> Ни один человек не обладает изначально правом командовать своими ближними.
>
> Дидро

Из всех описанных в последние годы шаблонов проектирования меня больше всего впечатляет структура Command — одна из самых простых и элегантных. Но как мы увидим далее, видимая простота часто бывает обманчивой. Поэтому диапазон применения шаблона Command практически не имеет ограничений.

Простота применения шаблона Command, как показано на рис. 13.1, является почти курьезной. Листинг 13.1 также не является слишком "серьезным". Может показаться абсурдным, что наш шаблон состоит из интерфейса с одним методом.

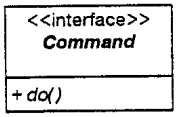


Рис. 13.1. Шаблон Command

Листинг 13.1. Command.java

```
public interface Command
{
  public void do();
}
```

Однако на самом деле данный шаблон обладает очень интересной особенностью. Именно в ней заключается его "скрытая" сложность. Большинство классов ассоциируются с набором методов, а также с соответствующим им множеством переменных. Шаблон Command устроен по-другому. Можно сказать, что он включает в себя функции без переменных.

Если строго придерживаться основ объектно-ориентированного программирования, то получается, что такая система приводит к функциональной декомпозиции. Она поднимает значимость и свойства функции до уровня класса. Неслыханно! Однако при столкновении двух этих принципов происходят вещи, заслуживающие пристального внимания и изучения.

Простые команды

Несколько лет назад я консультировал представителей фирмы, производящей копировальную технику (ксероксы). Я помогал одной из команд разработчиков в проектировании и внедрении встроенной программы, работающей в режиме реального времени, которая была предназначена для нового поколения ксероксов.

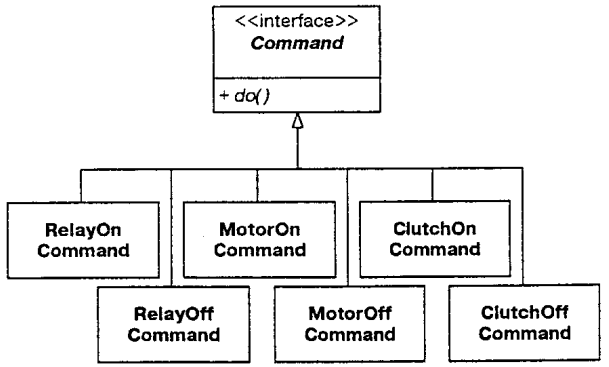


Рис. 13.2. Некоторые простые команды программы управления ксероксом

Мы буквально натолкнулись на идею использования шаблона Command для контроля аппаратных средств. Была создана иерархия, похожая на ту, что изображена на рис. 13.2.

Роль данных классов очевидна. Когда вы вызываете функцию do() из класса RelayOnCornmand, она включает некоторые реле. При вызове do() из MotorOffCommand происходит отключение двигателя. Адрес двигателя или реле включается в объект как аргумент.

Имея в своем распоряжении такую структуру, мы можем оперировать объектами Command в пределах всей системы и применять к ним функцию do(), не задаваясь при этом вопросом какой именно тип объекта Command будет использован. Это ведет к дальнейшим любопытным упрощениям.

Данная система управляется событиями. Реле срабатывали и отключались, двигатели включались и выключались, муфты сцеплялись и расцеплялись в зависимости от определенных событий, происходящих в системе. Многие из этих событий обнаруживаются сенсорами. Например, когда оптический сенсор "видит", что лист бумаги достиг определенной точки, необходимо включить соответствующий захват. Мы обеспечили это путем простой привязки ClutchOnCommand к объекту, который контролировал данный оптический сенсор (рис. 13.3).

Такая простая структура обладает массой преимуществ. Объект Sensor "не знает", что он делает. При обнаружении события он просто вызывает функцию

Рис. 13.3. Программное обеспечение сенсорного контроля

do() из объекта Command, к которому она относится. То есть, все объекты Sensors не должны быть связаны со всеми захватами или реле. Им также не требуется информация о структуре и пути листа бумаги в аппарате. Функция сенсора в высшей степени проста.

