
public:
     SingletonDestroyer(Singleton*  = 0);
     ~SingletonDestroyer ();
     void setSingleton(Singleton*  s) ;
     Singleton*  getSingleton();
private:
     Singleton*  =singleton;
};
SingletonDestroyer::SingletonDestroyer  (Singleton*  s)  {
_singleton = s;
}
SingletonDestroyer::~SingletonDestroyer () {
     delete _singleton;
}
void SingletonDestroyer::setSingleton (Singleton* s) {
     _singleton = s;
}
Singleton* SingletonDestroyer::getSingleton () {
     return _singleton;
}
```

Класс Singleton объявляет статический член SingletonDestroyer, который автоматически создается при запуске программы. Когда пользователь впервые вызывает Singleton::instance, объект Singleton не только создается, но и передается

операцией instance статическому объекту SingletonDestroyer, тем самым фактически объекту SingletonDestroyer передается владение данным объектом Singleton. По завершении программы объект SingletonDestroyer автоматически разрушается, и объект Singleton ликвидируется вместе с ним. Таким образом разрушение объекта Singleton становится неявным.

Обратите внимание на служебное слово friend в декларации класса Singleton. Оно необходимо для того, чтобы разрушитель мог получить доступ к защищенному деструктору объекта Singleton. Другого выхода нет, поскольку ранее мы уже рассматривали аргументы против открытого деструктора. Этот пример иллюстрирует наиболее оправданный способ использования friend, когда данное свойство языка используется для того, чтобы определить дополнительный уровень защиты, а не помогать ликвидировать недостатки плохого проекта.

Чтобы максимизировать повторное использование кода, особенно если в программе существует несколько типов объектов Singleton, можно определить шаблонный класс Destroyer[1]:

```
template <class DOOMED>
class Destroyer {
public:
    Destroyer(DOOMED* = 0);
    ~Destroyer();

    void setDoomed (DOOMED*);
    DOOMED* getDoomed();

private:
   //Предотвращает создание копий Destroyer
   //пользователями с целью избежать повторного
   //удаления
   Destroyer(const Destroyer<DOOMED>&);
   void operator=(const Destroyer<DOOMED>&);

private:
    DOOMED* _doomed;
};
template <class DOOMED>
Destroyer<DOOMED>::Destroyer (DOOMED* d) {
        _doomed = d;
}
template <class DOOMED>
Destroyer<DOOMED>::~Destroyer () {
         delete  _doomed;
}
template <class DOOMED>
void Destroyer<DOOMED>::setDoomed (DOOMED* d) {
        _doomed = d;
}
template <class DOOMED>
DOOMED* Destroyer<DOOMED>::getDoomed () {
         return  _doomed;
}
```

Тогда определение класса Singleton примет следующий вид:

Приверженцы процесса стандартизации легко узнают в этом классе шаблон стандартного библиотечного класса auto_ptr.

```
class Singleton {
public:
    static Singleton* instance();

protected:
    Singleton();
    Singleton(const Singleton&);
    friend class Destroyer<Singleton>;
    virtual ~Singleton() { }
private:
    static Destroyer<Singleton> _destroyer;
};
Destroyer<Singleton> Singleton::_destroyer;
Singleton* Singleton::instance () {
    if (!_instance) {
        _instance = new Singleton;
        _destroyer.setDoomed(_instance);

    return _instance;
}
```

При неявной деструкции могут возникнуть две проблемы. Во-первых, неявной деструкцией нельзя воспользоваться, когда необходимо удалить экземпляр Singleton до окончания программы. В таком случае трудно предложить подход, который не требует явной ликвидации. Понадобится или добавить некий механизм (например, счетчик ссылок), чтобы справиться с проблемой висящих ссылок, или заставить клиента получать доступ к экземпляру Singleton исключительно посредством операции Singleton::instance.

Один из способов осуществления второго варианта состоит в следующем: (1) сделать так, чтобы операция instance возвращала ссылку на экземпляр Singleton и (2) запретить копирование и инициализацию, объявив конструкторы присваивания и копирования закрытыми (private):

```
class Singleton {
public:
    static Singletons instance();
protected:
// …
private:
    Singleton(const Singleton&);
    Singleton& operator = (const Singleton&);
    //
};
```

К сожалению, в данном подходе не предусмотрена "защита от дурака", поскольку клиент всегда может получить адрес значения, возвращаемого instance, или отбросить эти предосторожности вовсе. Но это не так уж важно, поскольку, как отмечалось ранее, шаблон SINGLETON преимущественно используется для создания долгоживу-щих объектов и явное удаление встречается не слишком часто.

Вторая и более серьезная проблема возникает тогда, когда в программе имеется несколько объектов Singleton, зависящих друг от друга. В таком случае порядок разрушения объектов может оказаться существенным.

Рассмотрим проект нашей файловой системы, в котором дважды применялся шаблон SINGLETON. Первый раз он использовался для контроля количества объектов User, производимых классом User. Второй раз шаблон применялся для того, чтобы гарантировать единственность объекта Grouping, задающего соответствие между пользователями и

группами, к которым эти пользователи принадлежат. Объект Grouping позволяет устанавливать не только индивидуальную защиту, но и защиту для групп пользователей. Поскольку иметь несколько объектов Grouping бессмысленно и даже опасно, мы сделали класс Grouping одноэлементным множеством (Singleton).

Объект Grouping поддерживает ссылки как на объекты User, так и на объекты Group. Он не владеет объектами User, но может владеть объектами Group. В любом случае желательно удалять объект Grouping *перед* удалением объектов User, поскольку возможно появление висящих ссылок (которое, скорее всего, не вызовет проблем, так как объект Grouping вряд ли будет заниматься их разыменованием в процессе своего разрушения, однако гарантировать это нельзя).

В этой связи следует отметить, что использующий разрушитель подход, основанный на языке с неконкретизированным механизмом реализации, начинает давать сбои, когда важен порядок разрушения объектов. Если приложению требуется несколько зависимых одноэлементных множеств, возможно, придется вернуться к явной ликвидации. Ясно одно: нельзя использовать более одного разрушителя, если деструкторы объектов Singleton зависят друг от друга.

В качестве альтернативы можно отказаться от разрушителей и вместо этого использовать стандартную функцию atexit () , как предложил Тим Пирлз (Tim Peierls, [Peierls96]):

> "Я считаю, что функция atexit () является хорошим способом убирать экземпляры Singleton в C++, когда нужно обеспечить существование единственных экземпляров на протяжении жизни программы без замен. Разрабатываемый стандарт многое обещает:
>
> §3.6.3, пункт 1. Функцию atexit () из библиотеки <cstlib> можно использовать для указания функции, вызываемой при выходе. Если используется atexit (), реализация не должна разрушать объекты, инициализированные перед вызовом atexit () , до тех пор, пока не будет вызвана функция, указанная в вызове atexit ().
>
> Данный подход может не сработать только в том случае, если статически инициализированный объект, деструктор которого зависит от некоторого экземпляра Singleton, инициализируется после создания этого экземпляра Singleton, т.е. посредством некой другой статической инициализации. Это свидетельствует о том, что классы (со статическими экземплярами) не должны зависеть от экземпляров Singleton при разрушении. (Или, по крайней мере, должны иметь возможность проверять существование требуемого экземпляра Singleton в процессе деструкции.)"

Хотя в данном случае удалось обойтись без разрушителей, реальная проблема — удаление взаимозависимых экземпляров Singleton— осталась. Кто же займется "сборкой мусора"?

Довольно давно я получил следующее сообщение от Скотта Мейерса [Meyers95]:

> "Моя версия шаблона SINGLETON весьма похожа на вашу, но вместо статического класса и функции instance, возвращающей указатель, я использую статическую функцию и возвращаю ссылку:

```
Singleton& Singleton::instance () {
    static  Singleton s;
    return  s;
}
```

Эта версия, похоже, обладает всеми достоинствами вашего решения (нет ни одной конструкции if, независимость от порядка инициализации транслируемых блоков и т.д.) плюс позволяет использовать синтаксис объектов вместо синтаксиса указателей. Мое решение значительно уменьшает вероятность того, что вызывающая программа по невнимательности удалит экземпляр Singleton при попытке избежать утечки памяти.

Может быть, я что-то не учел, и существует причина, по которой следует возвращать указатель на статический класс вместо ссылки на статическую функцию?"

Статическая функция несколько усложняет создание подклассов Singleton, так как instance всегда создает объект типа Singleton. (Более подробно о расширении класса Singleton см. *Design Patterns,* с.130), а о непреднамеренном удалении экземпляра Singleton не стоит беспокоиться, если его деструктор не является открытым. Однако в конечном счете различие между исходным подходом и подходом, предложенным Мейерсом, невелико, во всяком случае для однопоточных приложений.

Позднее Эрих Гамма (Erich Gamma) заметил более существенное осложнение, связанное с предложенным Скоттом вариантом [Gamma95]:

"Оказывается, что предложенный Скоттом подход невозможно сделать безопасным в многопоточной системе, если несколько процессов могут вызывать instance. Проблема в том, что некоторые C++-компиляторы генерируют внутренние структуры данных, которые нельзя защитить посредством блоков. В таких ситуациях придется получать блок в месте вызова — довольно неудачное решение."

Чуть позднее аналогичную проблему обнаружил и попытался устранить Даг Шмидт (Doug Schmidt, [Schmidt96b]):

"Шаблон Double-Checked Locking (блокировка с двойной проверкой) [SH98] появился на свет, когда я просматривал гранки рубрики Джона Влиссидеса (John Vlissides) "Разработка шаблонов" для апрельского номера журнала за 1996 год. В своей заметке Джон рассказывал о применении шаблона SINGLETON для обеспечения защиты в многопользовательской файловой системе. По иронии судьбы, у нас незадолго до этого возникли непонятные проблемы с утечкой памяти в многопоточных версиях ACE на мультипроцессорах.

Прочитав заметку Джона, я внезапно понял, что проблему вызывают несколько экземпляров Singleton, инициализированных благодаря условиям состязательности. Увидев связь между этими событиями и выделив основные движущие силы (избежать ненужной блокировки для нормального использования объекта Singleton), я легко нашел решение."

Спустя месяц Даг прислал мне следующее сообщение [Schmidt96c]:

"Один из моих аспирантов (Тим Харрисон (Tim Harrison)) недавно реализовал библиотечный C++-класс под названием Singleton, который в общих

чертах превращает в одноэлементные множества (Singleton) существующие классы. Мы используем этот класс в ACE, и его можно назвать квазиполезным. Положительное свойство заключается в том, что он автоматизирует применение шаблона Double-Checked Locking и упрощает параметризацию LOCK-стратегии. Предлагаю ознакомиться с ним.

```
template <class TYPE, class LOCK>
class Singleton {
public:
    static TYPE* instance ();
protected:
    static TYPE* _instance() ;
    static LOCK _lock;
};

template <class TYPE, class LOCK>
TYPE* Singleton<TYPE, LOCK>::instance() {
    //действия шаблона Double Check...

    if (_instance == 0) {
       Guard<LOCK> monitor(_lock);
        if (_instance ==0) _instance = new TYPE;
    }
    return _instance;
}
```

Я был заинтригован, особенно определением "квази-полезный". Я спросил, означают ли его слова, что данный подход не предотвращает создание нескольких объектов базового типа (тогда такой тип не является одноэлементным множеством), на что получил следующий ответ [Schmidt96d]:

"Да, это действительно так. Еще одна проблема состоит в том, что многие C++-компиляторы (например, g++) не реализуют статические данные-члены внутри шаблонов. В этом случае приходится реализовать статический метод instance примерно такого вида:

```
template <class Type, class LOCK>
TYPE* Singleton<TYPE, LOCK>::instance () {
    static TYPE* _instance = 0;
    static LOCK _lock;

    if (_instance ==0) //. . .

return  instance;
}
```

Ох уж эта межплатформенная переносимость C++! ;-)"

Я написал ему, что для того, чтобы сделать некий класс одноэлементным множеством (Singleton), можно сделать его производным классом этого шаблона, передавая данный подкласс шаблону в качестве параметра (как это описано у Коплина [Coplien95]). Например:

```
class User : public Singleton<User, Mutex> {...}
```

Таким образом удается сохранить семантику Singleton и избежать описания всех его многопоточных разновидностей.

Замечание. Я самостоятельно не опробовал эту вариацию и не имел случая использовать ее, но она мне нравится с эстетической точки зрения. Я привык считать SINGLETON одним из наиболее тривиальных наших шаблонов, которому далеко до таких шаблонов, как COMPOSITE, VISITOR и др.; возможно, этим и объясняется его слабость в некоторых вопросах.

Проблемы, связанные с шаблоном OBSERVER

Отрасль информационных технологий знаменита своими отказами от обязательств. Разработчики могут с легкостью отказаться от всякой ответственности за свои творения. Следующее предупреждение выдержано в том же духе:

> ПРЕДУПРЕЖДЕНИЕ. Данный раздел содержит умозрительные проекты, основанные исключительно на общих рассуждениях. Автор и издатель не дают никаких гарантий относительно этих проектов, но вы можете их использовать по своему усмотрению.

В данном разделе я размышляю о проблеме проектирования, которая занимает меня уже на протяжении 10 лет.

Предположим, нужно создать каркас для бизнес-приложений, которые манипулируют примитивными данными, такими как суммы в долларах, имена, адреса, проценты и т.п. Данные представляются посредством одного или нескольких интерфейсов пользователя: фиксированный текст для неизменных алфавитно-цифровых данных; текстовое поле ввода для редактируемых данных; клавиши, кнопки и раскрывающиеся меню для специально оговоренных вводных данных; визуальные представления, такие как круговые и столбчатые диаграммы и всевозможные графики.

Важно сделать так, чтобы изменения интерфейса пользователя не влияли на функциональные возможности, а изменения функциональных возможностей— на интерфейс пользователя. Поэтому интерфейс пользователя отделяется от лежащих в основе данных приложений. Фактически, получается такое же разбиение, как предусмотрено в основанной на Smalltalk модели Model-View-Controller (MVC) [KP88]. MVC не только отделяет данные приложений от интерфейса пользователя, но и допускает существование нескольких пользовательских интерфейсов с одними и теми же данными.

В книге *Design Patterns* описан шаблон OBSERVER, который показывает, как осуществить такое разбиение. OBSERVER фиксирует связи между примитивными данными и их всевозможными представлениями следующим образом.

1. Каждый фрагмент данных инкапсулируется в некотором объекте Subject (субъект, который соответствует *модели* в MVC).
2. Каждый новый пользовательский интерфейс определенного субъекта инкапсулируется в некотором объекте Observer (наблюдатель, который соответствует *представлению* в MVC).
3. Субъект может иметь сразу несколько наблюдателей.
4. Изменяясь, субъект уведомляет об этом своих наблюдателей.
5. Наблюдатели, в свою очередь, опрашивают свои субъекты с целью получения информации, которая влияет на их внешний вид, и обновляются в соответствии с этой информацией.

Субъект хранит основную информацию, а наблюдатели обновляются всякий раз при изменении информации субъекта. Когда пользователь сохраняет результаты своей работы, сохраняется именно субъект; наблюдатели сохранять не нужно, поскольку отображаемая ими информация поступает из субъекта.

Рассмотрим пример. Чтобы пользователь мог изменить числовое значение, например, процентную ставку (Rate), приложение предлагает поле текстового ввода и пару клавиш, позволяющих двигаться вверх-вниз, как показано на рис. 3.1. Об изменении субъекта, хранящего значение процентной ставки (скажем, вследствие того, что пользователь увеличил это значение на один пункт, нажав кнопку со стрелкой вверх), уведомляется его наблюдатель — иоле текстового ввода. В ответ данное поле производит корректировку своего значения, чтобы отразить новое значение процентной ставки.

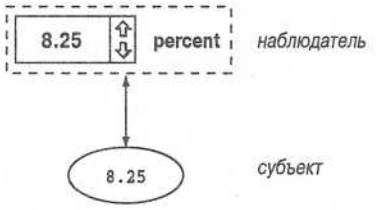


РИС. 3.1.

Теперь предположим, что помимо примитивных данных использующее каркас приложение нуждается в более высокоуровневых абстракциях, таких как займы, контракты, описания бизнес-партнеров и продуктов. Хотелось бы скомбинировать эти абстракции с уже имеющимися на более низком уровне субъектами и наблюдателями.

На рис. 3.2 представлен интерфейс пользователя для ввода информации о займе. Данный интерфейс реализован в виде наблюдателя некоторого субъекта. На рис. 3.3 показано, что этот наблюдатель в действительности является композицией элементарных наблюдателей, а субъект — композицией соответствующих элементарных субъектов.

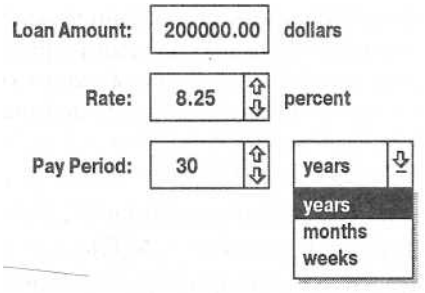


РИС. 3.2. *Интерфейс пользователя для ввода*

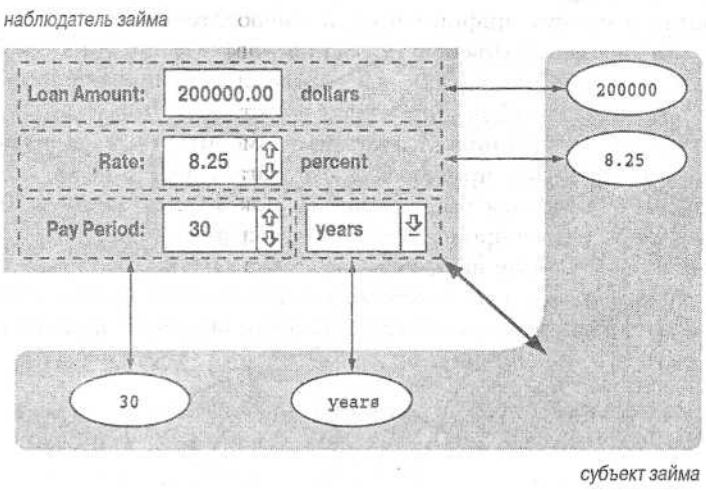


РИС. 3.3. *Субъект и наблюдатель, представляющие информацию о займе*

Такой проект имеет четыре положительных свойства.

1. Можно определять, модифицировать и расширять субъекты независимо от наблюдателей и наоборот, что позволяет осуществлять поддержку и вносить усовершенствования.
2. Приложение может содержать только необходимые ему функциональные возможности. Это особенно важно, когда каркас предлагает много функциональных возможностей. Если, например, приложение не нуждается в графическом представлении данных, оно может не включать в себя наблюдатели в виде разнообразных диаграмм.
3. К одному субъекту можно присоединить произвольное количество наблюдателей. Поле текстового ввода и кнопки вверх-вниз, вероятно, будут реализованы как отдельные наблюдатели. (Из соображений простоты это не показано на рисунках.) Можно даже создать невизуальные наблюдатели— например, определить объект, который фиксирует изменения данных субъекта, не модифици руя реализацию этого субъекта.
4. Можно реализовать новые субъекты и наблюдатели на основе уже существующих, что соответствует принципам повторного использования.

Выглядит очень обнадеживающе, но у этого подхода имеются и недостатки.

На рис. 3.4 показано, что эти композиции объектов можно рассматривать как вложенные иерархии. Субъект займа (loan subject) содержит свои примитивные субъекты, а наблюдатель займа (loan observer) — соответствующие примитивные наблюдатели. Обратите внимание на обилие объектов (овалы) и ссылок (линии). Имеются связи не только между субъектом и наблюдателем займа, но и между каждым элементарным субъектом и его наблюдателем.

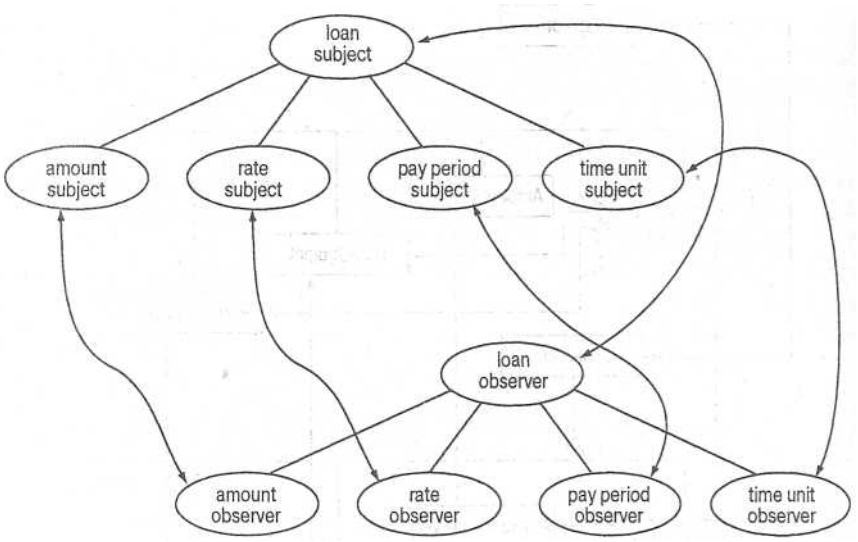


РИС. 3.4. *Структуры субъекта и наблюдателя займа*

Таким образом, шаблон OBSERVER приводит к возникновению значительной динамической избыточности. Если писать код субъекта и наблюдателя займа с нуля, можно легко обойтись без большинства этих связей, не говоря уже о многих объектах. Все это цена, которую приходится платить за возможность повторного использования.

Но это только часть проблемы: помимо динамической наблюдается и статическая избыточность. Рассмотрим классы, реализующие данные структуры объектов. Шаблон OBSERVER предписывает выделить в базовом классе иерархии классов Subject и Observer, а также определить протокол уведомления и интерфейс для прикрепления и открепления наблюдателей. Подклассы ConcreteSubject реализуют конкретные субъекты, добавляя любой интерфейс, который нужен их конкретным наблюдателям для отображения изменений. Тем временем подклассы Concrete Observer определяют различные представления своих субъектов, а их операция Update описывает, как они обновляются.

Данные статические взаимосвязи представлены на рис. 3.5. Весьма запутанно, не правда ли? Это достаточно типично для параллельных иерархий. Помимо дополнительных затрат на кодирование и поддержку данных иерархий, имеются и концептуальные потери: программисты вынуждены разбираться с гораздо большим количест вом классов, интерфейсов и подклассов.

Не лучше ли обойтись одной иерархией классов и одной иерархией экземпляров? Беда в том, что нам не хотелось бы отказываться от четкого размежевания данных приложений и интерфейса пользователя, которое дают отдельные иерархии классов и объектов. Где же выход?

Подумаем, что является основные источником расходов при разбиении на субъект-наблюдатель. Очевидно, это дополнительные затраты памяти на хранение одно временно существующих объектов и связей. Параллельные иерархии объектов требуют удвоенного количества объектов и внутрииерархических связей помимо связей между иерархиями. Получается, что надо хранить слишком много информации. Нуж но выяснить, насколько она необходима и действительно ли часто требуется доступ к этой информации. Одно дело отделить интерфейс пользователя от данных приложений, но нужно ли постоянно поддерживать уйму связей между ними?

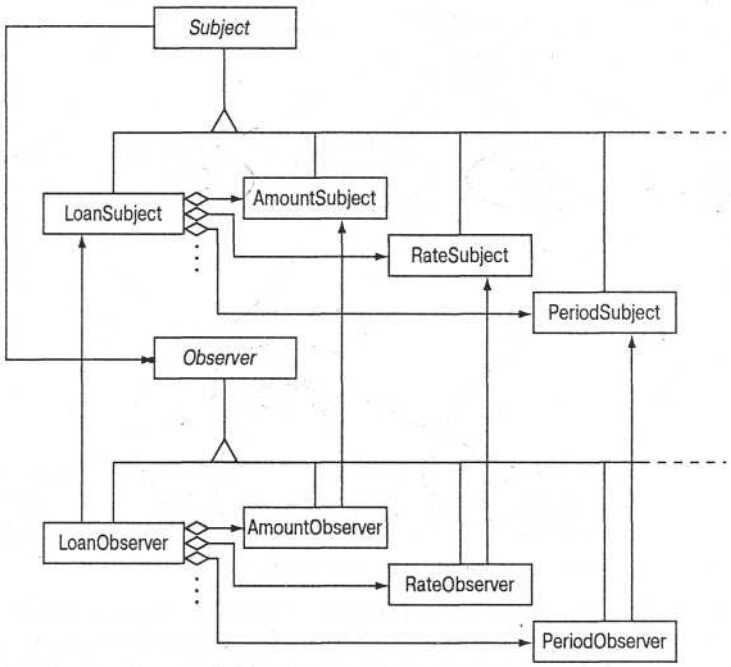


РИС. 3.5. *Структуры классов субъекта и наблюдателя займа*

Предположим, ответ на оба эти вопроса отрицательный, какова тогда альтернатива? Если ограничиться одной иерархией объектов, можно гарантированно избавиться и от лишних объектов, и от связей. Нужно найти способ представлять информацию субъекта, не поддерживая параллельную иерархию наблюдателей и не смешивая без разбору эти две иерархии в одну.

Поскольку здесь речь идет о памяти, стоит подумать над классическим компромиссом между временем и пространством. Вместо *хранения* информации попробуем *вычислять* ее непосредственно в процессе выполнения. Нет необходимости хранить информацию, которую можно воссоздать по требованию, при условии, что она будет использоваться не очень часто. Остается выяснить, что такое "очень часто". Это настолько часто, чтобы оказать неприемлемое влияние на производительность.

К счастью, число ситуаций, когда наблюдатели производят какие-то действия, очень невелико, по крайней мере в нашем приложении. Главным образом, эти ситуации возникают в одном из трех случаев.

1. При изменении субъекта.
2. При вводе пользователя.
3. При необходимости создать (перестроить) графическое представление интерфейса пользователя (или его части).

Как раз эти случаи должен обрабатывать код наблюдателя. Если устранить объекты-наблюдатели, то именно тогда придется на ходу вычислять, что нужно делать.

Не следует думать, что нам не понадобятся *никакие* объекты для выполнения работы наблюдателей. Мы будем использовать объекты, только значительно меньшее их количество,

чем предлагается шаблоном OBSERVER (желательно фиксированное число, а не пропорциональное размеру иерархии субъектов). Вместо того, чтобы хранить огромное множество связей с каждым субъектом, мы будем вычислять необходимые связи.

Три описанные выше ситуации обычно предусматривают обход структуры наблюдателя или структуры субъекта (а зачастую и обеих). Изменение субъекта и результирующее изменение его наблюдателя может привести к необходимости обхода всей структуры наблюдателя, чтобы, например, перестроить затронутые этим изменением элементы пользовательского интерфейса. Аналогично вычисление, которое позволяет определить, на каком элементе пользовательского интерфейса был произведен щелчок мыши, включает в себя по крайней мере частичный обход структуры наблюдателя. То же можно сказать и о перестройке графического представления интерфейса.

Поскольку в любой из этих ситуаций все равно необходимо совершать обход, в процессе обхода можно вычислить то, что пришлось бы в противном случае хранить. Обход предоставляет достаточно возможностей, чтобы выполнить задачи, которые субъекты не в состоянии осуществить в одностороннем порядке.

Например, можно обновить внешний вид модифицированного субъекта, совершая обход структуры субъекта и перестраивая пользовательский интерфейс во всей его полноте. Без сомнения, такой упрощенный подход не очень эффективен, поскольку предположительно только небольшая часть интерфейса пользователя нуждается в изменении. К счастью, альтернативный подход также достаточно прост. Пусть каждый субъект содержит специальный бит[2], который показывает, был ли субъект изменен. При обходе попутно производится переустановка этих специальных битов, поэтому, перестраивая интерфейс, можно игнорировать все субъекты, кроме "меченых", что сделает эффективность данного подхода сопоставимой с эффективностью шаблона OBSERVER.

Однако возникает закономерный вопрос: как узнать, что делать на каждом шаге обхода? И где реализованы эти знания?

При использовании шаблона OBSERVER каждый объект Observer знает, как строить свой фрагмент представления. Код для представления определенного объекта ConcreteSubject находится в соответствующем классе ConcreteObserver. Субъект поручает свое представление своему наблюдателю (-ям). Именно эти поручения приводят к возникновению множества дополнительных объектов и ссылок.

Отказавшись от наблюдателей, необходимо найти новое место, куда можно поместить код представления субъекта. Следует учесть, что представление строится во время обхода инкрементным образом (один субъект за другим) и должно меняться в зависимости от типа субъекта. Код, выполняемый в каждой точке обхода, зависит от двух факторов: типа субъекта и типа его представления. Если есть только иерархия субъектов, как выбрать правильный код?

Эта проблема является следствием удаления из субъекта презентационных функциональных возможностей. Необходимо найти такое решение, которое позволит не объединять в одном объекте свойства субъекта и наблюдателей, а также не прибегать к неоправданным динамическим проверкам типа.

---

[2] *Его реализация должна быть скрыта за интерфейсом get / set, поэтому реальный объем выделенной памяти может меняться.*

Еще раз о VISITOR

Итак, мы вновь столкнулись с проблемой, подобной той, которая уже рассматривалась в главе 2 при проектировании файловой системы. Нужно было сделать так, чтобы объекты файловой системы (файлы и каталоги) могли выполнять множество разнообразных операций, и при этом избежать добавления новых операций в класс Node (абстрактный базовый класс объектов файловой системы), поскольку добавление каждой новой операции потребует вмешательства в существующий код, что приведет к засорению интерфейса Node.

Тогда мы применили шаблон VISITOR, Новые функциональные возможности реали-зовывались в отдельных объектах Visitor, и удавалось избежать необходимости вносить изменения в базовый класс Node, Основное свойство посетителей заключается в восстановлении информации о типе посещаемого объекта. Попробуем в данном случае определить класс посетителей под названием Presenter, который осуществляет все необходимое для представления заданного субъекта, включая его графическое представление, обработку вводов и т.д. Интерфейс этого класса может выглядеть следующим образом:

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

    virtual void visit{LoanSubject*);
    virtual void visit(AmountSubject*);
    virtual void visit(RateSubject*};
    virtual void visit(PeriodSubject*);
    //операции visit для других ConcreteSubjects

    virtual void init(Subject*);
    virtual void draw(Window*, Subject*);
    virtual void redraw(Window*, Subject*);
    virtual void handle(Event*, Subject*);
     //другие операции, выполняемые при обходе
};
```

Применение шаблона VISITOR требует добавить операцию accept дли всех типов объектов, которые можно посещать. Все эти операции реализуются одинаково. Например,

```
void LoanSubject::accept (Presenter& p) {
    p.visit(this);
}
```

Чтобы создать представление заданного субъекта, на каждой стадии обхода вызывается subject->accept (p), где subject типа Subject*, а p — экземпляр Presenter. В этом и заключается преимущество шаблона VISITOR: обратное обращение к данному экземпляру статически разрешается в корректную специфическую для подкласса операцию класса Presenter, фактически идентифицирующую конкретный субъект представления; таким образом, нет необходимости в динамической проверке типа.

Чтобы выяснить, какой класс отвечает за выполнение обхода, посмотрим еще раз на интерфейс класса Presenter. Помимо операций, предусмотренных шаблоном VISITOR, он содержит операции init, draw, redraw и handle. Эти операции осуществляют один или несколько обходов в ответ на стимулирующие воздействия, такие как ввод пользователя, изменение состояния субъекта или другие обстоятельства, описанные выше, и предоставляют клиентам простой интерфейс, позволяющий под-держивать жизнеспособность и "свежесть" представления. На рис. 3.6 графически представлен процесс обхода. По сравнению с рис. 3.4 количество объектов и связей (пунктирные линии) существенно уменьшилось.

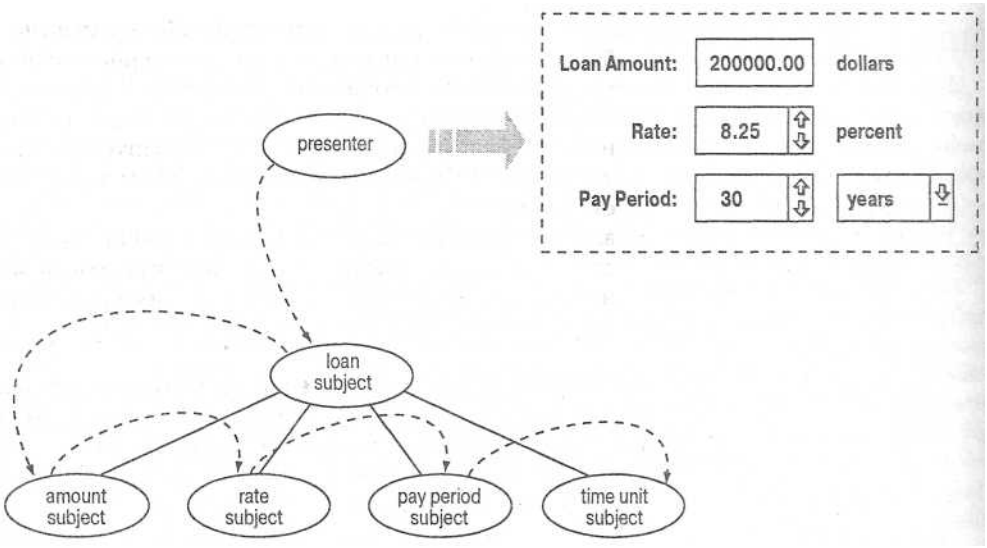


РИС. 3.6. *Обход, совершаемый "представителем"*

Как известно, шаблон VISITOR плохо работает, когда структура посещаемого класса нестабильна, и это предвещает возникновение проблем в нашем каркасе бизнес-приложения. В принципе, набор подклассов Subject является достаточно всеобъемлющим, чтобы программистам не нужно было определять своп собственные подклассы. Однако все-таки необходимо предусмотреть возможность определять презентации новых подклассов Subject, не изменяя существующий каркас. В частности, не хотелось бы добавлять новые операции visit к классу Presenter для поддержки новых подклассов Subject.

В главе 2 в интерфейсе класса Visitor была определена операция-уловитель visit, которая реализовала поведение по умолчанию. Сделаем то же самое и добавим операцию

```
virtual  void visit(Subject*) ;
```

к интерфейсу класса Presenter. Если есть некое стандартное поведение, которое должны реализовать все операции visit, можно поместить его в операцию visit (Subject*) и сделать так, чтобы остальные операции visit вызывали ее по умолчанию. Таким образом удается избежать дублирования стандартных функциональных возможностей.

Операция-уловитель предлагает нечто большее, чем обычное повторное использование кода. Она обеспечивает надежный вход, с помощью которого можно осуществлять посещения заранее не предусмотренных подклассов Subject.

Предлагаемый сценарий призван прояснить, почему так важна операция-уловитель visit. Если прикладной программист определяет новый подкласс Rebate Subject класса Subject, то операция accept реализуется в новом подклассе так же, как и во всех остальных подклассах:

```
void RebateSubject::accept (Presenter& p) {
   p.visit(this);
}
```

Когда RebateSubject::accept вызывает visit, передавая себя в качестве аргумента, компилятор должен найти соответствующую операцию в интерфейсе объекта Presenter. Если в качестве всеобщего уловителя операции Presenter:: visit (Subject*) нет, компилятор выдаст сообщение об ошибке. Если же всеобщий уловитель есть, тогда ситуация иная. Компилятор достаточно сообразителен, чтобы знать, что объект RebateSubject является также объектом Subject, и все в порядке, пока обеспечена совместимость типов.

Однако, удовлетворив требования компилятора, мы не многого добились. Операция Presenter::visit (Subject*) была реализована до того, как был создан класс RebateSubject. Это означает, что она не может делать ничего, помимо реализуемого ею стандартного поведения, что, возможно, вовсе не является поведением.

Что же дальше?

Мы старались избежать необходимости изменять интерфейс класса посетителей (в данном случае это класс Presenter), потому что прикладной программист не может изменять интерфейс, определенный каркасом. Однако ничто не мешает прикладному программисту порождать подклассы Presenter. Именно таким образом можно добавить код для представления объектов подкласса Rebate Sub j ect.

Определим подкласс NewPresenter. Помимо унаследованных функциональных возможностей класса Presenter добавим в него код, позволяющий создавать представление объектов Rebate Subject посредством замещения операции-уловителя[3]

```
void NewPresenter::visit (Subject* s) {
    RebateSubject* rs = dynamic_cast<RebateSubject*> (s) ;

    if (rs) {
      // представить RebateSubject
    } else {
      Presenter::visit(s); // реализовать стандартное поведение
    }
}
```

Неприятный сюрприз данного подхода — динамическая проверка типа, чтобы удостовериться, что посещаемый субъект действительно является объектом RebateSubject. Если мы абсолютно уверены, что NewPresenter::visit (Subject*) может вызываться только путем посещения Rebate Subject, тогда можно заменить динамическую проверку статической, но это достаточно рискованно. Если же нужно организовать посещение и представление нескольких новых подклассов Subject, динамическая проверка *необходима.*

Очевидно, что все эти действия направлены на преодоление присущего шаблону VISITOR недостатка. Если приходится постоянно добавлять новые подклассы Subject, то весь основанный на применении шаблона VISITOR подход превращается в стиль программирования, когда рассматриваются бесконечные частные случаи. Однако если в приложении определяется сравнительно немного новых подклассов (как и должно быть в случае, когда в проекте отдается предпочтение композиции), то преимущества от использования шаблона VISITOR налицо. Может возникнуть вопрос, почему динамическая проверка типов помещена в операцию visit, а не в RebateSubject::accept, например, следующим образом:

Особенность языка C++: *поскольку имя операции visit перегружено, необходимо выполнить замещение всех этих операций в классе NewPresenter, чтобы избежать возражений со стороны компилятора. Можно отказаться от перегрузки и указать конкретное имя субъекта в операции visit или использовать using, как предложил Келвин Хенней (Kelvin Henney). Эта проблема обсуждалась в главе 2.*

```
void RebateSubject::accept (Presenter& p) {
    NewPresenter* np = dynamic_cast<NewPresenter*>(&p) ;

    if (np) {
        np->visit(this);

    } else {
        Subject::accept(p); //реализует поведение по умолчанию
    }
}
```

В краткосрочной перспективе оба подхода работают одинаково хорошо, но второй подход сложнее поддерживать. С течением времени может оказаться, что новых подклассов Subject накопилось слишком много. В таком случае нужно быть готовым изменить класс Presenter, чтобы добавить операции visit для пока еще не поддерживаемых подклассов, таких как RebateSublect. После такого изменения интерфейс класса Presenter вновь будет содержать все классы, которые могут посещать его объекты.

Посмотрим, как выполняется данное преобразование в рассматриваемых случаях. Если приведения типов помещены в операции visit новых подклассов Concrete-Visitor, придется изменить класс Presenter и удалить приведения типов из таких операций, как NewPresenter::visit (Subject*). Однако изменения коснутся только одной иерархии классов и не затронут иерархию классов Subject, поскольку все операции accept хороши в том виде, какие они есть. Например,

```
void RebateSubject::accept (Presenter& p) {
    p.visit(this);
}
```

по прежнему компилируется успешно, однако теперь вызов visit разрешается статически в только что добавленный Presenter::visit (RebateSubject *)[4].

Если же поместить приведения типов в операции accept, изменений получится больше: придется менять все операции accept, а это однозначно нежелательно, поскольку главной мотивацией при применении шаблона VISITOR является желание избежать изменений иерархии посещаемых элементов.

Итак, не следует применять шаблон VISITOR в тех случаях, когда приходится постоянно определять новые подклассы класса Element. Однако даже когда определение подклассов не предполагается, создание новых подклассов должно быть *допустимым,* если нет веских оснований для противного. Следовательно, уловитель необходим. Нужно поместить в его определение проверки типов и выполнить перегрузку операций visit (если это возможно).

В основанной на применении шаблона VISITOR альтернативе OBSERVER все еще остались определенные проблемы. Первая проблема касается размеров класса Presenter. В один посетитель помещаются функциональные возможности нескольких классов

---

[4] *Заметьте, что если имя visit не перегружено, т.е. если тип посещаемого субъекта входит в имя операции visit, то приходится изменять и accept. Это происходит потому, что операция accept каждого нового подкласса класса Subject явно обращается к операции-уловителю:*

```
void RebateSubject::accept (Presenter& p) {
    p.visitSubject(this);
}
```

Поэтому следует использовать перегрузку, если язык ее поддерживает.

ConcreteObserver. Чтобы в результате не получился огромный монолит, в некоторый момент придется производить декомпозицию Presenter, возможно, применяя для уменьшения размеров посетителей другие шаблоны. Например, можно использовать шаблон STRATEGY, что позволит операциям visit передавать свои обязанности объектам Strategy. Однако одной из причин применения шаблона VISITOR было стремление уменьшить количество используемых объектов. Делая посетитель составным, мы уменьшаем преимущества от применения шаблона. Тем не менее, вряд ли в результате получится также много объектов и ссылок, как требуется при применении OBSERVER.

Вторая проблема касается состояний наблюдателей. Номинально основанный на применении шаблона VISITOR подход заменяет множество наблюдателей одним посетителем. Изначально предполагалось, что можно вычислять состояние наблюдателя вместо того, чтобы хранить его; но если каждый наблюдатель хранит свое собственное уникальное состояние и не все эти состояния можно вычислять в процессе выполнения, возникает вопрос: как быть с этими состояниями? Посетитель может хранить невычис-ляемое состояние, которое меняется от объекта к объекту, в собственной ассоциативной памяти (хэш-таблице), ключом которой выступает субъект. Однако затраты на реализацию хэш-таблицы и шаблона OBSERVER практически одинаковы.

Возможно, все это выглядит несколько надуманно. Прелесть "бумажных" проектов в том и состоит, что их не нужно компилировать, выполнять и в конце концов подавать в готовом виде. Но если здесь есть хоть какая-то полезная идея, воспользуйтесь ею!

Шаблон Generation Gap

Меня очень часто спрашивают: "Когда же вы опубликуете второй том шаблонов?" В *Design Patterns* действительно говорится, что ряд шаблонов не вошел в данную книгу, поскольку они показались недостаточно важными. За несколько лет мы законсервировали по крайней мере с полдюжины шаблонов по причинам "не кажется достаточно важным" и "нет достаточно известных применений".

Среди тех, которые были отложены из-за недостатка известных применений, оказался и шаблон GENERATION GAP. Тем не менее, я вскоре опубликовал данный шаблон [Vlissides96], отметил этот его недостаток и попросил читателей найти дополнительные примеры его применения. Я получил несколько ответов, которые включил в новое описание этого шаблона.

Название шаблона

Generation Gap

Классификация

Структурный

Назначение

Позволяет модифицировать или дополнить сгенерированный код и не утратить эти модификации при повторной генерации данного кода.

Мотивация

Заставить компьютер генерировать код обычно предпочтительней, чем писать его самому, при условии, что предложенный компьютером код является

- корректным;
- достаточно эффективным;
- функционально полным;
- обслуживаемым.

Многие средства автоматической генерации (такие как автоматические разработчики интерфейса пользователя, генераторы парсеров, разнообразные "мастеры" и 4GL-компиляторы) без проблем генерируют корректный и эффективный код. Некоторые программисты даже предпочитают изучать сгенерированный компьютером код вместо объемной документации при ознакомлении с новым программным интерфейсом. Но создание *функционально полного и обслуживаемого* кода — совсем другое дело.

Как правило, невозможно автоматически создать законченное нетривиальное приложение исключительно с помощью одного из перечисленных выше средств; некоторые функциональные возможности приходится реализовывать по старинке, т.е. вручную с помощью языка программирования. Это происходит потому, что высокоуровневые метафоры, которые делают автоматические средства столь мощными, редко бывают достаточно выразительны, чтобы описать каждую деталь. Абстракции автоматического средства неизбежно отличаются от абстракций языка программирования. Полученная в результате смешанная метафора может отрицательно повлиять на возможности расширения и поддержки приложения.

Рассмотрим построитель интерфейса пользователя, который позволяет собирать не только элементы интерфейса пользователя, такие как кнопки, полосы прокрутки и меню (так называемые "пользовательские средства управления"), но и более примитивные графические объекты: линии, окружности, многоугольники и текст[3]. Построитель интерфейса позволяет строить с помощью этих объектов изображения и затем связывать с ними некое поведение. Следовательно, с его помощью можно лучше описать приложение, чем в случае, когда предлагаются только "средства управления".

Данный построитель можно использовать для создания полного интерфейса пользователя для приложения под названием "будильник". Изобразим циферблат, собрав его из линий, многоугольников и текста, как показано на рис. 3.7. Затем добавим кнопки для задания текущего времени, времени подачи сигнала и для выключения сигнала. Построитель позволяет скомпоновать все эти элементы в полный интерфейс пользователя для данного приложения.

Однако построитель не позволяет указать, как эти элементы ведут себя при выполнении приложения. В частности, необходимо программировать поведение кнопок и стрелок. Самое основное — нужно иметь возможность ссылаться на эти объекты в своем коде. Построитель позволяет отбирать объекты "на экспорт", т.е. можно давать им имена и потом ссылаться на них при программировании. На рис. 3.8 пользователь выбирает для экспортирования секундную стрелку, экземпляр класса Line. Построитель выводит диалоговое окно (рис. 3.9), в котором пользователь может набрать экспортируемое имя данного экземпляра Line, например _sec_hand.