Все сложности, связанные с идентификацией нужных реле и их связи с сенсорами, делегируются функции инициализации. В определенный момент при инициализации всей системы каждый объект Sensor связывается с конкретным объектом Command. Это позволяет поместить всю схему *проводки*[1] *(логические связи между сенсорами и "командами")* в одном месте и выделить ее из общего кода системы. При этом представится возможность создать простой текстовый файл, содержащий описание всех связей между объектами Sensors и Commands. Программа инициализации будет считывать этот файл, и конфигурировать систему соответствующим образом. Т.е. вся *проводка* системы будет описываться вне общего тела программы и при ее изменении не придется перекомпилировать весь код.

После включения *понятия* команды описанная схема позволяет нам "отделить" логические связи от используемых при этом устройств. Благодаря этому достигаются серьезные преимущества.

Транзакции

Еще одно применение шаблона Command, удобное при работе с программой расчета зарплаты, — создание и выполнение транзакций. Предположим, например, что мы пишем программу, поддерживающую базу данных всех сотрудников компании (рис. 13.4). Существует множество операций, которые пользователи могут выполнять при работе с базой. Они могут добавлять в нее информацию о новых работниках, удалять информацию о старых или изменять атрибуты существующих сотрудников.

Как только пользователь решает добавить запись о новом работнике, он должен указать всю информацию, необходимую для успешного создания новой записи. Прежде чем оперировать полученной информацией, система должна проверить ее на предмет синтаксической и семантической корректности. В этой работе может помочь шаблон Command. Объект Command в этом случае служит в качестве хранилища непроверенных данных, использует методы их проверки, а затем выполняет транзакцию.

К примеру, обратите внимание на рис. 13.5. Объект AddEmployeeTransaction содержит такие же поля данных, как и Employee. В нем также присутствует указатель на объект payClassification. Эти поля и объект создаются на основе данных, введенных пользователем при добавлении записи о новом сотруднике.

[1] Логические взаимосвязи между Sensors и Commands.

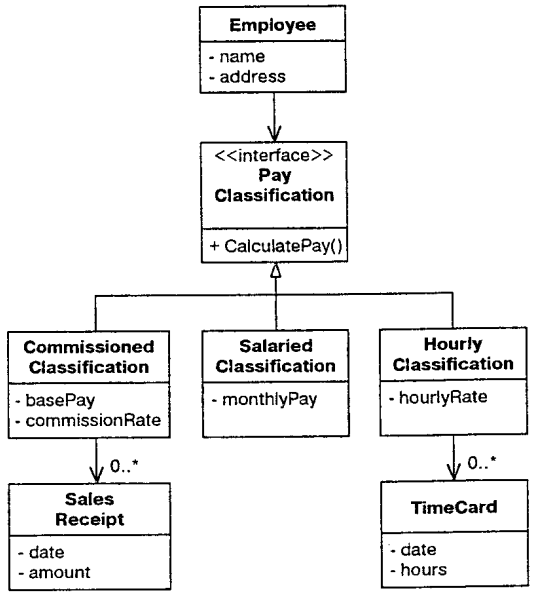


Рис. 13.4. База данных сотрудников

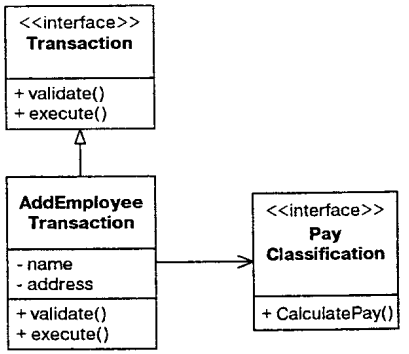


Рис. 13.5. Транзакция AddEmployee

Метод проверки корректности данных обрабатывает все данные без исключения. Он проверяет их синтаксическую и семантическую корректность. Можно даже проверить, не противоречат ли новые данные уже существующим. К примеру, есть возможность удостовериться, что записи о данном работнике в базе еще не существует (исключить дублирование данных).