---

[5] *Примером такого построителя является* ibuild — *составная часть пакета InterViews[VT91].*

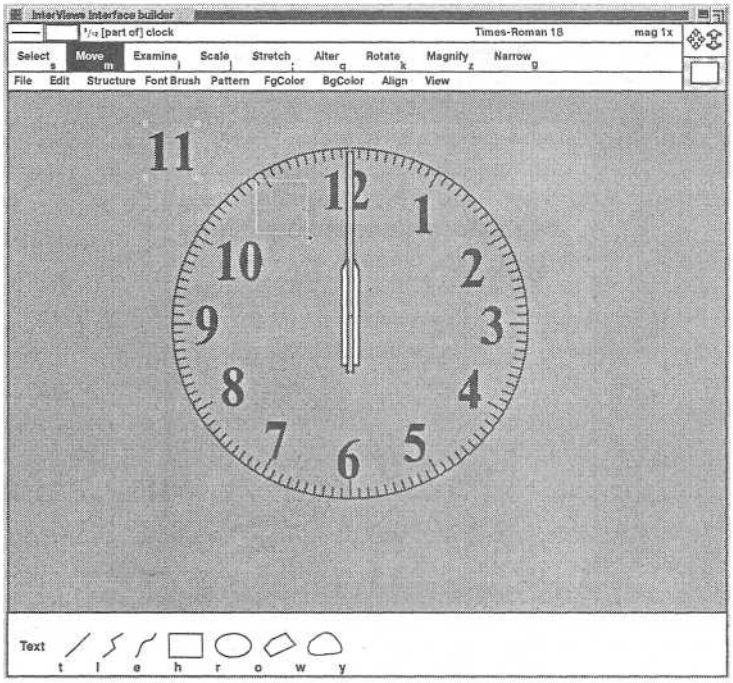


РИС. 3.7. *Построение циферблата*

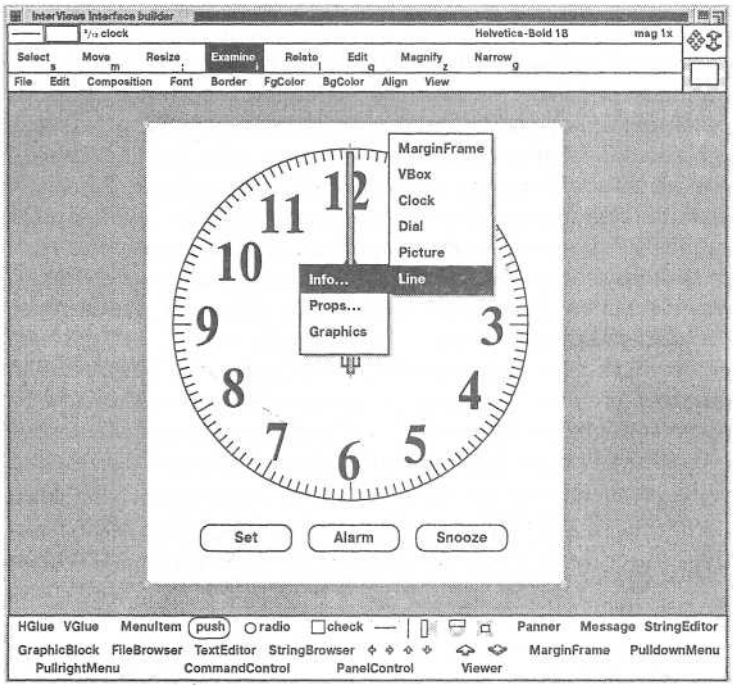


РИС. 3.8. *Отбор секундной стрелки для экспортирования*

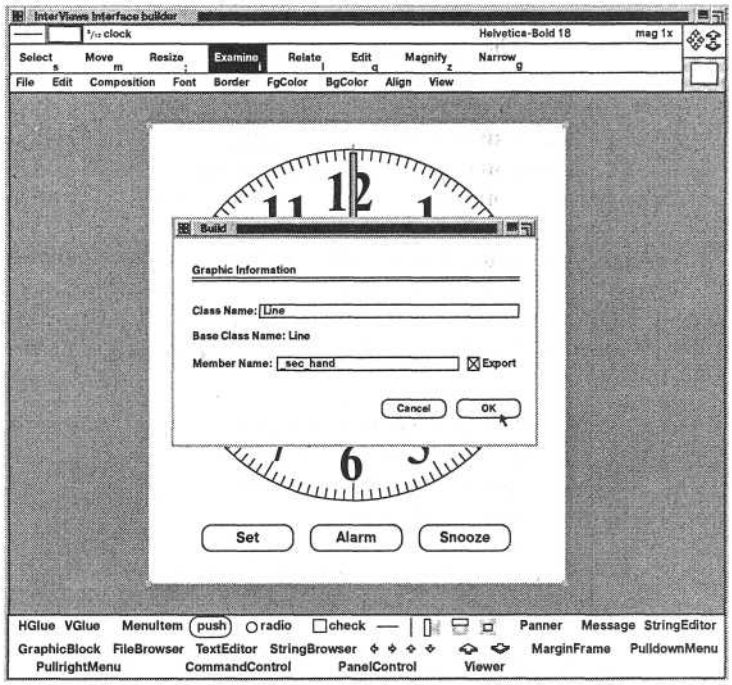


РИС. 3.9. *Именование и экспортирование секундной стрелки*

Построитель может создавать другие элементы пользовательского интерфейса часов, предлагая диалоговые окна, позволяющие пользователю задавать текущее время и время подачи сигнала. После окончания задания интерфейса построитель генерирует код, который собирает все графические компоненты и располагает их так, как было указано. Он также собирает диалоговые окна и реализует стандартные действия для всех кнопок, но на этом его возможности практически заканчиваются, как и возможности большинства других автоматических разработчиков приложений. Вам придется вернуться к самостоятельному программированию, чтобы указать, что конкретно делают кнопки при нажатии, как часы отслеживают время и как они поддерживают соответствующее внешнее представление. Большую часть поведения приходится программировать вручную. Как это делается?

Наиболее простой подход состоит в том, чтобы взять сгенерированный построителем код и изменить его в соответствии со своими нуждами. Например, можно добавить код, который генерирует некое событие каждую секунду. Можно написать соответствующий обработчик событий, который каждую секунду будет поворачивать секундную, минутную и часовую стрелки на определенный угол (минус 6° для секундной стрелки)[6]. Другой новый фрагмент кода будет реализовывать поведение кнопок. Изменения вносятся в код до тех пор, пока не получится работающее приложение.

---

[6] *Предполагается, что линии и другие графические объекты имеют интерфейс, позволяющий поворачивать их на заданный угол.*

Рассмотрим теперь проблему поддержки. Предположим, возникла идея реорганизовать пользовательский интерфейс с тем, чтобы кнопки оказались *над* циферблатом, а не *под* ним. Другими словами, нужно изменить только внешнее представление, а не поведение. Достоинство любого построителя интерфейсов состоит в том, что он позволяет очень легко вносить такие косметические изменения. Но построитель, к сожалению, ничего не знает о произведенных модификациях ранее сгенерированного им кода. Генерируя новый код, он или просто сотрет внесенные вручную изменения, или заставит вносить их повторно.

Существует несколько подходов к решению этой проблемы. Например, построитель может отмечать в генерированном коде, допускает ли фрагмент модификации пользователя или нет (в таком случае фрагмент снабжается предупреждениями о недопустимости модификаций). Однако подобный подход неприемлем, по крайней мере, по двум причинам.

1. *Беспорядочность.* Хотя это усовершенствование препятствует незапланированным модификациям, код, написанный вручную, по-прежнему смешивается с кодом, сгенерированным автоматическим разработчиком. Результат выглядит дотакой степени беспорядочно, что могут понадобиться специальные средства, чтобы сделать его более читабельным (например, с помощью сокрытия или выделения по требованию отдельных фрагментов), но, как правило, подобные средства не в состоянии полностью ликвидировать эту проблему.
2. *Высокая вероятность ошибок.* Поскольку модификации проводятся исключительно по соглашению, компилятор не в состоянии отслеживать недопустимые изменения. Если по ошибке модифицируется не тот код, генератор впоследствии может просто отбросить эти изменения.

Более сложный подход состоит в том, чтобы находить различия между модифицированным и изначально сгенерированным кодом, а затем пытаться внести эти изменения в повторно генерируемый код. Нужно ли говорить, что это рискованное предложение, если производимых вручную модификаций много или они достаточно нетривиальны.

Идеальное решение должно быть более надежным, а чтобы обеспечить возможность сопровождения, необходимо отделить сгенерированный код от производимых вручную модификаций. Однако добиться строгого разделения достаточно сложно, поскольку модификации часто требуют получения доступа к тем фрагментам сгенерированного кода, которые не должны быть открытыми. Например, доступ к объекту Line, представляющему секундную стрелку, не должен разрешаться извне часов, поскольку данный объект — реализационный артефакт. Даже более высокоуровневый интерфейс для поворота секундной стрелки не следует делать открытым — в обычных часах такая возможность, как правило, отсутствует.

Шаблон GENERATION GAP позволяет решить данную проблему с помощью механизма наследования. Сгенерированный код инкапсулируется в базовом классе, а модификации — в соответствующих подклассах.

В нашем примере с часами автоматический построитель инкапсулирует генерируемый им код в сердцевинном классе (core class) под названием Clock_core. Весь код, с помощью которого построитель реализует часы, — включая графические объекты, пользовательские средства управления и способ их компоновки — находится в этом классе. Никто никогда не создает экземпляры сердцевинного класса. Создаются экземпляры класса Clock, общее название которого класс расширений (extension class). Построитель генерирует класс расширений одновременно с сердцевинным классом.

Как видно из названия, класс Clock является подклассом Clock_core. Но это именно тривиальный подкласс: он не может ничего добавлять или удалять, не может модифицировать состояние или поведение сердцевинного класса. Он делает только то, что делает его суперкласс. Тем не менее, код, создающий объекты Clock, *всегда* реализует класс Clock, класс расширений, а не сердцевинный класс Clock_core.

Где же производятся модификации? Можно модифицировать сердцевинный класс, чтобы он работал с другим кодом приложения, но последующее редактирование в среде построителя и выполнение повторной генерации приведет к уже описанной проблеме смешивания кода. Поэтому при добавлении, изменении и удалении функциональных возможностей необходимо модифицировать класс *расширений,* а не сердцевинный класс. Можно определить новые функции-члены, переопределить или расширить виртуальные функции сердцевинного класса (рис. 3.10). Если экспортируемые экземпляры объявлены как защищенные переменные-члены в C++, класс расширений может осуществлять доступ к ним, не показывая их клиентам.

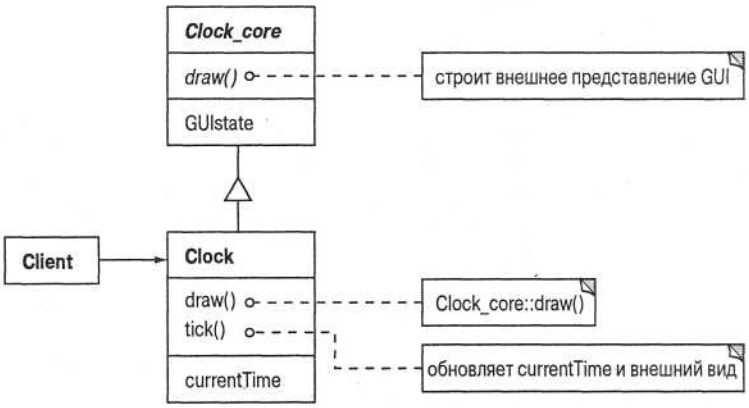


РИС. 3.10. *Расширение сердцевинного класса*

Если в будущем понадобится модифицировать внешний вид интерфейса, построитель может повторно генерировать *только* сердцевинный класс, который не подвергался модификации, таким образом изменения, внесенные в класс расширений, не будут затронуты. Затем производится повторная компиляция приложения, которая отразит изменение внешнего представления. Только радикальные изменения интерфейса (например, удаление секундной стрелки или других экземпляров, которых касались внесенные изменения) могут привести к необходимости отмены произведенных изменений.

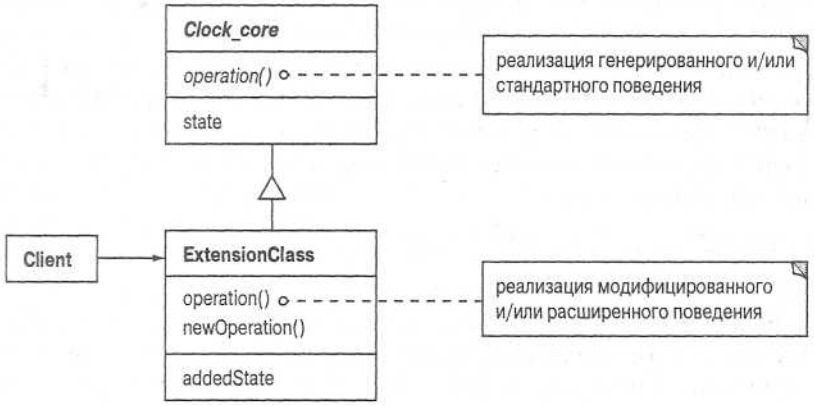
Применимость

Шаблон GENERATION GAP следует применять, когда выполнены *все* приведенные ниже условия.

- Код генерируется автоматически.
- Сгенерированный код можно инкапсулировать в одном или нескольких классах.
- Повторно генерируемый код, как правило, сохраняет интерфейс и переменные экземпляров предыдущей редакции.

Сгенерированные классы, как правило, не интегрированы в существующие иерархии классов. Если же это не так, и используемый язык программирования не поддерживает множественное наследование интерфейса, то генератор кода должен позволять вам определять суперкласс для любого базового класса, который он генерирует.

Структура



Участники

CoreClass (Clock_core)

- Абстрактный класс, содержащий сгенерированную автоматическим средством реализацию.
- Никогда не модифицируется вручную.
- Переписывается автоматическим средством при повторной генерации.

ExtensionClass (Clock)

- Тривиальный подкласс класса CoreClass.
- Реализует расширения или модификации класса CoreClass. Программист может изменять класс ExtensionClass, чтобы добавить состояние и/или расширить, модифицировать или заместить поведение CoreClass.
- Позволяет сохранить расширения и модификации при повторных генерациях.

Клиент

- Создает экземпляры класса ExtensionClass и обращается только к этому классу.

Взаимодействия

- Класс ExtensionClass наследует автоматически генерированное поведение класса CoreClass, расширяя и замещая его при необходимости.

- Класс CoreClass демонстрирует и/или делегирует определенные функциональные возможности классу ExtensionClass, что позволяет модифицировать и дополнять его поведение.

Последствия

Шаблон GENERATION GAP обеспечивает следующие преимущества.

1. *Модификации отделены от автоматически генерированного кода.* Все производимые вручную модификации поступают в класс ExtensionClass, где они инкапсулируются и хранятся отдельно от генерированного кода. Кроме того, интерфейс класса ExtensionClass может предлагать дополнительную информацию о модификации, отмечая добавляемые или замещаемые операции.
2. *Модификации могут иметь привилегированный доступ к реализационным деталям.*
   Связь наследования между классами CoreClass и ExtensionClass означает, что программисты и автоматические программные средства могут использовать все выразительные возможности языка реализации, чтобы контролировать доступ к генерируемым элементам.
3. *Последующая повторная генерация не требует повторного внесения модификаций.* Автоматическое программное средство повторно генерирует только класс CoreClass, не затрагивая ExtensionClass, тем самым модификации сохраняются. Хотя модификации не нужно применять повторно, в них может понадобиться внести изменения, если истинно хотя бы одно из следующих утверждений.

   - Модификации касались членов, которые больше не существуют (пример "синтаксической несовместимости").
   - Повторно генерируемый код семантически отличается от предыдущей версии, так что операции имеют другой смысл ("семантическая несовместимость").

     Синтаксическую несовместимость обычно легче исправить, чем семантическую. Однако обе несовместимости снижают эффективность применения шаблона, поэтому они не должны быть характерной особенностью генерируемого кода.
4. *Классы CoreClass и ExtensionClass могут разрабатываться и тестироваться независимо.*
   Чем крупнее проект разработки программного обеспечения, тем больше вероятность вовлечения в него людей с узко специальными навыками. Эксперты в предметной области могут концентрировать свои усилия на анализе или моделировании, в то время как другие специалисты занимаются проектированием, реализацией, тестированием или разработкой документации. Необходимо гарантировать, что эти группы взаимодействуют эффективно, а результаты их работы интегрируются без проблем. Чем более независимы группы, тем больше возможностей для параллельной работы, но тем сложнее фаза интеграции.
   Шаблон GENERATION GAP предлагает бесконфликтное сотрудничество и интеграцию путем расщепления функциональности классов CoreClass и ExtensionClass. В примере, предложенном в разделе "Мотивация", специалист по интерфейсам пользователя может разрабатывать с помощью автоматического разработчика интерфейсов класс Clock_core, в то время как эксперт в предметной области занимается интеграцией класса Clock в структуру лежащего в основе

приложения. После того как разработчики согласовали интерфейс класса Clock_core, они могут работать независимо.

Данный шаблон также позволяет разработчикам тестировать сердцевинный класс и класс расширений по отдельности. До модификации класс Clock_core допускает создание экземпляров и может принести определенную пользу. Специалист по пользовательским интерфейсам может использовать результат работы автоматического генератора для того, чтобы оценить внешний вид, эргономичность, эффективность и другие аспекты интерфейса пользователя, которые класс Clock, по всей видимости, затрагивать не будет. Тем временем эксперт в предметной области может тестировать подсистемы, лежащие в основе пользовательского интерфейса, модифицируя класс Clock и экспериментируя с ним посредством программирования, т.е. не выполняя сборку и даже не реализуя пользовательские средства управления. Если стороны удовлетворены своими классами, осуществить их сборку будет достаточно просто.

Два основных недостатка шаблона заключаются в следующем.

1. *Удвоение числа классов.* Для каждого класса, который предполагалось использовать, шаблон создает пару классов CoreClass/ExtensionClass. Он может также создавать классы, которые ничего не содержат, в том случае, например, если генерируемый код чисто процедурный. Создание каждого дополнительного класса приводит к дополнительным издержкам. Даже если этот дополнительный класс не занимает память и не замедляет скорость выполнения программы, то уж точно требует дополнительной работы на концептуальном уровне.
2. *Может оказаться достаточно сложно интегрировать созданные классы в существующие иерархии классов.* Чтобы класс расширений наследовал существующий класс,необходимо множественное наследование. Этого же эффекта можно добиться, если сделать так, чтобы CoreClass наследовал существующий класс, но тогда по требуется изменить интерфейс класса CoreClass— тем самым теряется смысл применения данного шаблона. Эту проблему можно решить с помощью генератора кода, если пользователю будет разрешено указывать родительский класс для сердцевинного класса.

Реализация

Реализация шаблона GENERATION GAP в значительной мере зависит от среды и языка программирования. Рассмотрим следующие четыре вопроса.

1. *Запрещение модификаций сердцевинного класса.* Кардинальное требование шаблона GENERATION GAP заключается в том, что программисты никогда не модифицируют сердцевинный класс. К сожалению, гарантировать это достаточно сложно. Если язык и среда программирования основаны на файлах, наиболее безопасный способ предотвратить такие изменения заключается в том, чтобы поместить объявление класса и его реализацию в один или несколько защищенных от записи файлов. Однако эти файлы не должны быть защищены настолько, чтобы автоматическое средство было не в состоянии переписать их при повторном генерировании кода. Может понадобиться предоставить автоматическому средству специальные полномочия, независимые от полномочий его пользователей.

В программных средах, использующих некую разновидность базы данных для хранения программной информации, ситуация несколько проще. Такие среды обычно предлагают более избирательный контроль доступа к исходному коду программы, чем может обеспечить обычная файловая система.

2. *Контроль доступа к внутренним составляющим сердцевинного класса.* Как следует из раздела "Мотивация", класс расширений нуждается в доступе к внутренним элементам сердцевинного класса. Существование связи наследования между сердцевинным классом и классом расширений делает это достаточно простым, поскольку в большинстве объектно-ориентированных языков подкласс может осуществлять доступ практически ко всему, что он наследует от своего родительского класса.

   Однако следует помнить, что чем больше информации класс CoreClass поставляет своим подклассам, тем выше вероятность, что повторное генерирование кода повредит модификации, сделанные в классе ExtensionClass. Такие языки, как C++, Java и Eiffel предлагают несколько уровней контроля доступа для сокрытия информации от подклассов. Управляя доступом к внутренним элементам сердцевинного класса, автоматический генератор кода может минимизировать количество информации, получаемое классом расширений от его сердцевинного класса.
3. *Соглашения об именовании.* Поскольку шаблон предлагает разбить один класс на два, имена полученных классов должны отражать их происхождение и тесную взаимосвязь. В то же время клиенты ничего не знают о расщеплении. Поэтому за классом ExtensionClass должно остаться имя исходного класса, а класс CoreClass должен получить имя, производное от имени его подкласса, хотя обычно принято делать наоборот. В нашем примере, чтобы получить имя сердцевинного класса, к имени класса расширений Clock было добавлено "_core".
4. *Степень грануляции операций класса CoreClass.* Основное преимущество автоматической генерации кода заключается в том, что она достигает уровня, на котором пишутся программы. Зачастую для реализации высокоуровневой метафоры автоматический генератор должен создать значительное количество кода. В результате программист, который захочет модифицировать или дополнить сгенерированный код, столкнется со всеми его сложностями. Как же в таком случае менять этот код?

   Ключ к решению данной проблемы находится в интерфейсе класса CoreClass. Операции должны быть достаточно мелко гранулированы, чтобы программист смог переопределить необходимые функции и использовать без изменений все остальные. Если, например, в классе CoreClass вся функциональность реализована в виде одной большой операции, то программист не сможет внести даже минимальные изменения в его функции, не изменяя реализацию операции в целом. Если же операция разбита на несколько небольших, разумно выбранных элементарных операций (таких, как фабричные методы), составляющих шаблонный метод, это даст программисту больше возможностей модифицировать только необходимые функции.

Образец кода

Ниже приводится подлинная декларация класса Clock_core, сгенерированная автоматическим разработчиком графического интерфейса пользователя *ibuild* [VT91]:

```
class Clock_core : public MonoScene {
public&
    Clock_core (const char*) ;
protected:
    Interactor* Interactor();

    virtual void setTime();
    virtual void setAlarm();
    virtual void snooze();
protected:
    Picture* _clock;
    SF_Polygon* _hour_hand;
    SF_Rect* _min_hand;
    Line* _sec_hand;
};
```

MonoScene является разновидностью класса Decorator для представления пользовательских средств управления в пакете разработчика InterViews GUI [LVC89]. Класс Interactor — базовый класс пользовательских средств управления в пакете InterViews, таким образом, MonoScene — некий подкласс класса Interactor. Пакет InterViews также поставляет классы SF_Polygon, SF_Rect, Line и Picture, реализующие графические объекты. Класс Picture — это класс Composite шаблона COMPOSITE, а остальные классы ведут себя как классы Leaf данного шаблона. Пользователь *ibuild* экспортирует эти экземпляры, чтобы сделать их доступными для класса расширений.