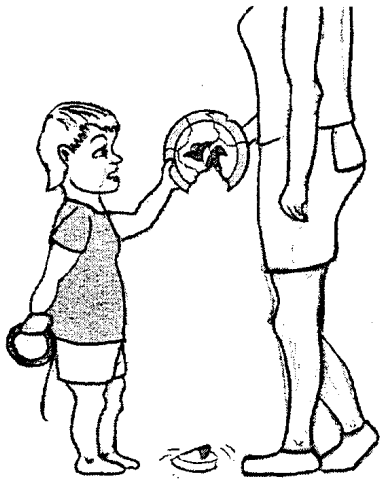
После завершения проверки происходит внесение информации в базу данных. В нашем простом примере новый объект Employee будет создан и загружен

с использованием полей данных объекта AddEmployeeTransaction. Объект PayClassification будет перемещен или скопирован в Employee.

Физическое и временное разделение связей

Основное преимущество данной технологии заключается в разделении кода, который получает данные от пользователя, кода, который проверяет их (и оперирует ими) и всеми остальными объектами программы. К примеру, для ввода данных о новых сотрудниках можно использовать диалоговые окна графического интерфейса. При этом нет смысла включать в код интерфейса алгоритмы проверки и загрузки этих данных — в случае подобного объединения нельзя будет использовать данные алгоритмы при работе с другими интерфейсами. Путем выделения алгоритмов в отдельный класс AddEmployeeTransaction, мы физически отделяем их код от интерфейса. Более того, мы отделяем код, управляющий работой базы, от остальных объектов.

Временное разделение



Можно также разделить код проверки и загрузки (обработки) другим способом. При вводе данных необязательно сразу же вызывать процедуры проверки и загрузки их в базу. Объекты могут быть внесены в отдельный список, а операции над ними можно произвести позже.

Допустим, что мы работаем с базой данных, которая должна оставаться неизменной в течение дня. Изменения могут вноситься только в промежуток времени между 24.00 и 01.00. Дожидаться полуночи и пытаться успеть ввести все данные и команды за час – не самое лучшее решение. Гораздо удобнее вносить данные по мере их поступления, чтобы затем они хранились в определенном месте (после проверки), а после полуночи автоматически загружались в базу. Схема command предоставляет нам такую возможность.

Отмена действия (UNDO)

На рис. 13.6 в шаблон Command добавлен метод undo(). Он основан на следующем принципе: если метод do() производного класса Command можно использовать для запоминания деталей операций, им же и выполняемым, то ме-

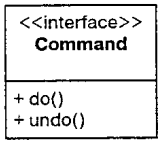


Рис. 13.6. Отмена действия в шаблоне Command

тод undo() можно использовать для "отката" (отмены предыдущих действий) и возврата системы к исходному состоянию.

Представим себе, к примеру, приложение, которое позволяет пользователю рисовать на экране геометрические фигуры. На панели инструментов находятся кнопки, дающие пользователю возможность рисовать окружности, квадраты, треугольники и так далее. Предположим, что пользователь нажимает на кнопку *"нарисовать окружность"*. Система создает объект DrawCircleCornmand, а затем вызывает для него функцию do(). DrawCircleCommand отслеживает перемещения мыши пользователем и ожидает нажатия в рабочей области окна ("окне рисования"). Когда пользователь щелкает мышью, DrawCircleCommand регистрирует его, устанавливая точку щелчка в качестве центра окружности, и рисует анимированный круг, радиус которого изменяется вместе с перемещением мыши. Когда пользователь снова щелкает мышью, объект DrawCircleCommand перестает выводить на экран анимированную окружность и добавляет новый объект к списку форм, отображаемых на экране. Он также сохраняет идентификатор новой окружности в виде переменной, которая затем возвращается методом do(). После этого система помещает DrawCirlceCommand в стек завершенных (выполненных) команд.

Через какое-то время пользователь щелкает на кнопке отмены действия на панели инструментов. Система обращается к данным из стека и вызывает функцию undo() результирующего объекта Command, при получении сообщения от undo() объект DrawCircleCommand удаляет окружность, сверяясь со списком идентификаторов объектов, отображаемых на экране.

Используя такую технику, вы можете легко встроить функцию отмены предыдущих действий практически в любое приложение. Код, отменяющий команду, всегда сходен с кодом, запускающим ее на выполнение.

Шаблон Active Object

Одним из моих любимых применений шаблона Command является реализация шаблона Active Object[2]. Это очень старый способ обеспечить управление

[2][Lavender96].

сразу несколькими потоками. В той или иной форме он использовался при создании многозадачных ядер для множества систем, применяемых в промышленности.

В основе данной технологии лежит очень простой принцип. Обратим внимание на листинги 13.2 и 13.3. Объект `ActiveObjectEngine` поддерживает связный список объектов `Command`. Пользователи могут добавлять новые команды в основную программу или вызывать функцию `run()`. Функция `run()` просто обрабатывает связный список, выполняя и затем удаляя каждую команду.

Листинг 13.2. `ActiveObjectEngine.Java`

```
import java.util.LinkedList;
import java.util.Iterator;

public class ActiveObjectEngine
{
  LinkedList itsCommands = new LinkedList();

  public void addCommand(Command c)
  {
    itsCommands.add(c);
  }

  public void run()
  {
    while (!itsCommands.isEmpty())
    {
      Command c = (Command) itsCommands.getFirst();
      itsCommands.removeFirst();
      c.execute();
    }
  }
}
```

Листинг 13.3. `Command.java`

```
public interface Command
{
  public void execute() throws Exception;
}
```

На первый взгляд, этот код не производит сильного впечатления. Но представим, что произойдет, если один из объектов `Command` в связном списке скопирует сам себя и поместит копию обратно в список. Тогда список никогда не опустеет, а функция `run()` никогда не завершит работу.

Обратим внимание на тестовый случай, код которого приведен в листинге 13.4. Программа создает нечто под названием `sleepCommand`. Кроме того,

перед созданием объекта `SleepCommand` имеет место задержка в 1000 миллисекунд. Затем `SleepCommand` помещается в `ActiveObjectEngine`. При вызове функции `run()` подразумевается, что прошло нужное количество времени.

Листинг 13.4. `TestSleepCommand.java`

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class TestSleepCommand extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"TestSleepCommand"});
  }

  public TestSleepCommand(String name)
  {
    super(name);
  }

  private boolean commandExecuted = false;

  public void testSleep() throws Exception
  {
    Command wakeup = new Command()
    {
      public void execute() {commandExecuted = true;}
    };
    ActiveObjectEngine e = new ActiveObjectEngine();
    SleepCommand c = new SleepCommand(1000,e,wakeup);
    e.addCommand(c);
    long start = System.currentTimeMillis();
    e.run();
    long stop = System.currentTimeMillis();
    long sleepTime = (stop-start);
    assert("SleepTime " + sleepTime + " expected > 1000",
           sleepTime > 1000);
    assert("SleepTime " + sleepTime + " expected < 1100",
           sleepTime < 1100);
    assert("Command Executed", commandExecuted);
  }
}
```

Давайте рассмотрим этот тестовый случай более внимательно. Конструктор (функция-член класса с тем же именем, что и сам класс, создающая и инициализирующая объект данного класса) объекта `sleepCommand` содержит три аргумента. Первый аргумент — время задержки в миллисекундах. Второй — `ActiveObjectEngine`, внутри которого команда будет запущена на исполне-

ние. И наконец, существует третий командный объект-"будильник". Смысл его заключается в том, что SleepCommand будет ждать определенное время (указанное в миллисекундах), а затем даст команду пробуждения системы.

Листинг 13.5 представляет реализацию объекта SleepCommand. При выполнении программы SleepCommand проверяет, не был ли он запущен ранее. Если нет, то объект записывает время старта. Если время задержки еще не прошло, он помещает себя обратно в ActiveObjectEngine. По прошествии указанного времени в ActiveObjectEngine помещается wakeup.

Листинг 13.5. SleepCommand.java