Хотя на первый взгляд кажется, что класс Clock_core определяет совсем немного функций-членов, в действительности его интерфейс достаточно велик, в основном из-за больших размеров интерфейса класса Interactor. Класс Clock_core наследует значительную часть стандартного поведения как от класса Interactor, так и от класса MonoScene. Из собственных операций класса Clock_core только операция Interior действительно что-то делает: она собирает пользовательские средства управления и графические объекты (как экспортируемые, так и неэкспортируемые) и формирует пользовательский интерфейс. Операция Interior не является элементарной; и поскольку сборка описывается исключительно в построителе интерфейса, нет необходимости замещать ее — достаточно перестроить пользовательский интерфейс в построителе.

Теперь нужно добавить некое поведение с помощью программирования. Для этого следует модифицировать класс расширений. Перед модификацией он выглядит следующим образом.

```
class Clock : public Clock_core {
public:
    Clock(const char*);
}
```

Конструктор ничего не делает, но благодаря генерированному коду, который наследует класс Clock, можно создавать экземпляры этого класса, отображающие циферблат, хотя и без всякого поведения. Остается только заместить некоторые операции.

Операции setTime, setAlarm и snooze были заданы в построителе. Это операции, которые вызываются при нажатии соответствующих кнопок. Они ничего не делают по умолчанию; нужно переопределить их в классе Clock, чтобы они выполняли полезную работу. Кроме того, необходимо добавить код для вращения стрелок в ответ на события, генерируемые таймером Interviews, которые часы получают каждую секунду.

Итак, модификации, позволяющие сделать класс вполне развитым приложением, выглядят следующим образом.

```
class Clock : public Clock_core {
public:
    Clock(const char*);

    void run();

    virtual void setTime();
    virtual void setAlarm();
    virtual void snooze ();

    virtual void Update();
private:
    void getSystemTime(int& h, int& m, int& s);
    void setSystemTime(int h, int m, int s);
    void alarm();
private:
    float _time;
    float _alarm;
};
```

Модифицированный конструктор инициализирует _alarm (где хранится время подачи сигнала), и time (время последнего обновления), устанавливая их значения равными нулю. Операция run реализует цикл обработки событий: каждую секунду ожидается поступление события и обновляется внешний вид циферблата, чтобы отразить текущее время, сообщаемое getSystemTime.

Функция run представляет собой шаблонный метод, элементарными операциями которого являются операции alarm и Update. Операция alarm вызывается, когда должен сработать будильник, а операция Update (унаследованная от класса Interactor) вызывается для обновления показаний часов. Чтобы минимизировать перестройки интерфейса, операция Update задает количество поворотов каждой стрелки в зависимости от разности между текущим временем и временем последнего обновления. Таким образом, поворачиваются только те стрелки, которые должны изменить свое положение.

Операции setTime, setAlarm и snooze замещены с тем, чтобы они могли выполнять свои задачи. В частности, операции setTime и set Alarm должны предлагать диалоговые окна (созданные с помощью *ibuild),* чтобы получить данные от пользователя. Вспомогательные функции getSystemTime и setSystemTime инкапсулируют обращения к системе для извлечения и задания системного времени.

Известные применения

Первым известным применением шаблона GENERATION GAP является построитель интерфейса пользователя *ibuild* [VT91].

Когда я впервые описал данный шаблон [Vlissides96], на этом мне пришлось остановиться. Я также отметил тогда, что именно недостаток известных применений не позволил включить шаблон GENERATION GAP в книгу *Design Patterns,* и обратился к читателям с просьбой присылать примеры. Благодаря их откликам появилась возможность указать дополнительные известные применения. Дэвид Ван Кэмп (David Van Camp, [VanCamp96]) написал:

> "Прочитав описание вашего шаблона, я сразу вспомнил средство под названием Visual Programmer, которое предлагалось вместе с Symantec C++ 6.0 для Windows/DOS от компании Blue Sky Software. Мне всегда нравились средства компании Blue Sky, и это не стало исключением. Насколько я помню, Visual Programmer автоматически генерирует два множества исходных файлов — одно для генерируемого кода, а второе представляет собой множество пустых классов для модификаций, производимых по желанию пользователя (т.е. делает то же, что и GENERATION GAP). Руководство датировано 1993 годом, так что это было довольно давно. Я никогда не использовал на практике данное средство, но помню, что оно произвело на меня большое впечатление."

Дэвид также указал в качестве примера Forte Express, разработчик приложений для распределенных сред [Forte97]. В нем используется вариация шаблона GENERATION GAP, которая позволяет вносить фрагменты произвольного кода между кодом, генерируемым Express, и кодом библиотеки Forte.

Не следует думать, что данный шаблон подходит исключительно для построителей интерфейса. Крис Кларк (Chris Clark) и Барбара Зино (Barbara Zino) приводят в своем письме [CZ96] пример, который опровергает это мнение:

> "Мы использовали вариацию шаблона GENERATION GAP в Yacc++ и Language Objects Library. Как можно судить по его названию, данное автоматическое средство по описанию языка генерирует лексические анализаторы (лексеры) и синтаксические анализаторы (парсеры). Более важным является то, что к данному средству прилагается библиотека, содержащая каркас типичного компилятора. Автоматическое средство генерирует передаточные таблицы, которые должны присоединяться к соответствующему [лексическому и синтаксическому] анализатору *{замечательный пример применения шаблона* STRATEGY- *прим.авт.).* Следовательно, при генерировании указанного пользователем класса автоматическое средство генерирует класс, производный от соответствующего класса анализатора. Это расщепление аналогично описанному вами расщеплению в шаблоне GENERATION GAP. Реализационный код и переменные экземпляров находятся в библиотечном классе. Производный класс может замещать любую из функций-членов, которые необходимо модифицировать по желанию пользователя."

Некоторые читатели предложили рассматривать в качестве примера применения шаблона генерацию заглушки в архитектуре CORBA. Особенно подробно написал об этом Джеролф Уэндлэнд (Geroif Wendland, [Wendland97]), он даже привел в своем письме значительные фрагменты кода из книги Джона Сигела (Jon Siegel) *Corba Fundamentals and Programming* [Siegel96], которые здесь опущены.

> "В основанной на технологии CORBA системе Orbix применяется аналогичный шаблон (возможно, в точности такой же). Я посылаю вам фрагмент кода, взятый из книги.
> Предположим, что используется совместимый с CORBA BOA-подход. Рассмотрим в качестве примера интерфейс StoreAccess. Он компилируется IDL-компилятором, который, в свою очередь, генерирует класс StoreAccessBOAImpl. Этот класс содержит все функциональные возможности, необходимые для кооперации с работающей системой Orbix, а также операции, которые были ранее указаны в IDL-описании. Чтобы задать свои собственные операции, необходимо создать подкласс класса StoreAccessBOAImpl и переопределить операции, заданные в IDL-описании. Программист может самостоятельно выбирать имя для данного подкласса; в Orbix предлагается именовать реализацию StoreAccess_i. IDL-компилятор системы Orbix обеспечивает средства создания кодовых структур для генерируемых объектов StoreAccess_i. После того как эти структуры использовались и дополнялись, их невозможно генерировать повторно, не потеряв всех изменений. Однако существует возможность создать файлы, содержащие определение и реализацию класса StoreAccessBOAImpl, и регенерировать их сколь угодно часто. Предложенная вами схема именования отображается в схему, используемую в Orbix, следующим образом:
> StoreAccessBOAImpl <=> StoreAccess_core
> StoreAccessBOAImpl_i <=> StoreAccess"

Благодаря этим и другим откликам я пришел к выводу, что теперь шаблон GENERATION GAP, наконец, набрал критическую массу известных применений. Остался последний раздел, и описание шаблона можно считать законченным.

Родственные шаблоны

В сердцевинном классе часто используются шаблонные методы, чтобы генерируемый код был достаточно гибким и его легко было использовать повторно. Фабричные методы могут позволить классам расширений контролировать объекты (типа Strategies), которые используются внутри сердцевинного класса.

Я всегда с особым чувством относился к шаблону GENERATION GAP, даже тогда, когда он еще не был полноправным шаблоном. С его помощью удалось преобразовать *ibuild* из средства, генерирующего высокофункциональный, но трудно поддерживаемый код, пригодный только в качестве первого приближения интерфейса пользователя, в настоящее работающее средство создания приложений, которое поддерживает не только GUI, но и определенное поведение. Мы просто не знали о других применениях данного шаблона, но я не удивился, узнав, что они есть.

Если вы занимаетесь генерированием кода в любой объектно-ориентированной области, шаблон GENERATION GAP вам пригодится. Я прошу вас сообщить мне, как вы его использовали, особенно если ваш способ отличается от описанных в этой книге, поскольку указать много известных применений всегда непросто.

Стирание типов

Когда мы пытались восстановить потерянную информацию о типе, то столкнулись с определенными трудностями: приходилось снова и снова использовать dynamic_cast, но у нас не было выбора, поскольку используемый каркас ничего не знал о тех дополнениях, которые мы внесли в его интерфейсы. Возникновение проблем такого рода— верный признак дефекта проектирования. Положительный момент состоит в том, что подобный дефект иногда можно превратить в достаточно полезную особенность проекта.

Представим себе каркас системы управления в реальном времени, который определяет абстрактный базовый класс Event. Базирующееся на этом каркасе приложение использует подклассы класса Event для моделирования конкретных событий. Различным приложениям будут нужны разные типы событий (события, генерируемые крылатой ракетой, заметно отличаются от событий генерируемых торговым автоматом).

При таком разнообразии специфических для области применения событий разработчик каркаса не должен даже пытаться создать полный исчерпывающий интерфейс класса Event. Вместо этого в классе Event определяется всего несколько операций, имеющих смысл для всех типов событий:

```
virtual long timestamp() = 0;
virtual const char* rep () = 0;
```

Операция time stamp определяет точное время возникновения события, а rep возвращает низкоуровневое представление этого события, например, пакет байтов, полученный от сети или контролируемого прибора. Именно в подклассах необходимо определить и реализовать более конкретные, предназначенные для определенных приложений операции.

Рассмотрим торговый автомат. Его подкласс CoinInsertedEvent добавляет операцию Cents getCoin(), которая возвращает сумму брошенных покупателем монет. Другой вид событий CoinReleaseEvent реализуется тогда, когда покупатель хочет получить обратно свои монеты. Операция getCoin и другие аналогичные операции реализуются с использованием rep. Клиенты таких событий могли бы непосредственно использовать операцию rep, если объявить ее открытой. Но в этом мало смысла, поскольку операция rep практически не предлагает абстракций и клиентам чрезвычайно сложно извлечь необходимую информацию. Поэтому более вероятно, что операция rep будет объявлена защищенной, и доступ к ней будут иметь только подклассы, где она используется для реализации интерфейсов более высокого уровня.

Однако существует проблема, которая является следствием невозможности определить для событий универсальный интерфейс. Очевидно, что каркас ничего не знает и не может знать о зависящих от области применения подклассах Event, ведь прикладные программисты определяют их уже после того, как каркас был спроектирован, разработан и помещен на CD-ROM. О событиях каркасу известно только то, что они реализуют минимальный основополагающий интерфейс, состоящий из операций timestamp и rep.

В этой связи возникает два вопроса.

1. Как каркас создает экземпляры, зависящие от области применения подклассов?
2. Каким образом код приложения получает доступ к собственным операциям подклассов, если все, что оно получает от каркаса — это объекты типа Event?

Ответ на первый вопрос можно найти в любом из нескольких порождающих шаблонов, описанных в книге *Design Patterns*. Например, в каркасе можно определить фабричные методы (шаблон FACTORY METHOD), которые возвращают экземпляры зависящих от области применения подклассов Event. Когда каркасу нужен новый экземпляр, он использует фабричный метод вместо вызова new. Приложение замещает эти фабричные методы с тем, чтобы они возвращали конкретные, зависящие от области применения, экземпляры.

Чтобы не создавать подклассы, единственная задача которых возвращать специфические для области применения события, можно использовать шаблон PROTOTYPE. Он предлагает композиционную альтернативу шаблону FACTORY METHOD. Если к базовому классу Event добавить операцию

```
virtual Event* copy ( ),
```

код каркаса может использовать события для создания копий этих событий. Тогда вместо операции

```
Event* e = new CoinReleaseEvent;
```

которую каркас, возможно, выполнить не в состоянии, поскольку здесь содержится ссылка на зависящий от области применения подкласс, можно написать

```
Event* e = prototype->copy( );
```

где prototype ссылается на экземпляр известного каркасу типа, а именно Event. Поскольку copy — полиморфная операция, e может быть экземпляром любого подкласса класса Event. Средство реализации каркаса должно только убедиться, что prototype перед использованием был инициализирован (в данном случае — неким экземпляром соответствующего подкласса Event). Приложение может сделать это во время фазы инициализации или в любое удобное время до того, как каркас вызывает

```
prototype->copy().
```

Теперь перейдем ко второму вопросу: есть ли шаблоны, позволяющие восстановить информацию о типе экземпляра? Точнее, если каркас предлагает операции вида

```
virtual Event* nextEvent();
```

откуда приложение узнает, какой тип события возвратит nextEvent, чтобы правильно вызвать операции конкретного подкласса?

Как всегда, есть грубый подход:

```
Event* e = nextEvent();
CoinInsertedEvent* ie;
CoinReleaseEvent* re;
//аналогичные декларации других видов событий

if (ie = dynamic_cast<CoinInsertedEvent*>(e)) {
    //вызвать операции подкласса CoinlnsertedEvent для ie
} else if (re = dynamic_cast<CoinReleaseEvent*>(e) ) {
    //вызвать операции подкласса CoinReleaseEvent для re
} else if(...) {
    //и т.д.
}
```

Крайне неприятно, если такие проверки типов приходится делать всякий раз, когда приложение обрабатывает полученное от каркаса событие. Однако неприятностей станет еще больше, если мы определим новый подкласс Event. Должен существовать более удачный способ!

Стандартный метод восстановления утраченной информации о типе без использования динамических проверок заключается в применении шаблона VISITOR. В данном случае сначала нужно добавить к базовому классу Event операцию void accept (EventVisitor&), где EventVisitor — базовый класс объектов, которые могут посещать объекты Event. Поскольку каркас определяет класс Event, он должен также определять класс EventVisitor, и здесь мы вновь сталкиваемся с проблемой: каким должен быть интерфейс EventVisitor?

Как известно, интерфейс абстрактного класса Visitor должен определять операции visit для каждого типа объектов, которые может посещать посетитель. Но что делать, если тип этих объектов неизвестен каркасу? Посетитель событий торгового автомата нуждается в операциях следующего вида:

```
virtual  void visit(CoinInsertedEvent*);
virtual  void visit(CoinReleaseEvent*);
//и  другие  подобные  операции  для  каждого  события
```

Такие операции невозможно определить в классе каркаса, каким является EventVisitor. Похоже, что в данном случае даже шаблон VISITOR не спасет нас от надоевшей операции dynamic_cast. Жаль.

Несколько позже будет предложено радикальное решение проблем, возникших с Event. Однако бывают случаи, когда вместо того, чтобы оплакивать потерю информации о типах, нужно умело использовать сложившуюся ситуацию. Забудем временно о классе Event и рассмотрим на первый взгляд не связанный с нашей проблемой шаблон MEMENTO.

Назначение шаблона MEMENTO состоит в том, чтобы зафиксировать и вывести за пределы объекта его внутреннее состояние, таким образом позднее можно будет восстановить объект в этом состоянии. На первый взгляд достаточно просто, но следует упомянуть об очень важном ограничении: все это нужно сделать, *не нарушая инкапсуляцию объекта.* Иными словами, внутреннее состояние объекта должно быть *доступно,* но не *видимо* другим объектам. Неужели такое возможно?

Простой пример поможет показать, что противоречия здесь нет. В разделе "Реализация" шаблона ITERATOR (см. *Design Patterns)* описан *cursor* (курсор) — некий итератор, который просто указывает текущую позицию при обходе. Во время обхода обходимая структура "передвигает" курсор, чтобы он указывал на следующий элемент, подлежащий рассмотрению. Структура может также по запросу клиента "разыменовывать" курсор (т.е. возвращать элемент, на который он указывает), например, таким образом:

```
Structure  s;
Cursor  c;

for  (s.first(c);  s.more(c);  s.next(c))  {
     Element  e  =  s.element(c);
     //использование  Element  e
}
```

Курсор не содержит операций, доступных клиентам. Только обходимая структура может получать доступ к внутреннему содержимому курсора. Структура имеет исключительные привилегии, поскольку содержащаяся в курсоре информация является со ставной частью внутреннего

состояния структуры и должна оставаться инкапсулированной. Следовательно, в шаблоне MEMENTO курсор — это некий объект-хранитель (Memento), структура является создателем (Originator), а клиент выступает в роли смотрителя (CareTaker).

Чтобы, не нарушив инкапсуляцию, вывести за пределы объекта его внутреннее состояние, необходимо реализовать так называемый двухсторонний (two-faced) объект. Структура видит обширный интерфейс, который разрешает доступ к информации о состоянии. Другие клиенты видят суженный или даже вовсе несуществующий интерфейс Memento; предоставление клиентам доступа к любому состоянию внутри объекта Memento означало бы нарушение инкапсуляции структуры. Но как создать объект с двумя различными интерфейсами в C++?

В шаблоне MEMENTO предлагается использовать служебное слово friend. Класс Originator является дружественным Memento, поэтому ему разрешен доступ к широкому интерфейсу, запрещенный другим классам.

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

private:
    friend class Structure;

    Cursor () {_current = 0;}

    ListElem* getCurrent () const {return _current;}
    void setCurrent (ListElem* e) {_current = e;}

private:
    ListElem* _current;
};
```

В данном сценарии Cursor хранит только некий указатель. Он указывает на ListElem— класс, который используется внутри Structure для представления узлов двунаправленного списка. Объекты ListElem наряду с указателем на некий объект Element поддерживают указатели на предшествующий и последующий элементы списка. Операции класса Structure манипулируют _current, чтобы отслеживать позицию при обходе:

```
class Structure {
public:
    // …
    virtual void first (Cursor& c) {
        c.setCurrent(_head);
            //_head — это голова связанного списка,
            //поддерживаемого внутри Structure
    }
    virtual bool more (const Cursor& c) {
        return c.getCurrent ()-> next;
    }
    virtual void next (Cursor& c) {
        c.setCurrent(c.getCurrent()->_next);
        //делает текущим следующий ListElem*
    }
```

```
        virtual Elements element (const Cursor& c)  {
            return *c.getCurrent()->_element;
        }
        // …
};
```

В результате шаблон MEMENTO позволяет структуре вывести за ее пределы и поместить в хранитель Cursor достаточно тонкую информацию, чтобы отметить текущее состояние обхода.

Те, кто хорошо знаком со свойствами friend, могут заметить возможные серьезные недостатки такого подхода. Поскольку дружественность не наследуется, подкласс Substructure класса Structure не имеет привилегий доступа, предоставляемых этим шаблоном. Иначе говоря, код класса Substructure не имеет доступа к "секретному" интерфейсу Cursor.

Если класс Substructure просто наследует свои операции работы с курсором от класса Structure, ничего не произойдет. Но если в классе Substructure необходимо переопределить эти операции или реализовать другие зависящие от курсора функции, он не сможет использовать закрытые операции класса Cursor. Например, предположим, что в Substructure поддерживается свой собственный связанный список элементов подкласса, которые следует включить в обходы. Когда next достигает конца связанного списка Structure, он должен незаметно передвигаться в начало списка Substructure. Для этого потребуется переопределить next и соответственно переустановить переменную-член курсора _current.

Одним из способов решения проблемы может быть определение в классе Structure защищенных операций, параллельных интерфейсу Cursor за исключением того, что их реализация делегируется объекту Cursor:

```
class Structure {
    // …
protected:
    ListElem* getCurrent (const Cursor& c) const {
        return c.getCurrent();
    }
    void setCurrent (Cursor& c, ListElem* e) {
        c.setCurrent (e);
    }
    // …
};
```

Это, фактически, распространяет привилегии класса Structure на его подклассы. Однако создание параллельных интерфейсов обычно является ошибкой, поскольку приводит к избыточности и затрудняет изменение интерфейса. Хотелось бы этого избежать, а в идеале — вовсе отказаться от использования служебного слова friend.

Именно здесь можно превратить дефект проектирования в полезное свойство. Я даже придумал для него название: *стирание типов* (type laundering). Идея состоит в том, чтобы определить класс Cursor как абстрактный базовый класс, который содержит только те аспекты интерфейса, которые должны быть открытыми, в данном случае это один деструктор:

```
class Cursor {
public:
    virtual ~Cursor ( ) {}

protected:
    Cursor( );
    Cursor(const Cursor&);
};
```

Конструктор по умолчанию и конструктор копирования защищены, чтобы воспрепятствовать созданию экземпляров, т.е. гарантировать, что Cursor ведет себя как абстрактный класс. Того же самого можно добиться, объявив деструктор чисто виртуальным, но тогда придется определять этот деструктор в подклассах, даже если он там не нужен. В любом случае подклассы определяют привилегированный интерфейс:

```
class ListCursor : public Cursor {
public:
    ListCursor () { _current = 0;}

    ListElem* getCurrent () const {return _current;}
    void setCurrent (ListElem* e) {_current = e; }

private:
    ListElem* _current;
};
```

Таким образом, операции класса Structure, использующие Cursor в качестве аргумента, должны выполнить приведение его типа к ListCursor, прежде чем они смогут получить доступ к расширенному интерфейсу:

```
class Structure {
public:
    // …
    virtual void first (Cursor& c) {
      ListCursor* lc;

       if (lc = dynamic_cast<ListCursor*> (&c) ) {
            lc->setcurrent(_head);
                  //_head - голова связанного списка,
                  //поддерживаемого внутри Structure
       }
   }
    // …
};
```

Динамическое приведение типа гарантирует, что структура будет получать доступ и модифицировать только объекты ListCursor.

Осталось описать, как создаются экземпляры курсоров. Очевидно, что клиенты больше не могут создавать экземпляры класса Cursor или его подклассов непосредственно, так как только конкретная структура Structure (или ее подкласс) знает, какой тип курсора используется. Вместо этого нужно воспользоваться вариацией шаблона FACTORY METHOD, чтобы сделать процесс создания экземпляров абстрактным:

```
class Structure {
public:
```

```
    virtual Cursor* cursor () {return new ListCursor;}
    // ..
};
```

Поскольку cursor () возвращает объект типа Cursor*, клиент не имеет доступа к собственным операциям подкласса до тех пор, пока не начнет (динамически) случайным образом выполнять приведение типов, чтобы определить тип — и даже это не даст результата, если ListCursor не экспортируется в файл заголовка. В то же время подклассы Structure имеют все возможности переопределять операции манипулирования с курсором, такие как more, next и element.

На рис. 3.11 кратко характеризуется реализация, основанная на стирании типов. Сравните ее со структурной диаграммой шаблона MEMENTO в книге *Design Patterns*. Основное отличие состоит в появлении дополнительного подкласса ConcreteMemento, который добавляет привилегированный интерфейс к пустому основному интерфейсу Memento. Создатели знают, с какими конкретно хранителями они имеют дело (ведь именно создатели инстанцируют их). Однако смотрители (caretakers) практически ничего не могут делать с хранителями, поскольку все, что они видят— лишь пустые интерфейсы. Хотя на диаграмме этого не видно, стирание типов освобождает C++-реализацию от использования friend и связанных с этим неудобств.

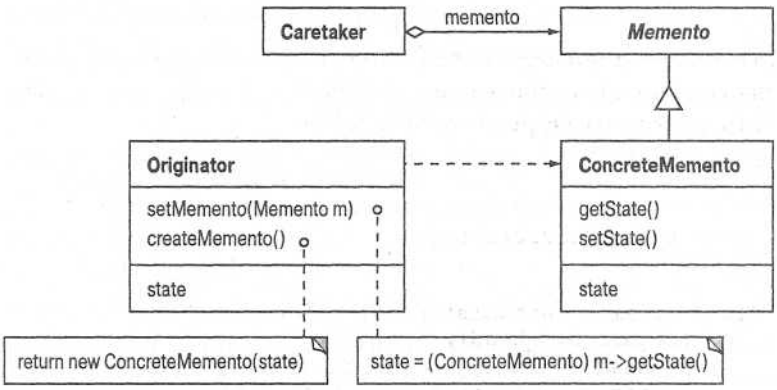


РИС. 3.11. *Усовершенствованная структурная диаграмма, шаблона MEMENTO со стиранием типов*

Потрясающе, насколько простое стирание типов может улучшить проект!

Борьба с утечками памяти

Оказывается, реализация cursor () не была последним штрихом нашего проекта. Если бы мы реализовали данный подход в языке со сборкой мусора, это было бы справедливо. Но в C++ мы сделали клиента ответственным за удаление курсора, созданного операцией cursor (), а это благоприятная возможность для утечки памяти. То, что операция возвращает Cursor*, а не Cursor&, приводит также к неприятным разыменованиям при передаче динамически размещенного курсора операциям вида first и more.

Можно обойти обе эти проблемы, применив подход, известный как панацея Дейкстры (Dijkstra): добавить уровень косвенности. В данном случае мы реализуем вариант предложенной Коплином идиомы Handle-Body [Coplien92]. Роль абстрактного базового класса Memento будет играть не класс Cursor, а новый определенный нами "body"-класс CursorImp:

```
class CursorImp {
public:
    virtual ~CursorImp() {}

    void ref () {++_count;}
    void unref () {if (--_count == 0) delete this;}

protected:
    CursorImp () {_count = 0;}
    CursorImp (const Cursor&);

private:
    int _count;
} ;
```

Как и большинство "тел" в идиоме Handle-Body, объекты CursorImp снабжены счетчиком ссылок. Конкретные подклассы класса CursorImp являются также ConcreteMemento, т.е. они определяют привилегированные интерфейсы:

```
class ListCursorImp : public CursorImp {
public:
    ListCursorImp () { _current = 0;}

    ListElem* getCurrent () const {return _current;}
    void setCurrent (ListElem* e) {_current = e;}
        //те же привилегированные операции, что и раньше

private:
    ListElem* _current;
} ;
```

Теперь мы подошли к основному отличию между данным и исходным подходами: клиенты не работают напрямую с объектами CursorImp. Вместо этого телом нашего класса CursorImp будет управлять конкретный класс Cursor:

```
class Cursor {
public:
    Cursor (CursorImp* i) { -imp = i; _imp->ref();}

    Cursor (Cursor& c) {
        _imp = c.imp();
        _imp->ref(); }
    ~Cursor ()              {_imp->unref() ; }
    CursorImp* imp ()    {return _imp;}

private:
    static void* operator new (size_t)    {return 0;}
    static void operator delete (void *)  {}

    Cursor& operator = (Cursor& c)        {return c;}
..... // для простоты и во избежание ошибок
```

```
        //запрещает динамическое размещение и присваивание
private:
    CursorImp* _imp;
};
```

В качестве управляющего класса Cursor агрегирует экземпляр подкласса CursorImp, а также следит за правильным обращением со счетчиком ссылок. Создатель (Structure) использует эти классы, чтобы возвратить обычный стековый объект Cursor:

```
class Structure { public:
    // ...
    virtual Cursor cursor () {
        return Cursor(new ListCursorImp);
    }
    // …
};
```

В данном случае операция cursor () возвращает некий объект Cursor, а не ссылку на него; тем самым гарантируется, что клиенты будут вызывать конструктор копирования:

```
Structure s;
Cursor c = s.cursor(); //единственная модификация
                       // исходного примера

for (s.first(c); s.more(c); s.next(c)) {
    Element e = s.element(c);
    //использование Element e
}
```

Здесь нет необходимости производить разыменование c, как в исходном случае, когда использовалась версия cursor () , возвращающая указатель.

Последнее изменение необходимо внести в код, который осуществляет динамическое приведение типа, чтобы восстановить тип ConcreteMemento:

```
class Structure {
    // …
    virtual void first (Cursor& c) {
        ListCursorImp* imp;
        if (imp = dynamic_cast<ListCursorImp*>(c.imp ())) {
            imp_>setCurrent(_head);
        }
    }
    // …
}
```

Конечно, в этой версии реализовать Memento несколько сложнее, чем в версии без счетчика ссылок. Зато основанная на стирании типов версия становится столь же простой в использовании для клиентов, как и версия, основанная на friend (которая, кстати, также имеет свои реализационные сложности).

Итак, нам больше не надо беспокоиться о сборке мусора.

Майкл МакКоскер [McCosker97] задал следующий вопрос:

> "Вы упоминали, что можно использовать чисто виртуальный (pure virtual) деструктор, чтобы заставить подклассы определять их собственный деструктор. В моем понимании в языке C++ все деструкторы вызываемые. Однако в среде, в которой я работаю (Win32 на PC), вызов деструктора без адреса приводит к ошибке ввиду отсутствия страницы. Хотелось бы знать, специфична ли эта проблема для данной среды или не следует использовать чисто виртуальный деструктор вовсе?"

Лично я считаю, что в языке C++ *лучше делать некий класс абстрактным путем объявления* защищенными его конструкторов, вместо *того, чтобы объявлять хотя бы одну* функцию-член чисто виртуальной. Пример со стиранием типов должен был отчасти продемонстрировать достоинства такого решения. Рассмотрим вновь класс Cursor:

```
class Cursor {
public:
    virtual ~Cursor() {}

protected:
    Cursor(); Cursor(const Cursor&);
};
```

Единственный альтернативный способ сделать класс Cursor абстрактным состоит в том, чтобы сделать его деструктор чисто виртуальным. Но что означает "чисто виртуальный деструктор"? Деструкторы не наследуются, все они вызываются последовательно. Таким образом, виртуальный деструктор должен быть определен. Возможно ли это?

Конечно! Обратимся к выдержке из руководства по C++[ASC96]:

> "§10.4, пункт 2. Чисто виртуальная функция должна определяться, только если она явно вызывается с помощью синтаксиса *qulified id*...Замечание: объявление функции не может одновременно содержать спецификатор *pure* и определение."

Поскольку виртуальный деструктор будет явно вызываться во время деструкции, из данной выдержки следует, что он *должен быть* определен — не в объявлении, а в отдельном определении:

```
class Cursor { public:
    virtual ~Cursor () = 0;
};

Cursor::~Cursor () {}
```

Я думаю, что в конечном итоге между этими двумя подходами нет большой разницы. В одном нужно сделать защищенными все конструкторы, а в другом — определить чисто виртуальный деструктор.

Pull- и push-модели

В примере, где рассматривалось управление в реальном времени, осталась одна нерешенная проблема: как каркас поставляет события коду приложения? Стандартный подход заключается в том, что каркас определяет операцию

```
virtual Event* nextEvent();
```

которую приложение вызывает всякий раз, когда оно хочет обработать событие. К сожалению, возвращаемое значение неизбежно придется приводить к типу, определяемому в приложении. В случае с торговым автоматом результатом будет следующий код:

```
Event* e = nextEvent();
CoinInsertedEvent* ie;
CoinReleaseEvent* re;
//аналогичные декларации для других типов событий

if (ie = dynamic_cast<CoinInsertedEvent*> (e)) {
    //вызвать специфические операции CoinInsertedEvent
    //для обработки ie
} else if (re = dynamic_cast<CoinReleaseEvent*> (e)) {
    //вызвать специфические операции CoinReleaseEvent
    //для обработки re } else if(...) {
    //...
}
```

Каркас знает только о существовании базового класса Event. Поэтому всякий раз, когда каркас получает событие, он фактически "стирает" всю информацию о его типе за исключением декларированной в классе Event — в том числе и все определенные подклассами расширения. В результате информация о типе теряется и приходится потрудиться, чтобы восстановить ее.

В тех случаях, когда расширения интерфейса являются нормой, это немалая проблема. Код обработчика событий не сохраняет тип. Результаты динамических приведений невозможно проверить во время компиляции, поэтому ошибки программирования, связанные с неправильным указанием типов, вовремя не обнаруживаются. Кроме того, проявляются классические недостатки стиля программирования, основанного на "распознавании и переключении": код становится громоздким и неэффективным, его сложно дополнять.

После того как нам не удалось решить данную проблему с помощью применения шаблона VISITOR, было обещано предложить радикальное решение. Пришло время выполнить обещание.

Первое отличие предлагаемого подхода заключается в том, как происходит доставка событий. До сих пор единственным способом извлечения событий приложением была операция nextEvent. Код приложения вызывает эту операцию, когда приложение готово обрабатывать новое событие. Если вызов nextEvent происходит тогда, когда ожидающих событий нет, генерация вызовов блокируется или операция nextEvent возвращает пустое значение, результатом чего будет активное

ожидание. Выбор сценария остается за разработчиком каркаса[7]. В любом случае именно потребитель инициирует обработку события. Это pull-модель основанного на событиях программирования, поскольку потребитель события (т.е. приложение) является активной сущностью, "вытягивающей" информацию из поставщика событий, каким в нашем случае выступает каркас.

Если pull-модель — это инь, то янь — это push-модель. В ней потребитель пассивно ожидает уведомления о возникновении события. Поскольку поставщик должен "проталкивать" информацию произвольному числу потребителей, модель требует регистрировать в поставщике потребителей, которых он должен уведомлять[8].

Выбор между pull- и push-моделью сводится к тому, где будет сосредоточено управ ление: push-модель стремится упростить код потребителя за счет усложнения кода поставщика, а pull-модель делает обратное. Таким образом, нужно оценить, насколько часто встречается код поставщика и потребителя. Если имеется всего лишь несколько поставщиков и множество потребителей, тогда push-модель предпочтительней. На пример, при моделировании некой схемы могут быть одни глобальные часы и множе ство более мелких схем, зависящих от них. В таком случае, вероятно, лучше сделан сложные часы в соответствии с push-моделью, чем усложнять каждую схему.

Нужно понимать, что все не так просто, могут существовать веские доводы в пользу pull-модели независимо от числа узлов, использующих код поставщика или потребителя. Однако для нашего каркаса системы управления в реальном времени разумно предположить, что будет гораздо больше потребителей событий, чем их поставщи-ков, и поскольку весомых аргументов против нет, выбрать push-модель.

Второе важное отличие данного подхода — отсутствие централизованного источника событий. До сих пор весь механизм доставки событий в каркасе был централизован в операции nextEvent, которая и приводила к стиранию типов. Если централизация приводит к возникновению проблем, очевидно, требуется подобрать некую форму децентрализации.

Однако сначала нужно определить, куда поместить интерфейс, который доставляет события коду приложений — или, по крайней мере, интерфейс доставки событий, *специфических для приложений;* при этом необходимо помнить, что цель — создать расширяемый и сохраняющий тип механизм доставки событий. Данный интерфейс не может находиться в каркасе, иначе мы опять придем к динамическому приведению типов. Нужно доставлять события, сохраняя их тип, и позволить приложениям определять новые типы событий, не изменяя существующий код каркаса и приложения. Эти ограничения, а также переход от pull- к push-модели приводят к отказу от операции nextEvent в качестве единственного интерфейса доставки событий.

Теперь необходимо выяснить, куда передать эти обязанности по доставке событий. Поскольку важно обеспечить расширяемость, рано или поздно придется

---

[7] *Многие каркасы предлагают и блокирующий,, и неблокирующий аналог nextEvent, что предоставляет возможность поддерживать оба варианта; некоторые каркасы также позволяют осуществлять блокировку с тайм-аутом.*

[8] *Push-модель— еще один пример "принципа Голливуда", о котором мы уже говорили в главе 2. Более подробно о pull- и push-моделях можно прочитать в замечательной статье Шмидта (Schmidt) и Виноски (Virwski) [SV97].*

рассмотреть вопрос, что будет меняться при расширении системы. Давайте сделаем это сейчас и согласимся, что изменения, которые нас более всего интересуют в данном контексте, — это *определение новых событий.* Каркас может предопределять некие общие события, такие как TimerEvent (сигнал таймера) или ErrorEvent (ошибка). Но большинство приложений будет определять свои собственные события на более высоком уровне абстракции, такие, например, как классы торгового автомата Coin-InsertedEvent и CoinReleaseEvent.

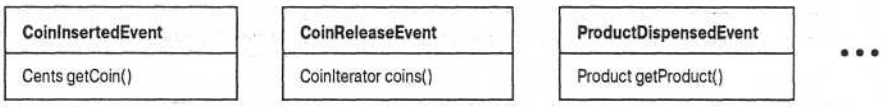
Таким образом, единицей изменений является вид события. Я уделяю этому особое внимание, поскольку соответствие единиц грануляции изменений и расширений — это ключ к минимизации возмущений, вызываемых расширениями. Определенное изменение функциональности должно вызывать соразмерное изменение реализации. Не хотелось бы, чтобы незначительное функциональное изменение приводило к обширньш модификациям кода. А как насчет обратного? Почему бы не сделать так, чтобы для значительного изменения функциональности требовалось лишь незначительное изменение кода?

Такое предложение выглядит заманчивым, но на самом *деле* является утопией. Попытка добиться значительных изхменений функциональности за счет малых изменений кода обычно приводит к одному из двух: или система становится функционально нестабильной и возрастает вероятность возникновения ошибок, или изменения выражаются не в самой системе, а в другой, как правило, интерпретируемой спецификации (например, на языке сценариев, а не на C++). В последнем случае систему вряд ли потребуется модифицировать, но лишь потому, что "системой" в данном случае является интерпретатор. Однако если добавление функций приводит к необходимости менять интерпретатор, это уже грандиозный прокол.

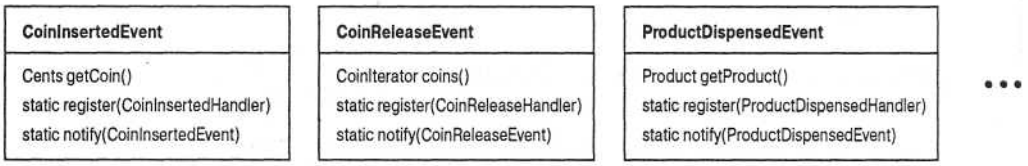
Итак, если считать принцип соответствия масштабов верным, какие из этого следуют выводы применительно к нашему проекту? Мы моделируем каждое единичное изменение — каждый новый вид событий — явно, в виде класса. Класс определяет и реализацию, и интерфейс. Согласно принципу соответствия код, расширяющий функциональность, должен содержать и код реализации, и все специализированные интерфейсы, которые необходимы клиентам для осуществления доступа, сохраняющего тип. Другими словами, возникновение каждого нового вида событий приводит к появлению *одного* нового класса. Больше никакие изменения или дополнения кода не нужны.

Подведем итог. Новый проект должен обладать следующими свойствами: (1) доставлять события, "проталкивая" их потребителям, и (2) создавать не более одного нового класса для каждого специфического для данного приложения события, не внося изменений в существующий код. Это непросто, но мы уже почти у цели.

Первым делом забудем об общем базовом классе для событий. Нормой являются конкретные интерфейсы подклассов. В базовом классе вряд ли будут какие-либо функции. Нам больше не понадобится операция nextEvent, возвращающая полиморфное значение. Если от базового класса больше проблем, чем пользы, лучше определить вместо него отдельные классы событий, имеющие интерфейсы, которые содержат именно то, что нужно клиентам, ни больше и не меньше.

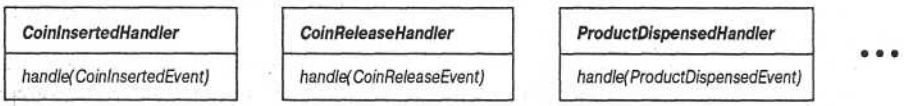


Затем каждый из этих классов получает свой регистрационный интерфейс:



Для регистрации нужны две операции: register и notify. Обе они являются статическими. Каждый экземпляр, который заинтересован в получении событий CoinInsertedEvent, должен зарегистрироваться в классе CoinInsertedEvent. Каждый объект может сообщить о появлении нового экземпляра CoinInsertedEvent, вызвав CoinInsertedEvent::notify и передав этот экземпляр в качестве параметра.

Не всякий старый объект можно зарегистрировать в классе событий, объект должен иметь определенный тип. Обратите внимание на аргумент операции register. Для класса CoinInsertedEvent регистрируемый объект должен быть типа CoinInsertedHandler, для CoinReleaseEvent — типа CoinReleaseHandler и т.д. Эти типы определяются в отдельных классах, которые существуют исключительно для того, чтобы определять интерфейс обработки событий:



Класс, который заинтересован в обработке одного или нескольких из этих событий, должен реализовать соответствующие интерфейсы. Например, класс CoinChanger управляет поведением подсистемы, выдающей сдачу в торговом автомате. Этой подсистеме нужно знать, когда пользователь нажимает кнопку возврата монет, чтобы можно было выдать сдачу, если это положено. Она также должна знать, когда оплата продукта успешно завершена, чтобы подготовиться к следующей сделке. Таким образом, CoinChanger реализует оба интерфейса — и CoinRealeseHandler, и ProductDispensedHandler, как показано на рис. 3.12.

Наконец, подсистема, выдающая сдачу, отвечает за уведомление других подсистем о поступлении монет в монетоприемник. Когда аппаратура фиксирует поступление монеты, класс CoinChanger реагирует на это созданием экземпляра CoinInsertedEvent (что показано пунктирной стрелкой на рис. 3.12). После инициализации данного события соответствующей информацией вызывается операция CoinInsertedEvent::notify, при этом новый экземпляр передается в качестве параметра.

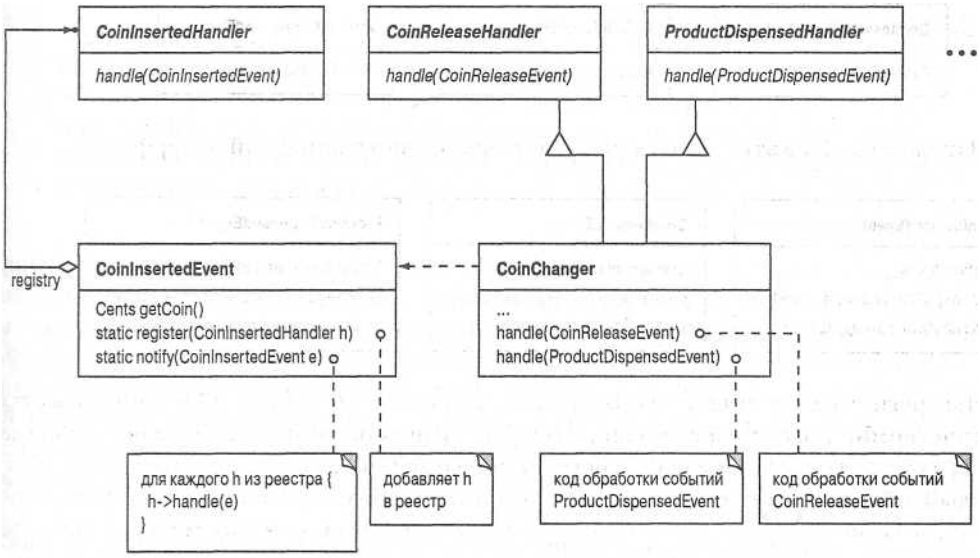


РИС. 3.12. *CoinChanger реализует оба интерфейса: CoinReleaseHandler и ProductDispensedHandler*

Операция notify совершает итерации по всем зарегистрированным реализаторам интерфейса CoinInsertedHandler (т.е. по всем объектам, которые интересовались поступлением монет), вызывает их операцию handle и передает дальше объект CoinInsertedEvent. Тем временем (возможно, при создании экземпляра) объект CoinChanger регистрируется в классах CoinReleaseEvent и ProductDispensedEvent. Таким образом, всякий раз, когда другие подсистемы торгового автомата генерируют CoinReleaseEvent или ProductDispensedEvent, экземпляр CoinChanger узнает об этом. Никаких проверок и приведения типов, никаких проблем.

Расширение производится также просто. Предположим, появилась новая модель торгового автомата, которая допускает расчет купюрами, и требуется включить в программное обеспечение новое событие BillAcceptedEvent. Все, что нужно сделать — это определить класс BillAcceptedEvent и соответствующий BillAcceptedHandler. После чего каждая подсистема, которую затрагивает это новое событие, должна выполнить следующие действия.

1. Зарегистрироваться в BillAcceptedEvent.
2. Наследовать BillAcceptedHandler.
3. Реализовать операцию BillAcceptedHandler::handle, чтобы обрабатывать данное событие.

К сожалению, мы не совсем достигли поставленной цели (определить только один новый класс и не менять существующий код). Пришлось ввести дополнительный интерфейс (BillAcceptedHandler), но это не слишком сложная работа. Изменения существующего кода коснулись только приложения, а не каркаса, который сам по себе может содержать определенное количество предопределенных классов событий и интерфейсов их обработки. Так что кое-чего мы добились.

Марк Бетц [Betz97] пишет:

> "Вам удалось решить вопрос "стирания типов" применительно к обработке событий в каркасе системы управления в реальном времени. Децентрализация действительно является решением, но есть вопрос, прямого ответа на который я не нашел: неужели вы будете проводить децентрализацию обработки событий до тех пор, пока вся она не окажется за пределами каркаса?"

И да, и нет. Каркас по-прежнему может определять классы событий и обработчики стандартных событий (я упоминал два из них, TimerEvent и ErrorEvent). Децентрализация не препятствует повторному использованию, это независимые процессы.

Если обработка событий должна проводиться с сохранением типа, как это было в нашем примере, то каркас не должен заниматься связыванием определенного типа события с обработкой этого события — в этом и состоит задача децентрализации. Когда каркас определяет единый интерфейс обработки событий, связывая событие с обработчиком, это предполагает определенную общность типов событий. Приложения должны устанавливать различие динамически.

Это может быть разумным компромиссом. Статическое задание типов во многих случаях помеха, а не преимущество, не могут же ошибаться 50000 программистов, работающих на Smalltalk! Но чем более крупной и долгоживущей является система, тем более вероятно, что она выиграет от статической типизации.

Кроме всего прочего, данный пример продемонстрировал применение шаблона MULTICAST, который в свое время доставил нам столько хлопот. Более подробно этот шаблон обсуждается в следующей главе.

Глава 4

Муки творчества

MULTICAST— это шаблон (если его вообще можно назвать шаблоном), который постоянно находится в стадии разработки. Я подумал, что будет весьма поучительно представить его здесь несмотря на все его несовершенство. Я даже включил в эту главу все наши обсуждения данного шаблона, которые могут оказаться еще более поучительными, чем сам шаблон. Мы не наделены даром предвидения, поэтому те, кто приписывает "банде четырех" экстраординарные способности, будут поражены хаотичностью нашего процесса разработки.

Я пропускаю раздел "Назначение" и сразу перехожу к мотивации. Здесь предлагается сценарий, весьма похожий, но не идентичный примеру с торговым автоматом, который рассматривался в предыдущей главе.

Мотивация

Программа *основана на событиях* (event-driven), если поток управления направляется внешними стимулами, называемыми *событиями.* Подобные программы часто встречаются в приложениях, осуществляющих управление в реальном времени. Основные задачи проектирования заключаются в том, чтобы сделать такие системы расширяемыми и в то же время сохраняющими тип.

Рассмотрим современный управляемый компьютером торговый автомат. Он состоит из нескольких подсистем, среди которых автомат, выдающий закуску, подсистема выдачи сдачи, панель выбора продуктов, алфавитно-цифровой дисплей и "черный ящик" — небольшой компьютер, который управляет всей системой. Взаимодействия между подсистемами и покупателем достаточно сложные, поэтому предлагается моделировать как подсистемы, так и их взаимодействия между собой в виде объектов.

Когда покупатель бросает в автомат монету, объект CoinChanger (который следит за подсистемой выдачи сдачи) создает объект CoinlnsertedEvent. Этот объект фиксирует подробные характеристики события, в том числе время и сумму в центах. Другие классы моделируют иные интересующие нас события. Класс KeyPressEvent обозначает нажатие кнопок на панели выбора продуктов. Экземпляр класса CoinReleaseEvent фиксирует запрос покупателя на возврат монет. Объекты ProductDispensedEvent и Product-RemovedEvent обозначают конечные результаты сделки по получению закуски. Множество классов событий достаточно большое и потенциально открытое: добавление устройства, принимающего банкноты, и связанных с ним событий (BillInsertedEvent) должно требовать минимальных изменений существующего кода.

Что происходит после создания события? Какие объекты будут использовать данное событие и как оно попадет к ним? Ответы на эти вопросы зависят от типа события (рис. 4.1). Объект CoinChanger интересуется всеми событиями CoinReleaseEvent,

которые происходят после того, как была брошена монета, однако данный объект не должен получать события CoinInsertedEvent, которые он создает для других объектов. Аналогично, объект Dispenser создает экземпляры ProductDispensedEvent, но он не заинтересован в их получении. Зато объект Dispenser весьма заинтересован в получении объектов KeyPressEvent, поскольку они определяют, какие закуски упаковывать. Таким образом, интересы различных подсистем меняются, возможно, даже динамически.

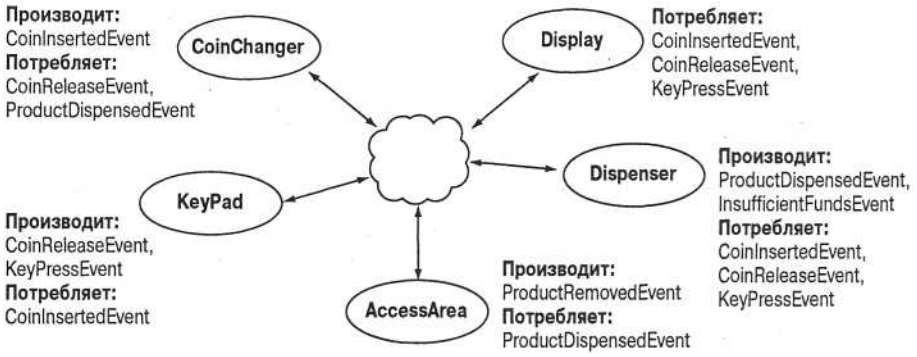


РИС. 4.1. *Торговый автомат: события, их поставщики и потребители*

Получилось запутанное множество зависимостей между объектами событий, их поставщиками и потребителями. Сложные зависимости нежелательны — в таком случае систему трудно понимать, поддерживать и модифицировать. Надо сделать так, чтобы можно было легко изменить связь между объектом и интересующим его событием не только статически, но и динамически.

Стандартное решение состоит в использовании реестра событий для отслеживания этих зависимостей. Клиенты регистрируют свой интерес к определенному событию в реестре. Создав событие, объект передает его в реестр, который направляет это событие заинтересованным объектам. При таком подходе требуется два стандартных интерфейса — один для событий, а второй для объектов, которые хотят эти события обрабатывать (рис. 4.2).

Когда некий экземпляр подкласса Handler (например, CoinChanger) получает событие посредством своей операции handle (которая реализует обработку данного события в CoinChanger), конкретный тип события статически неизвестен. Это важно, поскольку различные типы событий фиксируют различную информацию; невозможно придумать единый интерфейс Event, который сможет предусмотреть потребности всех подклассов. Следовательно, каждый подкласс Event расширяет базовый интерфейс Event операциями, специфическими для данного подкласса. Чтобы добраться до этих операций, CoinChanger должен попытаться привести тип события к типу, который он в состоянии обработать:

```
void CoinChanger::handle (Event* e) {
    CoinReleaseEvent* cre;
    ProductDispensedEvent* pde;
    //аналогичные объявления других обрабатываемых событий
```

```
    if   (cre = dynamic_cast<CoinReleaseEvent*>(e))    {
        //обработка  CoinReleaseEvent
    } else if (pde = dynamic_cast<ProductDispensedEvent*> (e) ) {
        //обработка ProductDispensedEvent
    } else if(...){
      //...
    }
}
```

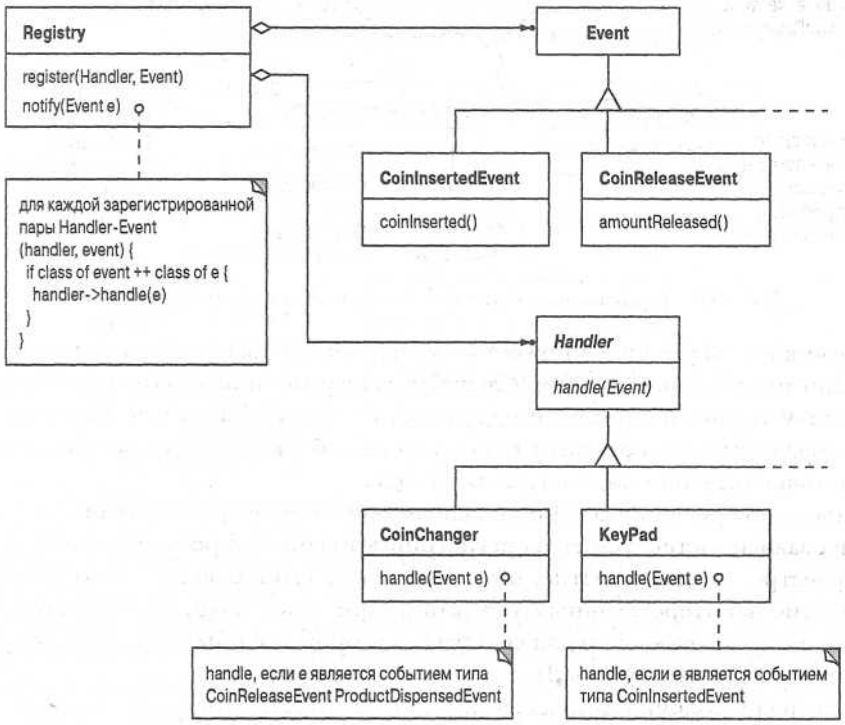


РИС. 4.2. *Подход с использованием реестра событий*

Проблема состоит в том, что данный подход не сохраняет тип. Чтобы добраться до специфических операций подклассов, необходимо производить динамическое приведение типов, результаты которого невозможно проверить во время компиляции. Это означает, что некоторые ошибки программирования, связанные с неверным указанием типа, могут остаться невыявленными до момента выполнения программы. Такой код содержит все недостатки стиля программирования, основанного на "распознавании и переключении": он громоздкий, неэффективный, его тяжело расширять.

Шаблон MULTICAST позволяет доставлять информацию заинтересованным объектам расширяемым и сохраняющим статическую типизацию образом. В шаблоне не требуется создавать иерархии Event или Handler. Вместо этого определяется абстрактный класс-обработчик для каждого конкретного класса событий — например,

CoinReleaseHandler для CoinReleaseEvent. Любой класс объектов, который хочет обрабатывать CoinReleaseEvent, должен наследовать классу CoinReleaseHandler. Аналогично происходит обработка других событий: заинтересованные стороны должны наследовать соответствующим классам обработчиков.

На рис. 4.3 класс CoinChanger наследует и классу CoinReleaseHandler, и классу ProductDispensedHandler, поскольку он должен обрабатывать как события CoinReleaseEvent, так и ProductDispensedEvent — возврат монет может понадобиться в обоих случаях. Каждый класс-обработчик определяет операцию handle, в которой подклассы реализуют свою обработку событий. Однако в отличие от подхода с использованием реестра, аргументом операции handle является требуемый конкретный тип события, и необходимость в динамическом преобразовании типов отпадает, т.е. данная операция сохраняет статическую типизацию.

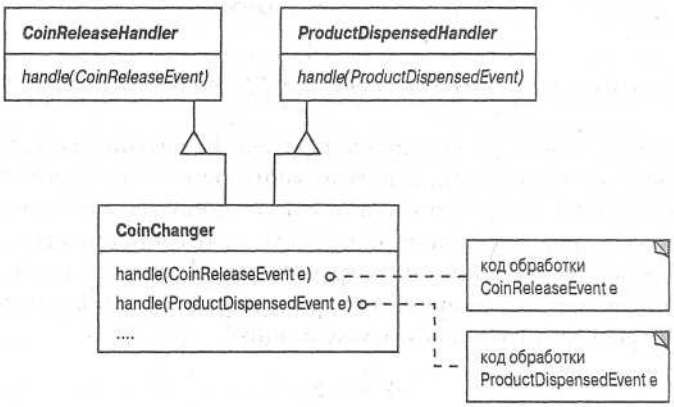


РИС. 4.3. *Реализация интерфейсов Handler в классе CoinChanger*

Теперь нужно определить, каким образом события *доставляются* заинтересованным объектам, т.е. как вызывается операция handle. Можно, как и прежде, определить класс Registry с операциями register и notify, только теперь будет множество классов обработчиков, не связанных отношениями наследования. Следовательно, нужна не одна, а несколько операций register, по одной для каждого типа обработчиков. Кроме того, придется добавлять еще одну операцию register к классу Register всякий раз, когда вводится новый вид событий. Иными словами, необходимо будет изменить существующий код.

Децентрализация позволяет решить эту проблему. Вместо того чтобы регистрироваться в одном большом реестре, заинтересованные клиенты могут регистрироваться непосредственно в объектах, создающих события. Например, если клиент интересуется событиями CoinInsertedEvent, он регистрируется в CoinChanger — классе, который поставляет эти события (рис. 4.4).

Всякий раз, когда объект CoinChanger генерирует CoinInsertedEvent, он вызывает свою операцию notify, чтобы доставить данное событие всем зарегистрированным обработчикам CoinInsertedHandler. Компилятор гарантирует, что обработчики получат объект требуемого типа — CoinInsertedEvent.

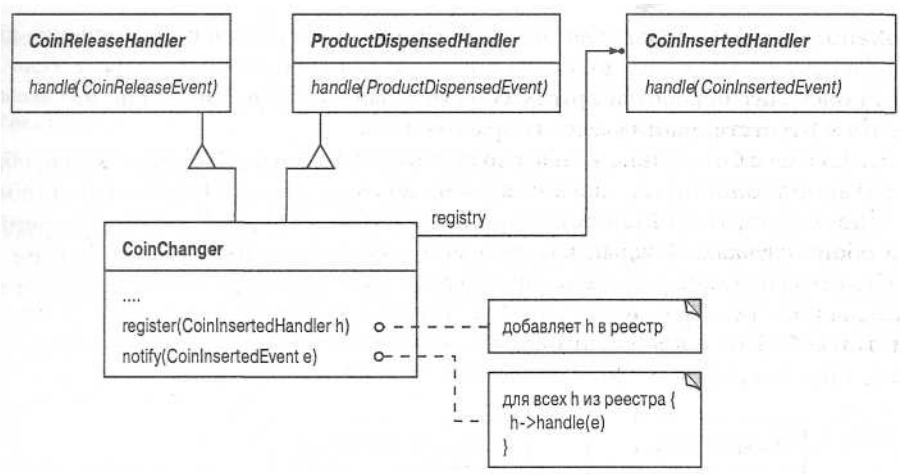


РИС. 4.4. *Регистрация в классе CoinChanger; интерфейс и реализация*

Аналогично клиент, заинтересованный в получении событий ProductDispensed-Event, регистрируется в Dispenser. В целом, заинтересованность в получении определенного класса событий регистрируется в классе, порождающем данные события. Подобная децентрализация регистрации способствует расширяемости: при определении нового вида событий изменения кода ограничиваются классом или классами, которые создают данные события, в то время как при централизованном подходе требуется изменять как интерфейс реестра, так и его реализацию.

Основное отличие проекта, основанного на применении шаблона MULTICAST, от проекта, предложенного в конце предыдущей главы, состоит в регистрации клиентами интереса к определенному событию в классе, который порождает данное событие (например, в классе CoinChanger). В исходном проекте клиенты регистрировались в самих классах событий. То же происходило и в моей первой редакции раздела "Мотивация", на что Эрих (Erich Gamma) возразил:

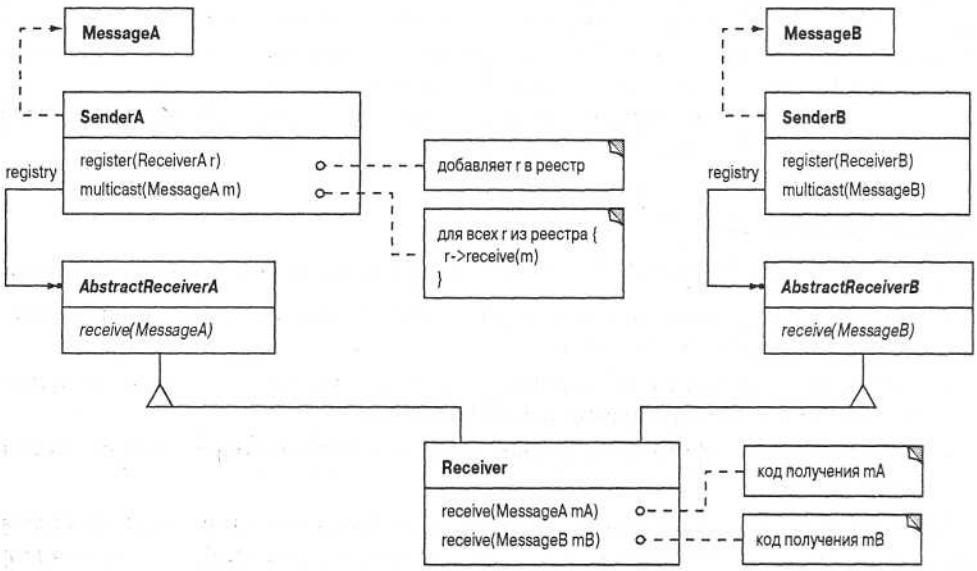
> "На мой взгляд, нужно регистрировать заинтересованность не в самом событии, а в его *отправителе.* В таком случае для торгового автомата следует предусмотреть операции вида addCoinReleaseHandler, addCoinInsertedHandler и т.д."

Со своей стороны, мне хотелось рассмотреть ситуацию, когда интерфейс регистрации помещается в классы событий. Цель — упростить расширение и избежать переделок при определении новых видов событий. Если регистрационный механизм находится в существующем классе, придется изменять его, чтобы включить новые операции регистрации. Помещая регистрационный интерфейс в сами события, проще включать новые события.

Однако с точки зрения моделирования Эрих прав, поскольку уведомление класса событий действительно может показаться чем-то неестественным. Даже если речь идет о статической операции с классом, а не с экземпляром, все равно это выглядит так, будто событие уведомляется о нем самом!

Поскольку соображение, высказанное Эрихом, соответствовало более общему подходу, мы согласились, что именно оно должно стать основой проектирования, поэтому в разделах "Структура", "Участники" и "Взаимодействия" поместили регистрационный интерфейс в отправитель. Если бы я продолжал настаивать на своем, дело закончилось бы путаницей: в разделе "Мотивация" предлагался один проект, а в других разделах — несколько иной.

Структура



Участники

Message (сообщение; ProductDispensedEvent)

- инкапсулирует информацию, которую необходимо передать от отправителя (Sender) получателю (Receiver).

Sender (отправитель; Dispenser)

- ведет реестр объектов-получателей;
- определяет интерфейс регистрации объектов-получателей;
- определяет и реализует интерфейс доставки сообщения зарегистрированным объектам-получателям.

AbstractReceiver (абстрактный получатель; ProductDispensedHandler)

- определяет интерфейс для получения объекта Message.

Receiver (получатель; CoinChanger)

- реализует один или несколько интерфейсов AbstractReceiver.

Взаимодействия

- Клиенты с помощью регистрационного интерфейса отправителя регистрируют получателей в отправителях;
- отправители порождают сообщения и доставляют их зарегистрированным получателям.

Таким образом, разделы "Структура", "Участники" и "Взаимодействия" иллюстрируют общий случай. В рассмотренных ранее примерах классы Sender и Message были просто объединены в классы Event. Это вполне жизнеспособная версия, поэтому мы вернемся к ее рассмотрению в разделе "Реализация".

До сих пор в нашей четверке в основном царило согласие, но при переходе к разделу "Применимость" появились определенные расхождения.

Применимость

Используйте шаблон MULTICAST, когда выполнены *все* перечисленные ниже условия.

- Определенные классы объектов могут быть заинтересованы в получении информации от других объектов.
- Информация имеет произвольную структуру и сложность и может изменяться по мере эволюции программного обеспечения.
- При передаче информации должно быть обеспечено сохранение статической типизации.

Хотя эти пункты сами по себе ни у кого не вызвали возражений, последний можно считать своеобразным источником наших дальнейших разногласий. Следует упомянуть, что Эрих (Erich Gamma), Ричард (Richard Helm) и я программировали на достаточно строго типизированном языке C++, а Ральф (Ralph Johnson) — на языке Smalltalk, где нет такого понятия, как статическая проверка типов. Возможно, это поможет понять, почему трое из нас считали, что MULTICAST заслуживает статуса самостоятельного шаблона, в то время как Ральф думал, что данный шаблон является всего лишь вариацией шаблона OBSERVER и его следует описать как часть новой расширенной версии OBSERVER.

Действительно, есть определенная схожесть между OBSERVER и MULTICAST. Оба поддерживают зависимости между объектами, оба передают информацию от одних объектов другим и оба направлены на поддержку расширяемости. Тем не менее, большинство из нас чувствовали, что между этими шаблонами имеется и существенное различие. Эрих первым выразил это:

> "MULTICAST очень близок к OBSERVER, но между ними есть и различие."

Я попытался сформулировать, в чем состоит это различие, и вот что у меня получилось:

> "В шаблоне OBSERVER говорится о зависимости один-ко-многим. Перед применением шаблона субъект и его наблюдатели, скорее всего, образовывали один объект. OBSERVER осуществляет разбиение этого объекта, чтобы обеспечить гибкость. Он не очень интересуется содержанием информации, которая передается между объектами, или ее расширяемостью; шаблон нацелен на уведомление и согласованность пар субъект-наблюдатель.
>
> В шаблоне MULTICAST основное внимание уделяется информации, которую отправитель пересылает получателю, ее расширяемости и сохранению типа. Более того, отправитель и получатель обычно не связаны в исходной модели и взаимодействия между ними большей частью непредсказуемы и динамичны."

Однако Ральфа (в силу его сложившихся взглядов) этот довод не убедил:

> "При реализации OBSERVER несомненно придется *интересоваться,* что и куда пересылается. Однако в описании данного шаблона в *Design Patterns* недостаточно говорится об этом; там содержатся подсказки относительно применения push-модели вместо pull-модели и т.п., но все это весьма расплывчато и неинформативно. Необходимо описать шаблон OBSERVER более конкретно и обсудить проблемы, которые могут возникнуть при его использовании. На мой взгляд, все, кто использует шаблон OBSERVER в крупных проектах, вынуждены будут задуматься об этом. Я нисколько не сомневаюсь, что отправитель и получатель в шаблоне MULTICAST не связаны в том, что они моделируют. Возможно, я ошибаюсь, но в среде VisualWorks, основанной на языке Smalltalk, отправители и получатели также независимы и их связи могут быть весьма динамическими. А поскольку VisualComponents (типичный Observer) и ValueModels (типичный Subject) постоянно используются повторно, перемешиваются и согласовываются, связи между ними также достаточно непредсказуемы."

Если учесть, что в языке Smalltalk отсутствует даже подобие множественного наследования интерфейсов, нет ничего удивительного в том, что Ральф считает шаблон MULTICAST некой разновидностью OBSERVER. Не кажется ли вам, что MULTICAST более похож на идиому для строго типизированных языков, чем на шаблон?

В следующем послании Ральф сформулировал, почему, на его взгляд, остальные члены четверки считают, что MULTICAST достоин быть самостоятельным шаблоном проектирования:

> "Я думаю, что одной из до сих пор не обсуждавшихся причин, почему вы считаете важным выделить MULTICAST, является статическая проверка типов. Вы стараетесь избежать приведения типов. Есть два различных способа сделать это. Один состоит в том, чтобы передать некий Event, с которым можно справиться. Другой— создать отдельные иерархии Handler (обработчиков). Я думаю, что если сделать первое, второе делать не понадобится, поэтому

> мне непонятно, почему MULTICAST нуждается и в том, и в другом. Возможно, это происходит потому, что вы хотите поместить поведенческие аспекты в класс Observer (т.е. Handler), а не в Event. Однако я считаю необходимым указать в самом шаблоне OBSERVER, насколько все усложняется при статической проверке типов.
>
> В целом, шаблон OBSERVER создает обширное пространство для всевозможных вариаций. Я не вижу, чем MULTICAST так уж отличается от других вариаций. OBSERVER — это шаблон, а не повторно используемый механизм. Он претерпевает изменения при каждом новом использовании. Говоря о MULTICAST, вы на самом деле говорите о неких часто встречающихся полезных вариациях шаблона OBSERVER. Но вы приняли решение выделить определенное множество вариаций, в то время как Другие множества вариаций, не менее интересные, вами игнорируются. Я считаю, что будет лучше исследовать ситуацию в целом."

Я должен был согласиться с большей частью сказанного; судя по всему, так же поступили остальные. Но невозможно до бесконечности нагружать шаблон лишней информацией! Поэтому вместо того, чтобы оспаривать эти заявления, Эрих задал иной вопрос:

> "Допустимо ли предлагать значительное уточнение и вариацию некого шаблона в виде самостоятельного шаблона? Вместо того чтобы иметь 20 разновидностей реализации, я предпочитаю иметь отдельный шаблон."

Здесь кроется важный вопрос, который касается масштабируемости шаблонов. Вот мои мысли по этому поводу:

> "Существует благоприятная возможность сделать наши шаблоны более масштабируемыми. В свое время я завел для каждого шаблона файл комментариев, откликов и любых новых мыслей. Многие файлы получились достаточно большими. Если мы включим, скажем, в шаблон SINGLETON все, что мы о нем узнали, результат будет ужасно громоздким. Некоторые из наших шаблонов — например, OBSERVER и COMPOSITE — и так уже достаточно громоздки.
>
> Как можно обеспечить "масштабируемость"? Одна из возможностей состоит в том, чтобы встроить все шаблоны в некий шаблонный язык. Однако многое из того, что было написано по данному поводу, кажется мне недостаточно убедительным. Это была бы настоящая победа, если бы удалось создать некую суперструктуру, которая охватывает существующие шаблоны и оставляет пространство для новых озарений, расширений и вариаций. Во всяком случае, из ситуации с MULTICAST я *сделал* вывод, что наши шаблоны не могут расти до бесконечности."

Одновременно я пришел к выводу, что нужно ответить на несколько фундаментальных вопросов:

1. Какая связь существует между шаблонами MULTICAST и OBSERVER?
2. Зависимы ли они? И если да, должны ли мы объединить их в один шаблон?

"Я продолжаю считать, что OBSERVER/MULTICAST — различные, но связанные шаблоны... Подумаем, в чем отличаются их концепции: в OBSERVER это конкретные наблюдатели и, возможно, аспекты[1] субъектов, в MULTICAST это типы событий— вот в чем заключается основное отличие этих двух шаблонов. Я не воспринимаю MULTICAST как некое расширение OBSERVER и наоборот...

Общей для данных шаблонов является концепция *решения* по регистрации и уведомлению. Но это в основном механизм, который позволяет динамически связываться отправителям и получателям. Он относится не к проблеме, которую решает каждый из шаблонов, а к тому факту, что эти шаблоны устанавливают связи между объектами, и механизм регистрации/уведомления является базовым механизмом (шаблоном?), позволяющим делать это. Рассмотрим также охват и вариации данных шаблонов. Вариации MULTICAST:

M_a: Глобальное уведомление с регистрацией в классе Event; например, MyEvent::register(MyEventHandler) — "широкое вещание", как в примере Джона.