```
public class SleepCommand implements Command
{
  private Command wakeupCommand = null;
  private ActiveObjectEngine engine = null;
  private long sleepTime = 0;
  private long startTime = 0;
  private boolean started = false;

  public SleepCommand(long milliseconds, ActiveObjectEngine e,
                      Command wakeupCommand)
  {
    sleepTime = milliseconds;
    engine = e;
    this.wakeupCommand = wakeupCommand;
  }

  public void execute() throws Exception
  {
    long currentTime = System.currentTimeMillis();
    if (!started)
    {
      started = true;
      startTime = currentTime;
      engine.addCommand(this);
    }
    else if ((currentTime - startTime) < sleepTime)
    {
      engine.addCommand(this);
    }
    else
    {
      engine.addCommand(wakeupCommand);
    }
  }
}
```

Можно провести аналогию между данной программой и многопоточным приложением, ожидающим определенного события. Когда поток в многопоточной

программе ожидает конкретного события, он часто запускает процедуру системного вызова, блокирующую данный поток до тех пор, пока ожидаемое событие не произойдет. Программа, код которой представлен в листинге 13.5, не обладает функцией блокирования. Вместо этого, если ожидаемое событие ((currentTime - startTime) < sleepTime) не произошло, она просто помещает себя обратно в ActiveObjectEngine.

При создании многопоточных систем использовалось множество вариантов подобной технологии, в настоящее время она не менее популярна. Потоки такого типа называются *"исполняемыми до завершения" (run-to-completion)* заданиями (RTC), поскольку каждый экземпляр Command исполняется, а затем завершает свою работу, и только после этого запускается следующий такой объект. Само название указывает на то, что объекты Command не блокируют друг друга.

Тот факт, что отдельные экземпляры Command исполняются по очереди, позволяет RTC-потокам использовать стек исполняемой программы. В отличие от потоков данных в обычной многопоточной системе, совсем не обязательно определять или выделять отдельный стек для каждого потока. Это может оказаться большим преимуществом, если память системы ограничена, а потоков данных должно быть очень много.

В продолжение нашего примера рассмотрим листинг 13.6, где приводится код простой программы, использующей объект sleepCommand и отображающей поведение многопоточной системы. Программа называется DelayedTyper.

Листинг 13.6. DelayedTyper.java

```
public class DelayedTyper implements Command
{
  private long itsDelay;
  private char itsChar;
  private static ActiveObjectEngine engine =
    new ActiveObjectEngine();
  private static boolean stop = false;

  public static void main(String args[])
  {
    engine.addCommand(new DelayedTyper(100,'1'));
    engine.addCommand(new DelayedTyper(300,'3'));
    engine.addCommand(new DelayedTyper(500,'5'));
    engine.addCommand(new DelayedTyper(700,'7'));

    Command stopCommand = new Command()
    {
      public void execute() {stop=true;}
    };

    engine.addCommand(
      new SleepCommand(20000,engine,stopCommand));
```

```
    engine.run();
  }

  public DelayedTyper(long delay, char c)
  {
    itsDelay = delay;
    itsChar = c;
  }

  public void execute() throws Exception
  {
    System.out.print(itsChar);
    if(!stop)
    delayAndRepeat();
  }

  private void delayAndRepeat() throws Exception
  {
    engine.addCommand(new SleepCommand(itsDelay,engine,this);
  }
}
```

Обратите внимание, что `DelayedTyper` включает в себя `Command`. Метод `Execute` просто выводит на печать символ, введенный в структуру, проверяет наличие флага `stop` и, если он не установлен, активизирует `delayAndRepeat`. Метод `delayAndRepeat` создает объект `SleepCommand`, используя задержку, величина которой передается инструкции. Затем `SleepCommand` внедряется в `ActiveObjectEngine`.

Поведение объекта `Command` легко предсказать. Он просто запускает цикл, периодически выводящий на печать определенный символ и выжидающий затем заданное время. Цикл прекращается в случае установки флага `stop`.

Основная программа запускает сразу несколько экземпляров `DelayedTyper` в `ActiveObjectEngine`, каждый из которых выводит на печать свой собственный символ и обладает своим временем задержки. Затем она активизирует модуль `SleepCommand`, который устанавливает флаг `stop` через некоторое время. Результатом работы программы является простая строка, состоящая из символов '1', '2', '3', '5' и '7'. Повторный запуск выводит на экран похожую строку, символы в которой будут расположены в другом порядке Ниже показаны результаты двух запусков программы:

```
13571131151137111315113171513111315173111135111371153111135 7...
13571113151317113151131171351113115173111315113171135111311 7...
```

Эти строки отличаются друг от друга потому, что генератор тактовой частоты процессора и генератор импульсов времени не могут быть идеально синхронизированы. Этот вид недетерминированного поведения является "визитной карточкой" многопоточных систем.

Недетерминированное поведение системы зачастую является настоящим "проклятием для разработчика". Любой, кто работал со встроенными системами реального времени, знает, насколько трудно бывает отлаживать программу с недетерминированным поведением.

Резюме

Простота шаблона `Command` объясняется его гибкостью и универсальностью. Этот шаблон имеет массу применений — от управления базами данных до контроля за аппаратными средствами, от работы с ядром многопоточной системы до создания инструментов для графического интерфейса.

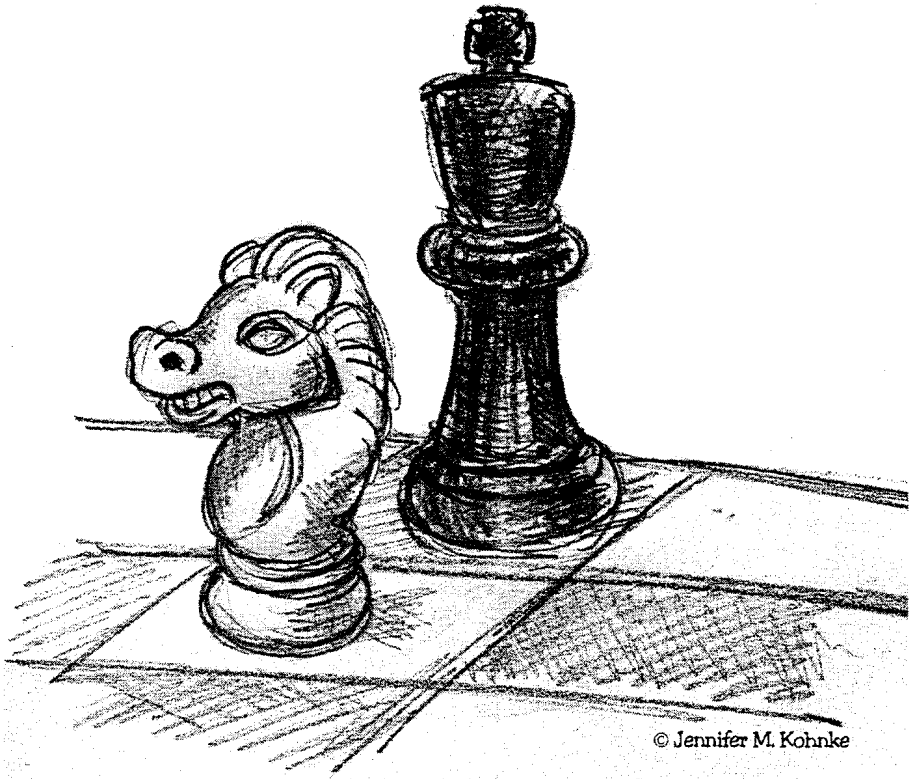
Многие полагают, что шаблон `Command` плохо вписывается в парадигму объектно-ориентированного программирования, поскольку в нем функции ставятся впереди классов. Иногда это действительно так, но в реальности такая технология вполне может оказаться полезной для любого разработчика программ.

Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Lavender R. G., Schmidt D. C. *Active Object: An Object Behavioral Pattern for Concurrent Programming*, in "Pattern Languages of Program Design" Reading, MA: Addison-Wesley, 1996.

14

Шаблоны Template Method и Strategy: наследование и делегирование



Лучшая стратегия в жизни — усердие.

Китайская пословица

Вспомните начало 90-х годов прошлого века ("детство" объектно-ориентированного программирования), когда все были увлечены идеей наследования. Следствия этого увлечения были достаточно серьезными. Благодаря наследованию можно легко создавать *различные программы*! Например, предположим, что имеется некоторый класс, который содержит набор полезных свойств. В этом случае можно создать подкласс, исключив свойства, которые не представляют особого интереса. Появляется возможность повторного использования кода путем простого наследования! При этом можно выполнять всеобъемлющие таксономии программных структур, отделяя каждый уровень повторно используемого кода от верхних уровней. В результате перед нами открывается новый мир.