$M_ь$: Локальное уведомление с регистрацией в отправителе; например, Sender::registerMyEventHandler(MyEventHandler) — как предпочитает Эрих и более близко к моей идее "узкого уведомления". M_c: Локальное уведомление с регистрацией в отправителе и использованием единого класса Event; например, Event. Вариации OBSERVER:

O_a: Уведомление о неспецифическом изменении (примитивный OBSERVER); например, Observer: : update ().

O_b: Уведомление о полуспецифическом изменении; например, рекомендации— Subject::register(Aspect, Observer). O_c: Уведомление о специфическом изменении; например о событии — Subject::registerMyEvent{MyEventHandler). Обратите внимание, что $M_ь$ и O_c, а также O_a и M_c очень похожи. С этой точки зрения, (a) OBSERVER— это уточнение MULTICAST, но точно так же (b) MULTICAST является уточнением OBSERVER. В случае (a) происходит ограничение масштаба событий до получения специализированного OBSERVER. В случае (b) происходит расширение масштаба Subject в OBSERVER до получения специализированного MULTICAST. Оба эти взгляда корректны.

В точности такое же явление наблюдается при помещении обходов в Visitor, в результате чего получается нечто близкое к шаблону ITERATOR, или при расширении STRATEGY до BUILDER.

---

[1] *Аспект (aspect) — это объект, который точно специфицирует изменение. Поставлял аспект вместе, уведомлением., можно упростить обновление наблюдателя. Аспекты упоминаются в книге* Design Patterns в *разделе, посвященном OBSERVER.*

> Мне нравится начинать с простейших шаблонов и анализировать, какие изменения нужно внести, чтобы получить другой шаблон. В рассматриваемом случае нужно идти от M_a к M_b, к M_c и далее к O_a , чтобы из MULTICAST получить OBSERVER. Я всегда придерживался мнения, что истинное понимание шаблонов приходит с осмыслением их взаимоотношений."

Однако Ральф уже учел почти все эти аргументы. Подробности не важны, достаточно лишь сказать, что мы оказались в тупике. Когда наша дискуссия достигла этого квазиматематического уровня, я понял, что дело близится к развязке. И я не ошибся: компромиссное решение было совсем близко. Но я опережаю события.

В этих дебатах меня больше всего беспокоила узость спектра высказанных мнений. Поэтому я очень обрадовался, когда Боб Мартин (Bob Martin, [Martin97]), сторонник строгой типизации, сообщил мне о своих мыслях по данному поводу.

> "Я хочу ответить на вопросы, которые возникли в ходе дискуссии вокруг MULTICAST. Является ли MULTICAST вариацией OBSERVER? Я думаю, что нет, и вот почему: наблюдатели знают свои субъекты, а обработчики в MULTICAST не обязаны знать свои источники событий. Если в шаблоне OBSERVER необходимо знать, когда меняется состояние некого субъекта, то регистрируется его наблюдатель. Но в шаблоне MULTICAST интерес представляет возникновение определенного события, а не источник его возникновения. (Поэтому мне больше нравится ваша идея поместить функцию регистрации в Event, чем идея Эриха включить регистрационный интерфейс в источник событий.)[2] Рассмотрим событие, состоящее в нажатии клавиши клавиатуры. Система может быть снабжена основной и дополнительной клавиатурой. Оба эти прибора могут быть источником рассматриваемых событий. Программному обеспечению не важен источник. Ему не важно, была ли клавиша "3" нажата на основной клавиатуре или на дополнительной, важно лишь, какая клавиша была нажата. Тоже самое происходит при использовании мыши и джойстика, или в любом другом случае, когда имеется несколько источников событий.
>
> Я считаю, что в этом состоит основное отличие между MULTICAST и OBSERVER. В MULTICAST действительно ведутся наблюдения, но не известно, за чем именно. Регистрация производится не в источнике интересующих нас стимулов."

Пример с основной и дополнительной клавиатурой особенно показателен. Он вновь подчеркивает то, что я отмечал ранее, сравнивая связь между шаблонами MULTICAST и [OBSERVER со связью между шаблонами ABSTRACT FACTORY и FACTORY METHOD. Как правило, реализация ABSTRACT FACTORY использует фабричный метод для каждого

---

[2] *Чтобы позиция Эриха была более понятна, он не возражал против помещения регистрационного интер-фейса в класс Event. Он просто хотел, чтобы раздел "Мотивация" отражал общий случай, описанный в после-дующих разделах.*

продукта, но это не означает, что ABSTRACT FACTORY является расширением FACTORY METHOD. Это разные шаблоны, поскольку их назначения совершенно различны.

Аналогично, восприятие MULTICAST в качестве расширения OBSERVER противоречит целям этих шаблонов: OBSERVER поддерживает зависимость один-ко-многим между объектами, в то время как задача MULTICAST — доставка инкапсулированной информации объектам способом, сохраняющим тип и допускающим расширения. Чтобы подчеркнуть данное отличие, я предложил использовать MULTICAST для реализации зависимости многие-к-одному — как в приведенном Бобом примере, что явно не совпадает с назначением OBSERVER.

Наконец удалось доказать, что данные шаблоны различны (по крайней мере так я думал), однако Эрих не дал мне насладиться победой, переведя дискуссию в иную плоскость:

> "В своем курсе "Шаблоны проектирования и Java" я объяснял новую модель обработки событий JDK 1.1. Это вынудило меня вновь вернуться к рассматриваемым вопросам. Оказывается, удобно начать объяснение проекта, представив данную модель как пример связи вида OBSERVER, тем более, что большинство программистов с ней знакомы. На следующем этапе я объяснял уточнения, связанные с сохранением типов событий и регистрацией заинтересованности. Поэтому я считаю важным зафиксировать, что MULTICAST — это уточнение OBSERVER, и описать, в чем заключаются уточнения, а не спорить о том, отличается ли MULTICAST от OBSERVER. Я по-прежнему считаю, что уточнение может быть самостоятельным шаблоном.
>
> В книге "банды четырех" есть другой пример подобного рода. При написании *Design Patterns* мы спорили о том, не *является* ли шаблон BUILDER просто неким Strategy для порождения объектов. Тем не менее, BUILDER вошел в книгу как самостоятельный шаблон.
>
> Что касается названия: после того как я прочитал аргументы, относящиеся к типам, меня привлекло название *типизированное сообщение."*

Утверждения в конце первого абзаца сообщения Эриха могут показаться спорными, но они направили мои мысли в новое русло. Все прояснялось. Это были последние часы шаблона MULTICAST в его первоначальном виде.

Однако я захотел удостовериться, что правильно понял следствия заявления Эриха:

> "Итак, вы предлагаете убрать регистрацию и доставку из MULTICAST, поместить их в OBSERVER и сохранить остаток как TYPED MESSAGE?"

Ответ последовая незамедлительно:

> "Да, именно так. Если помните, мы уже обсуждали регистрацию заинтересованности в OBSERVER *(Design Patterns,* с. 298). Чем больше я об этом думаю, тем больше мне нравится эта идея."

Показательно, что всегда спокойный Эрих не удержался от проявления эмоций. Немного позже свое благословение высказал и Ральф:

"Мне гораздо больше нравится, если шаблон MULTICAST/TYFED MESSAGE будет представлен так, как предлагает Эрих. Связь данного шаблона с OBSERVER очевидна, поэтому если мы не подчеркнем ее, может показаться, что мы стараемся что-то скрыть.

Когда мы писали нашу книгу, мы действительно пытались кое-что скрыть, Нам хотелось избежать разговоров о том, что один шаблон является специализацией другого или один шаблон содержится в другом в качестве его компонента. Мы не хотели отвлекаться, и решили говорить только о шаблонах. Отчасти поэтому мы не описали в качестве самостоятельного шаблона "абстрактный класс", поскольку он содержится в большинстве шаблонов.

Я думаю, это было оправданное решение при создании первого каталога шаблонов, но сейчас ситуация изменилась. Люди хотят знать о связях между шаблонами, и мы должны рассказать им об этом. Связь здесь настолько очевидна, что необходимо подчеркнуть ее, а не только упомянуть в разделе "Родственные шаблоны".

Итак, длительные споры, наконец, завершились. Необходимо расширить шаблон OBSERVER с учетом создания нового шаблона TYPED MESSAGE. Наличие этих двух дополняющих и усиливающих друг друга шаблонов делает существование отдельного шаблона MULTICAST излишним.

Что собой представляет новый шаблон? Он действительно достаточно похож на MULTICAST, по крайней мере в нынешней редакции. Мы еще достаточно далеки от окончательной версии TYPED MESSAGE. Тем не менее, ниже предлагается краткое описание этого шаблона в его современном состоянии.

Назначение

Инкапсулировать информацию в объекте, чтобы ее можно было передавать сохраняющим тип способом. Клиенты могут расширять данный объект, добавляя в него информацию, при этом сохранность типов не нарушается.

Мотивация

[В основном тот же пример с торговым автоматом, который предлагался для MULTICAST, с минимальными изменениями, чтобы подчеркнуть инкапсуляцию и расширение событий, и, напротив, убрать акцент с процесса уведомления. Некоторые фрагменты кода помещены в разделе "Образец кода".]

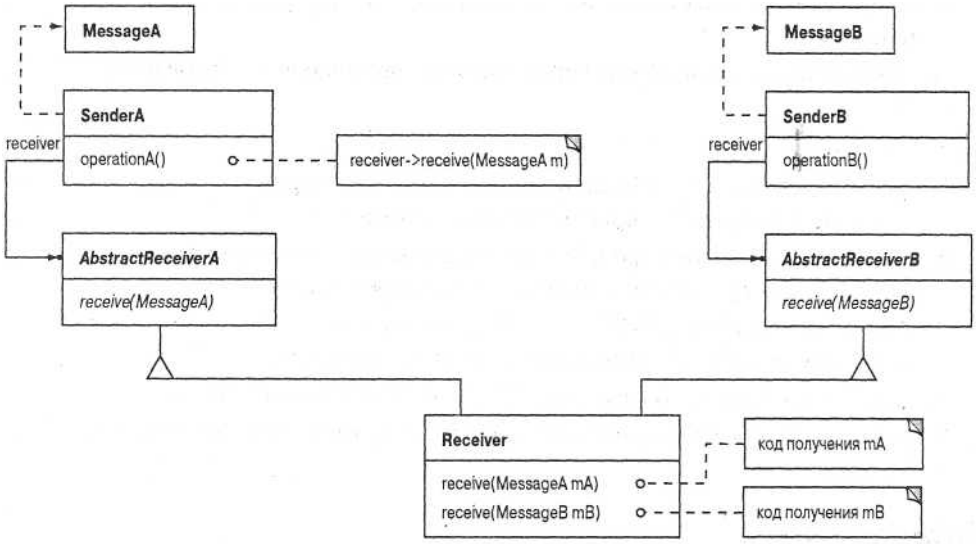
Применимость

Шаблон TYPED MESSAGE следует использовать, когда выполнены все перечисленные ниже условия.

- Определенные классы объектов могут быть заинтересованы в получении информации от других объектов.
- Информация имеет произвольную структуру и сложность и может меняться по мере эволюции программного обеспечения.

- Передача информации должна осуществляться способом, позволяющим сохранить статическую типизацию. [Аналогично MULTICAST.]

Структура



Участники

Message (ProductDispensedEvent)

- инкапсулирует информацию, которую следует передать от отправителя (Sender) получателю (Receiver).

Sender (Dispenser)

- поддерживает ссылку на объект-получатель (Receiver);
- реализует одну или несколько операций, посылающих сообщение получателю.

AbstractReceiver (ProductDispensedHandler)

- определяет интерфейс *для* получения объекта Message.

Reseiver (CoinChanger)

- реализует один или несколько интерфейсов AbstractReceiver.

Взаимодействия

- Отправитель порождает сообщение и доставляет его получателю.
- Сообщение пассивно — оно не инициирует связь с отправителями или получателями.

Последствия

1. Информация может передаваться способом, сохраняющим типы и допускающим расширения, без приведения типов или переключающих операторов...
2. При совместном использовании с OBSERVER поддерживает неявную активизацию...
3. Отсутствие множественного наследования (интерфейсов) затрудняет применение...
4. Может приводить к образованию спиральных графов наследования...

Реализация

1. Нет необходимости обеспечивать взаимно однозначное соответствие между классами Message и интерфейсами AbstractReceiver...
2. Слияние классов Message и Sender в один класс облегчает расширения, но при этом трудно определить, кто посылает конкретное сообщение...
3. Реализация шаблона TYPED MESSAGE в языке со слабой типизацией или с отсутствием поддержки множественного наследования...
4. Комбинированное применение TYPED MESSAGE и OBSERVER...
5. Реализационные компромиссы при определении общего базового класса для Message...

Образец кода

[Фрагменты кода из примера для торгового автомата, описанного в разделе "Мотивация", в том числе альтернативные реализации — проект с общим классом Sender-Message и реализация, использующая композицию вместо множественного наследования.]

Известные применения

Как отмечал Эрих, в основанной на делегировании модели событий Java в JDK 1.1 [Java97] шаблон TYPED MESSAGE используется совместно с OBSERVER. Я использовал его в одном проекте компании IBM Research, но эта работа до сих пор не опубликована. Предлагаю читателям присылать сообщения о других известных применениях.

Родственные шаблоны

[Безусловно, родственным шаблоном является OBSERVER.]

Типизированные сообщения можно ошибочно принять за команды (см. шаблон COMMAND [GoF95]). Различие здесь также состоит в назначении. Команда инкапсулирует некую операцию, в то время как типизированное сообщение инкапсулирует состояние. Первая является активной, второе — пассивным. Кроме того, в TYPED MESSAGE внимание уделяется возможности расширения при сохранении типов, чего в шаблоне COMMAND нет.

Шаблон TYPED MESSAGE может казаться достаточно близким шаблону MEMENTO, который часто используется совместно с COMMAND. Однако назначение MEMENTO противоположно назначению TYPED MESSAGE: объект-хранитель должен препятствовать передаче информации другим объектам. Инкапсулированная в нем информация предназначена только для его создателя.

Поговорим о совпадениях. Примерно в то же время, когда мы пришли к решению относительно TYPED MESSAGE, Поль Пеллетье (Paul Pelletier) применил рекурсивно вызываемый шаблон Коплина [Coplien95] к MULTICAST и независимо пришел к упрощенному способу реализации нашего нового шаблона [Pelletier97].

> "Прочитав о вашем проекте, основанном на применении MULTICAST, я подумал, что, возможно, с его помощью можно сделать неплохой шаблон для дальнейшего упрощения процесса создания новых сообщений... Некоторое время я всесторонне обыгрывал эту идею, после чего придумал следующий код. Единственное, в чем я не совсем уверен, это в способе использования данного шаблона, когда созданный нами производный класс используется в качестве аргумента:
> class CoinInsertedEvent : public TEvent<CoinInsertedEvent>
> Я никогда не видел, чтобы шаблон использовался таким образом, но теперь вижу, что это удобный способ выполнять проверку типов во время компиляции. Имеет ли такой способ использования шаблона какое-то специальное название?
>
> Другой положительный момент использования данного шаблона заключается в том, что интерфейс Handler автоматически генерируется как часть самого класса TEvent, что еще больше упрощает добавление новых событий:

```
#include <iostream.h>
#include <stdio.h>
#include <list>

using namespace std;

template <class T>
class TEvent {
public:
    class Handler {
    public:
        Handler () { TEvent<T>::register (this); }
        virtual int handleEvent(const T& t) = 0;
    };

    typedef list<Handler*> HandlerList;

    static void register (Handler* aHandler) {
        registry. push_back (aHandler) ;
    }
    static void notify (TEvent<T>* t) {
        HandlerList::iterator i;
        for (i = registry.begin(); i !=registry.end(); i++) {
```

```
                (*i)->handleEvent(*(T*) t);
            }
        }
}

void Notify () {T::notify(this);}
private:
    static HandlerList registry;
};
class CoinInsertedEvent :
    public TEvent<CoinInsertedEvent> {};
class CoinReleaseEvent :
    public TEvent<CoinReleaseEvent> {};
class ProductDispensedEvent :
    public TEvent< ProductDispensedEvent> {};
class CoinChanger :
    public CoinReleaseEvent::Handler,
    public ProductDispensedEvent::Handler {
public:
    int handleEvent (const ProductDispensedEvent& event) {
        cout << "Changer::Coin dispensed." <<endl; return 0;
    }
    int handleEvent (const CoinReleaseEvent& event) {
        cout << "Changer::Coin released." <<endl;
        return 0;
    }
};

TEvent<CoinInsertedEvent>::HandlerList
TEvent<CoinInsertedEvent>::registry;
TEvent<CoinReleaseEvent>::HandlerList
TEvent<CoinReleaseEvent>::registry;
TEvent<ProductDispensedEvent>::HandlerList
TEvent<ProductDispensedEvent>::registry;

int main (int, char**) {
    CoinReleaseEvent CoinReleaseEvent;
    CoinChanger coinChanger;
    ProductDispensedEvent ProductDispensedEvent;