Как и с большинством "других новых миров", порой кажется, что нам предоставляются неограниченные возможности. К 1995 году стало понятно, что методом наследования очень часто злоупотребляют, причем последствия подобного злоупотребления могут быть очень серьезными. Согласно высказыванию Гамма (Gamma), Хелма (Helm), Джонсона (Johnson) и Виссиди (Vlissides), "*хорошо выполненная композиция объектов лучше наследования классов*"[1]. Поэтому следует сократить использование наследования, заменяя ее композицией или делегированием.

В настоящей главе рассматриваются два шаблона, которые позволяют лучше понять различия между наследованием и делегированием. Достаточно часто шаблоны `Template Method` и `Strategy` позволяют решать подобные проблемы, а также являются взаимозаменяемыми. Следует отметить, что шаблон `Template Method` применяет технологию наследования, а шаблон `Strategy` — делегирование.

Шаблоны `Template Method` и `Strategy` позволяют решить проблемы, связанные с отделением обобщенного алгоритма от детализированного содержимого. Весьма часто именно эти шаблоны оказываются полезными в процессе разработки программного проекта. В нашем распоряжении окажется обобщенный алгоритм, который может применяться во многих случаях. В целях соответствия принципу инверсии зависимостей (DIP, Dependency-Inversion Principle) следует убедиться в том, что обобщенный алгоритм не зависит от детализированной реализации. Точнее, обобщенный алгоритм и его детализированная реализация должны зависеть от абстракций.

[1] [GOF95], с. 20.

Шаблон Template Method

Вспомните все программы, написанные вами в течение жизни. Не правда ли, что многие из них содержат следующую фундаментальную циклическую структуру (main).

```
Initialize();
while (!done()) // цикл main
{
  Idle(); // выполнение неких действий.
}
Cleanup();
```

Сначала инициализируется приложение. Затем вводится основной цикл. В теле этого цикла определяются все действия, выполняемые программой. Например, можно обрабатывать события, связанные с графическим интерфейсом пользователя, или обрабатывать записи в базе данных. После завершения перечисленных действий осуществляется выход из основного цикла, но перед выходом осуществляется очистка.

Эта структура носит настолько обобщенный характер, что имеет смысл поместить ее в отдельный класс, именованный `Application`. Затем этот класс может повторно использоваться при создании каждой новой программы. Прекрасная идея! Ведь нет ничего противнее, чем каждый раз переписывать цикл[2]!

Например, обратим пристальное внимание на листинг 14.1. Здесь присутствуют все элементы стандартной программы. Инициализируются переменные `InputStreamReader` и `BufferedReader`. Здесь мы имеем дело с основным циклом, в процессе выполнения которого считываются значения температуры, присвоенные переменной `BufferedReader` (по шкале Фаренгейта), которые затем преобразуются с использованием шкалы Цельсия. После завершения выполнения программы печатается соответствующее сообщение.

Листинг 14.1. `ftoc raw`