    CoinReleaseEvent.Notify();
    ProductDispensedEvent.Notify();
```

Заметим, что шаблон TYPED MESSAGE содержит все, кроме механизма регистрами и уведомления из данной реализации. Браво, Поль!

Глава 5

Разработка шаблонов: семь правил успеха

Если объектно-ориентированная разработка кажется вам слишком сложным делом, попробуйте заняться разработкой шаблонов! Как математику, мне нравится думать о ней как об "интегрировании" объектно-ориентированного проектирования: это сумма бесконечно малых опытов по интервалу приложений. Хотя разработка шаблонов кажется гораздо сложнее, чем то, что я учил в курсе интегрального исчисления. Интегралы не влияют друг на друга, что позволяет вычислять их независимо (хотя умение вычислять один часто помогает вычислить другие). Шаблон, напротив, не может работать в вакууме — он обеспечивает решение только одной проблемы, поэтому он должен взаимодействовать с другими шаблонами. Следовательно, разработчику шаблонов следует рассматривать не один шаблон, а несколько, причем некоторые из них еще даже не описаны — и это только одна из трудностей, с которыми приходится сталкиваться в процессе разработки. Вот почему честолюбивый новичок должен получить всю возможную помощь от опытных разработчиков шаблонов.

Мы многое узнали о разработке шаблонов в процессе написания книги *Design Patterns* и все еще продолжаем учиться. В заключительной главе я хочу изложить приобретенный нами опыт в виде семи правил, которых мы придерживались, зачастую неосознанно, на протяжении многих лет работы над шаблонами. Эти правила помогут читателю гораздо быстрее научиться создавать шаблоны.

Правило 1. Выделять время для размышлений

Наиболее важная деятельность при написании шаблонов — *рефлексия.* Брюс Андерсон (Bruce Anderson), оказавший вначале заметное влияние на нашу работу, годами твердил об этом. Нужно периодически осмысливать то, что вы сделали: подумать о разработанных системах, о возникших проблемах, о том, как они решаются (или не решаются).

Такие отвлечения сложно представить себе в наши дни, когда сроки разработки максимально сжаты. Но осмысление критически важно! Нет лучшего способа настроить себя на творческий лад. Можно написать горы программного кода, но его количество не является мерой производительности. Признак хорошего проекта как раз в обратном: хороший проект невелик и элегантен, он вмещает многое в небольшом объеме, в нем все реализовано "раз и только раз", как любит говорить

Кент Бек (Kent Beck). Кроме того, хороший проект достаточно гибок, в нем практически отсутствуют крупные фрагменты кода.

Сейчас не принято брать один месяц в год для размышлений, но можно записывать свой опыт по частям. Столкнувшись с нетривиальной проблемой, постарайтесь сразу же ее описать, проанализируйте, в чем состоит ее сложность. Начните работу над проблемой, фиксируйте каждую попытку применить новый подход. Если попытка не увенчалась успехом, укажите, почему это произошло; успешную попытку также постарайтесь зафиксировать. Практически каждый разработчик может выделить пять процентов своего времени, чтобы описать полученный опыт — для этого нужна только дисциплинированность.

Если серьезно отнестись к данному вопросу, получится удивительный результат. Накопленный таким образом опыт и есть исходный материал для шаблонов. Конечно, многое еще предстоит сделать, но в ваших руках уже будут все основные ингредиенты.

Важно также ознакомиться с как можно большим числом систем, созданных другими разработчиками. Лучший способ это сделать — поработать с этими системами. Если же не хватает времени и средств, нужно хотя бы прочитать о них. Постарайтесь понять, какие проблемы призваны решить рассматриваемые системы; проанализируйте, каким образом они это делают. Изучите спецификации и документацию. Прочитайте статьи об исследовательских системах, просмотрите материалы OOPSLA и ECOOP. Другим полезным источником информации о проектировании и реализации можно назвать *Software - Practice & Experience.*

При изучении определенной системы постарайтесь выявить уже известные шаблоны, оцените, как найденные решения отличаются от решений в уже опубликованных шаблонах. Обратите внимание на новые проектные решения — они могут представлять новые шаблоны, но помните, что действительно новых проектных решений не так уж много (чаще используются вариации уже известных решений). Новое и/или уникальное решение не всегда имеет достаточно широкое применение, чтобы из него можно было создать шаблон.

Если вам действительно удалось найти нечто, что кажется новым, убедитесь, что это решение применимо в других ситуациях, прежде чем пытаться описать его в качестве шаблона. При написании книги *Design Patterns* "банда четырех" руководствовалась незыблемым правилом: найти *два* примера существования определенной проблемы и ее решения, прежде чем писать для нее шаблон. Это было чрезвычайно важное правило: мы исследовали незнакомую территорию и нужно было убедиться, что написанное нами основано на реальности, и мы предлагаем решения проблем, которые существуют не только в нашем воображении. Поэтому пришлось отказаться от многих шаблонов, которые казались достаточно привлекательными и потенциально полезными, но не имели реального использования.

Правило 2. Приверженность структуре

Если у вас есть исходный материал, как приступить к его описанию в форме шаблона?

Прежде всего не следует думать, что существует *только одна* форма шаблона. Одна форма не может удовлетворить всех. Некоторые предпочитают описательный стиль, как у Александера (Alexander), другим нравится более строгий подход, используемый в *Design Patterns,* третьи применяют совсем иные структуры. Тем не менее, все подходы имеют общий атрибут — *структуру.*

Есть одна фраза, каноническое высказывание Александера: "Шаблон — это решение определенной проблемы в определенном контексте", с которой согласны практически все, кто занимается шаблонами. Я буду настолько смел, что скажу иначе: шаблон — это *структурированное представление* решения определенной проблемы в определенном контексте. Шаблоны имеют распознаваемые части, которыми можно руководствоваться при их применении и сравнении. Это имя, описание проблемы и контекста, обоснование решения и само решение. Такую структуру, по сути, имеют шаблоны Александера. В наших шаблонах производится дальнейшая декомпозиция указанных фундаментальных элементов на более специальные, какими являются разделы "Применимость", "Участники" и "Последствия". На конференциях Pattern Languages of Programs (PLoP) [CS95, MRB98, VCK96] было представлено очень много вариаций на эти темы.

Таким образом, первым делом при попытке изложить шаблон на бумаге нужно определиться с его структурой. Чем больше информации содержит шаблон, тем важнее становится его структура. Целостная структура придает шаблонам информативность и позволяет сравнивать их друг с другом. Структура также помогает находить нужную информацию. Менее структурированный подход означает больше описаний, что неплохо для обычного чтения, но неприемлемо, когда шаблоны нужно сравнивать и ссылаться на их различные элементы.

После того как выбрана определенная структура, нужно неукоснительно следовать ей. Не нужно опасаться изменять структуру, просто в таком случае придется изменить ее во всех шаблонах, и это обходится тем дороже, чем более совершенны ваши шаблоны.

Правило 3. Как можно раньше и чаще рассматривать конкретные примеры

В наших шаблонах раздел "Мотивация" предваряет остальные разделы, поскольку люди лучше воспринимают концепции, если их сначала пояснить на конкретном примере, а затем уже перейти к более абстрактной форме изложения. Конкретный пример в разделе "Мотивация" дает читателю общее представление о проблеме и ее решении. Кроме того, в этом разделе (также на конкретном примере) демонстрируется, почему потерпели неудачу другие подходы к решению проблемы. После такого введения читатель лучше подготовлен к восприятию общего решения.

Чтобы изложение было конкретным, необходимо представить множество примеров из реальной жизни. Примеры не должны быть исключительным атрибутом раздела "Мотивация". Следует использовать примеры и контрпримеры повсюду в шаблоне для иллюстрации основных положений. Даже наиболее абстрактные разделы наших шаблонов ("Применимость", "Структура", "Участники" и "Взаимодействия") иногда содержат примеры. Так, разделы "Взаимодействия" некоторых шаблонов включают в себя диаграммы динамического взаимодействия объектов. Обращение к таким примерам при обсуждении абстрактных аспектов шаблона позволяет сохранить конкретность даже при анализе абстракций.

Другое важное правило можно сформулировать следующим образом: "Говорите всю правду". Это означает, что вы обязаны предупредить читателя о потенциальных ловушках предлагаемого шаблона. Слишком просто остановиться только на положительных его аспектах; гораздо сложнее указать на его недостатки и откровенно обсудить их. Нет шаблонов, свободных от недостатков, будь то дополнительные расходы,

неудовлетворительное поведение в определенных обстоятельствах или что-нибудь еще. Нужно убедиться, что читатели понимают, почему шаблон может не достигнуть намеченной цели.

Правило 4. Шаблоны должны быть разными и взаимно дополняющими

Существует тенденция, которой следует избегать при параллельной разработке нескольких шаблонов. По мере работы над шаблоном он увеличивается в объеме и обрастает деталями, поэтому легко забыть об остальных шаблонах. В результате различие между шаблонами стирается, и читателям становится сложно воспринимать их в совокупности. Шаблоны перекрываются в своих целях и охвате. Все, что кажется вполне понятным автору, вовсе не так понятно со стороны. Читателю будет трудно определить, когда использовать один шаблон, а когда другой, поскольку их различия неочевидны.

Поэтому следует делать шаблоны ортогональными, взаимно дополняющими и усиливающими друг друга. Надо постоянно спрашивать себя: "Чем шаблон X отличается от шаблона Y?" Если два шаблона решают одинаковые или близкие задачи, их, вероятно, можно объединить. Не стоит переживать, если два шаблона используют одинаковые иерархии классов. В объектно-ориентированном программировании существует множество способов использования относительно небольшого числа присущих ему механизмов. Зачастую одна и та же организация классов может дать существенно различные структуры объектов, которые решают весьма широкий спектр проблем. Поэтому принимая решение о различиях шаблонов, нужно руководствоваться их назначением, а не структурами классов, используемых при их реализации.

Хорошим способом обеспечить ортогональность и синергию шаблонов является ведение двух отдельных документов, в одном из которых собраны общие свойства шаблонов, а в другом — различия. В книге *Design Patterns* этой теме посвящено несколько разделов. Такое простое действие, как попытка в письменной форме объяснить взаимосвязи шаблонов, позволяло по-иному взглянуть на наши шаблоны и не однажды заставляло пересматривать некоторые из них.

Я сожалею только о том, что мы не уделяли должного внимания взаимосвязям шаблонов с самого начала. Я рекомендую вам начинать делать такие вспомогательные записи как можно раньше. Это может показаться излишним, особенно если шаблонов еще немного, но как только у вас появляется хотя бы два шаблона, возникает вероятность их пересечения. Сравнивая и противопоставляя свои шаблоны с самого начала, вы сможете сделать их отличными друг от друга и взаимно дополняющими.

Правило 5. Удачное представление

Качество шаблонов в немалой степени определяется тем, насколько хорошо они представлены. Можно создать лучший в мире шаблон, но он никому не поможет, если вы не сможете должным образом сообщить о нем.

Под "представлением" я подразумеваю две составляющие: стиль набора и стиль написания. Хороший набор — это профессиональный макет и графика, не говоря уже о качестве принтера. Используйте все доступные средства программного обеспечения (текстовый процессор, графический редактор и т.п.). Применяйте рисунки для иллюстрации основных положений (вы можете думать, что они вам не нужны, но это не так). В худшем случае рисунки позволят избежать монотонности, а в лучшем —

помогут понять то, что не в состоянии прояснить никакие объяснения. Не обязательно, чтобы рисунки были формальными диаграммами классов и объектов; зачастую неформальные рисунки и даже скетчи содержат не меньше информации. Если вы не обладаете "художественным даром", пригласите художника.

Хороший стиль написания еще важнее. Пишите ясно и непретенциозно. Лучше отдать предпочтение простому стилю, чем слишком академичному. Читатели очень хорошо воспринимают тон диалога. Ясность и простота восприятия важны в любом описании, но при написании шаблонов они важны вдвойне: концепция шаблонов достаточно нова, и предмет достаточно сложен для понимания. Постарайтесь сделать все возможное, чтобы шаблон был доступным.

Лучший способ научиться писать в диалоговом стиле — это попытаться что-то написать самому. Пишите так, как будто вы разговариваете со своими друзьями. Старайтесь избегать длинных предложений и разделов. Не бойтесь использовать сокращения. Главное, чтобы это звучало естественно.

На некотором этапе следует прочитать одну-две книги по стилю письма. Выбор весьма обширный. Три мои любимые книги — это White *The Elements of Style* [SW79] (кстати, ее организация напоминает серию шаблонов), Joseph M. Williams *Style: Ten Lessons in Clarity and Grace* [Williams85] и John R. Trimble *Writing with Style: Conversations on the Art of Writing* [Trimble75]. В этих книгах описаны приемы и методы ясного изложения мыслей. Они помогут вам усовершенствовать ваши шаблоны независимо от их технического содержимого.

Правило 6. Неустанные итерации

Вам не удастся создать шаблон с первого раза. Вы не сможете сделать его совершенным даже с десятого. Шаблон, вероятно, никогда нельзя считать идеальным, а процесс его создания — завершенным. Новизна данной области исследований не имеет значения. Даже если будет существовать множество примеров удачных шаблонов и книг, способных помочь вам в их создании, все равно процесс разработки шаблонов (как и любой другой процесс разработки) останется итеративным процессом.

Поэтому настройтесь на то, что вам придется неоднократно писать и переписывать ваши шаблоны. Не радуйтесь совершенству одного шаблона, пока не начнете работу над следующим. Помните, что шаблоны не существуют изолированно; они взаимодействуют друг с другом. Значительные изменения одного шаблона скорее всего повлияют на другие шаблоны. Как и во всяком итеративном процессе, на некотором этапе удается достигнуть определенной стабильности шаблонов, достаточной для того, чтобы другие люди могли читать, понимать и комментировать их.

Правило 7. Собирать и учитывать отклики

Сервантес был прав: "Чтобы проверить качество пудинга, нужно его съесть". Лакмусовой бумажкой для шаблона служит его использование. В действительности ни один шаблон нельзя считать проверенным, пока он не будет использован кем-то, кроме его автора. Шаблоны имеют удивительное свойство быть совершенно понятными людям, которые знакомы с проблемой и ее решением. Такие люди уже использовали данный шаблон неосознанно, они немедленно узнают его, даже если он не слишком

хорошо описан. Гораздо сложнее сделать так, чтобы шаблон могли понять те, кто ни когда не сталкивался с данной проблемой ранее. Поэтому необходимо собирать учитывать отклики именно таких людей.

Привлекайте к обсуждению проектов коллег, использующих ваши шаблоны, и сами принимайте участие в подобных дискуссиях. Ищите возможности использовать разработанные шаблоны в повседневной работе. Старайтесь как можно шире распро-странить информацию о своих шаблонах. Можно предложить их электронным кон-ференциям PLoP или опубликовать в изданиях типа *C++Report, Smalltalk Report, Java Report и Journal of Object-Oriented Programming.* Это позволит получить множество интересных откликов.

Когда комхментарии начнут поступать, приготовьтесь услышать худшее. Я даже затрудняюсь сказать, сколько раз я был шокирован, когда узнавал, что нечто, казавшееся абсолютно понятным мне, оказывалось совершенно запутанным для других. Негативная реакция может разочаровать, особенно вначале, когда вы наиболее уязвимы и когда наиболее вероятно получение отрицательных откликов. Хотя некоторая критика может оказаться несправедливой или быть следствием простого непонимания, большая ее часть, вероятно, будет правильной. Дайте вашим рецензентам возможность посомневаться, учтите недостатки, чтобы доставить им удовольствие. В результате получатся качественные шаблоны, которыми смогут воспользоваться многие люди.

Универсального рецепта нет

Применение описанных выше правил, как вы понимаете, не гарантирует разработчику шаблонов успеха. Их список не является исчерпывающим; в частности, Межарос (Meszaros) и Добл (Doble) пошли еще дальше [MD98]. Во всяком случае, эти правила помогут вам эффективно применять ваши усилия. Чем лучше будут ваши шаблоны, тем больше пользы они принесут.

Создание шаблонов требует огромных затрат времени и сил, поэтому не всякий может этим заниматься. Попробуйте написать один-два шаблона, и тогда станет ясно, способны ли вы к этому делу или нет. Я надеюсь, что со временем многие пользователи шаблонов станут их создателями, как в свое время многие пользователи языков программирования стали их разработчиками.

Библиография

[AIS+77] Alexander C, Ishikawa S., Silverstein M. с соавт. *A Pattern Language.* Oxford University Press, New York, 1977.

[ASC96] *Accredited Standards Committee. Working paper for draft proposed international standard for information systems- programming language* C++. Doc. No. X3J16/96-0225, WG21/N1043, December 2, 1996.

[Betz97] Betz M. *Письмо по электронной почте от 27 мая 1997 г.*

[Burchall95] Burchall L. *Письмо по электронной почте от 21 июня 1995 г.*

[BCC+96] Веек К., Coplien J., Crocker R. с соавт. *Industrial experience with design patterns.* Proceedings of the 18' International Conference jn Software Engineering (pp. 103— 114), Berlin, Germany, March 1996.

[BFV+96] Budinsky F., Finnie M., Vlissides J. с соавт. *Automatic Code Generation from Design Patterns.* IBM Systems Journal, 35(2):151-I7l, 1996. *http://www.almaden.ibm.com/ journal/'sj/budin/budinsky. html*

[BMR+96] Buschmann F., Meunier R., Rohnert H. с соавт. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons Ltd., Chichester, England, 1996.

[Coplien92] Coplien J. *Advanced* C++ *Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[Coplien95] Coplien J. *Curiously Recurring Template Patterns.* C++Report,

7(2):40-43, 1995. [Coplien96] Coplien J. *Software Patterns.* SIGS Books, New

York, 1996.

[CS95] Coplien J., Schmidt D. (под. ред.) *Pattern Languages of Program Design,* Asddison-Wesley, Reading, MA, 1995.

[CZ96] Clark C, Zino B. *Письмо по электронной почте от 28 октября 1996 г.*

[Forte97] Fort—Software, Inc. *Customizing Forte Express Applications.*

Oakland, CA, 1997.

[Fowler97] Fowler M. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Reading, MA, 1997.

[Gabriel95] Gabriel R. *Письмо по электронной почте от 14 апреля 1995 г.*

[Gamma91] Gamma E. *Object-Oriented Software Development Based on ET++: Design Patterns, Class Library, Tools (на нем. языке).* Диссертация на соискание ученой степени доктора философии, Цюрихский университет, Институт информатики, 1991.

[Gamma91] Gamma E. *Письмо по электронной почте от 8 марта 1995* г.

[GoF95] Gamma E., Helm R., Johnson R. с соавт. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[Hay9б] Hay D. *Data Model Patterns: Conventions of Thought.* Dorset House, New York, 1996. [Henney96] Henney K. *Письмо по электронной почте от 15 сентября 1996 г.*

[HJE95] Huni H.Johnson R., Engel R. *A Framework for Network Protocol Software.* OOPSLA'95 Conference Proceedings (опубликовано как *ACM SIGPLAN Notices),* 30(10):358-369, 1995.

[Java97] JavaSoft, *Inc. Java Development Kit Version 1.1.* Mountain View, CA, 1997.

[Kotula96] Kotula J. *Discovering Patterns: An Industry Report.* Software — Practice *8c* Experience, 26(11):1261-1276, 1996.

[KP88] Krasner G., Pope S. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, l(3):26-49, 1988.

[LVC89] Linton M., Vlissides J., Calder P. *Composing User Interfaces with InterViezvs.* Computer, 22(2):9-22, 1989.

[Martin97] Martin R. *Письмо по электронной почте от 24 июля 1997 г.*

[McCosker97] McCosker M. *Письмо по электронной почте от 4 марта 1997 г.*

[Meyers95] Meyers S. *Письмо по электронной почте от 31 января 1995 г.*

[MD98] Meszaros G., DobleJ. *A Pattern Language for Pattern Writing.* В сборнике: [MRB98].

[MRB98] Martin R., Riehle D., Buschmann (под ред.). *Pattern Languages of Program Design 3.* Addison-Wesley, Reading, MA, 1998.

[PeIIetier97] Pelletier P. *Письмо по электронной почте от 22 июня 1997 г.*

[Peierls96] Peierls T. *Письмо по электронной почте от 16 февраля 1996 г.*

[Prechtelt97] Prechtelt L. *An Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation.* Technical Report 9/1997, University of Karlsruhe, Germany, June 1997.

[PUS97] Prechtelt L., Unger B., Schmidt D. *Replication of the First Controlled Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation.* Technical Report WUCS-97—34, Washington University, Department of Computer Science, St. Louis, December, 1997.

[PD96] *patterns-discussion@cs.uiuc.edu,* December 12, 1996.

[Schmid95] Schmid H. *Creating the Architecture of a Manufacturing Framework by Design Patterns. OOPSLA '95 Conference Proceedings* (опубликовано как *ACM SIGPLAN Notices),* 30(10:370-384, 1995.

[Schmid96a] Schmid H. *Письмо по электронной почте от 2 января 1996 г.*

[Schmid96b] Schmid H. *Письмо по электронной почте от 9 января 1996 г.*

[Schmid96c] Schmid H. *Письмо по электронной почте от 7 февраля 1996 г.*

[Schmid96d] Schmid H. *Письмо по электронной почте от 8 февраля 1996 г*

[Siegel96] SiegelJ. *CORBA Fundamentals and Programming.* Wiley, New York, 1996.

[SH98] Schmidt D., Harrison T. *Double-Checked Locking* В работе [MRB98].

[SV97] Schmidt D., Vinovski S. *The OMG Event Object Service.* C++ Report, 9(2):37-46, 52, 1997.

[SW79] Strunk W., White E. *The Elements of Style* (3 изд.). Macmillan, New York, 1979.

[Trimble75] Trimble J. *Writing with Style: Conversations on the Art of Writing.* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[VanCamp96] Van Camp D. *Письмо по электронной почте от 23 сентября 1996 г.* . [Vlissides96] Vlissides J. *Generation Gap.* C++ Report, 8(10):12-18, 1996.

[VCK96] Vlissides J., Coplien J., Kerth N. (под ред.) *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA, 1996.

[VT91] Vlissides J., Tang S. *A Unidraw-based User Interface Builder.* Proceedings of the ACM SIGGRAPH Fourth Annual Symposium on User Interface Software and Technology (pp. 201-210). Hilton Head, SC, November 1991.

[Wendland97] Wendland G. *Письмо по электронной почте от 10 января 1997 г.*

[Williams85] Williams J. *Style: Ten Lessons in Clarity and Grace* (2 изд.) Scott, Forsman and Co., Glenview, IL, 1985.

Предметный указатель

D

Design Patterns, 9; 13

P

PLoP, 18; 19; 127
Pull-модель, 100
Push-модель, 100

А

Абстрактные операции, 33
Абстрактный класс, 37
Аутентификация, 46

Б

Банда четырех, 9

В

Висящие ссылки, 58; 62

Г

Группа пользователей, 51

Д

Двунаправленное соответствие, 52
Деструктор чисто виртуальный, 98

З

Защита
  от записи, 40
  от чтения, 41
Защищенный деструктор, 41; 58
Защищенный узел удаление, 41

И

Инверсия управления, 44
Инкапсуляция, 46
Использование friend, 60; 92

К

Каркас, 45; 89
  стирание типов, 99
Класс Singleton
  деструктор, 57
Курсор, 91
  создание экземпляров, 94
  удаление, 95

М

Меченый бит, 70
Моделирование событий, 89
  восстановление типа, 90
  доставка, 99; 100; 108
    децентрализация, 100
  механизм регистрации, 102; 106; 108
  обработка с сохранением типа, 104
  расширения, 105
  сохранение типа, 105

Н

Несовместимость
  семантическая, 82
  синтаксическая, 82

О

Обходы, 70

Объект
  двухсторонний, 92
Объект Singleton
    неявная деструкция, 61
    удаление, 58
    удаление зависимых экземпляров, 61
    функция atexit (), 62
Операция-уловитель, 38; 72

П

Пароль, 46
Перегрузка имени функции, 38
Поведение по умолчанию, 38
Пользователь узла, 45
Приведение типов, 35
Приложения
  многопоточные, 63
  однопоточные, 63
Принцип Голливуда, 45
Проверка типа
  динамическая, 73

Р

Регистрационное имя, 45

С

Сборка мусора, 95
Символическая связь, 29
События, 105
Создание объектов User, 46
Стирание типов, 93
Структура, 91

У

Утечка памяти, 63; 95

Ц

Централизация ответственности, 52

Ш

Шаблон
  ABSTRACT FACTORY, 47
  BUILDER, 47
  COMMAND, 121
  COMPOSITE, 23; 34; 51
    назначение, 23
    преимущества и недостатки, 24
    структура, 24
    участники, 24
  FACTORY METHOD, 47; 90; 94
  GENERATION GAP, 75; 79
    известные применения, 86
    класс расширений, 79
    мотивация, 76
    назначение, 75
    недостатки, 83
    преимущества, 82
    применимость, 80
    реализация, 83
    сердцевинный класс, 79
    структура, 81
    участники, 81
  ITERATOR, 91
  MEDIATOR, 52
    недостатки, 54
    объект-посредник, 53
    объекты-коллеги, 52
  MEMENTO, 91; 95; 122
    назначение, 91
    участники, 92
  MULTICAST, 104; 105; 107
    вариации, 116
    мотивация, 105
    назначение, 115
    применимость, 111
    участники, 110
  OBSERVER, 32; 65; 70; 119
    вариации, 116
    избыточность, 68

недостатки, 67
преимущества, 67
связь с MULTICAST, 111
участники, 65
PROTOTYPE, 47; 90
PROXY, 30; 34
назначение, 30
участники, 31
SINGLETON, 47; 54; 57
назначение, 57
удаление экземпляров Singleton, 57 TEMPLATE METHOD, 42; 45
инверсия управления, 44
назначение, 42
элементарные операции, 43
TYPED MESSAGE, 119
мотивация, 119
назначение, 119
применимость, 119
участники, 120
VISITOR, 35; 71; 74; 91
использование using, 39
назначение, 35
условия применения, 39
участники, 36
определение, 14; 127
Шаблоны проектирования
выбор, 30; 47
плотная композиция, 34
поведенческие, 47
порождающие, 47
структурные, 47