```
import java.io.*;
public class ftocraw
{
  public static void main(String[] args) throws Exception
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    boolean done = false;
    while (!done)
    {
      String fahrString = br.readLine();
      if (fahrString == null || fahrString.length() == 0)
        done = true;
      else
      {
        double fahr = Double.parseDouble(fahrString);
        double celcius = 5.0/9.0*(fahr-32);
        System.out.println("F=" + fahr + ", C=" + celcius);
      }
    }
    System.out.println("ftoc exit");
  }
}
```

[2] Я с удовольствием поделюсь с вами идеями, дабы облегчить ваше существование.

Описанная программа включает все элементы циклической структуры main. Сначала выполняется инициализация, затем реализуется цикл main, после чего выполняется очистка и выход из программы.

Можно отделить эту фундаментальную структуру от программы `ftoc` с помощью шаблона `Template Method`. Благодаря применению этого метода весь обобщенный код помещается в реализованный метод для абстрактного базового класса. Этот метод перехватывает обобщенный алгоритм, но обработку всех деталей передает абстрактным методам базового класса.

Так, например, можно перехватить циклическую структуру main в абстрактном базовом классе `Application` (листинг 14.2).

Листинг 14.2. `Application.java`

```
public abstract class Application
{
  private boolean isDone = false;

  protected abstract void init();
  protected abstract void idle();
  protected abstract void cleanup();

  protected void setDone()
  {isDone = true;}

  protected boolean done()
  {return isDone;}

  public void run()
  {
    init();
    while(!done())
      idle();
    cleanup();
  }
}
```

Этот класс реализует описание обобщенного приложения, включающего цикл main. Этот цикл можно видеть в реализованной функции `run`. Нетрудно заметить, что вся работа передается абстрактным методам `init`, `idle` и `cleanup`. Метод `init` выполняет необходимую инициализацию. Метод `idle` выполняет основную работу в программе и может вызываться несколько раз до тех пор, пока не будет вызван `setDone`. Метод `cleanup` выполняет все, что требуется перед выходом из программы.

Можно также переписать класс `ftoc`, выполнив наследование из `Application` и просто указав параметры для абстрактных методов. Пример реализации на практике подобной технологии приводится в листинге 14.3.

Листинг 14.3. `ftocTemplateMethod.java`

```
import java.io.*;
public class ftocTemplateMethod extends Application
{
  private InputStreamReader isr;
  private BufferedReader br;

  public static void main(String[] args) throws Exception
  {
    (new ftocTemplateMethod()).run();
  }

  protected void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  protected void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() ==0)
      setDone();
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.println("F=" + fahr + ", C=" + celcius);
    }
  }

  protected void cleanup()
  {
    System.out.println("ftoc exit");
  }

  private String readLineAndReturnNullIfError()
  {
    String s;
    try
    {
      s = br.readLine();
    }
    catch(IOException e)
    {
      s = null;
    }
    return s;
  }
}
```


Обработка исключений занимает некоторое время, хотя при этом упрощается процесс рассмотрения того, как прежнее приложение `ftoc` преобразуется с помощью шаблона `Template Method`.

Не злоупотребляйте шаблонами

Наверное, вы уже подумали следующее: *"И это он серьезно? Неужели он думает, что я буду использовать класс* `Application` *при разработке новых приложений? В любом случае это меня не сильно касается и, как мне кажется, вся проблема просто "высосана из пальца".*

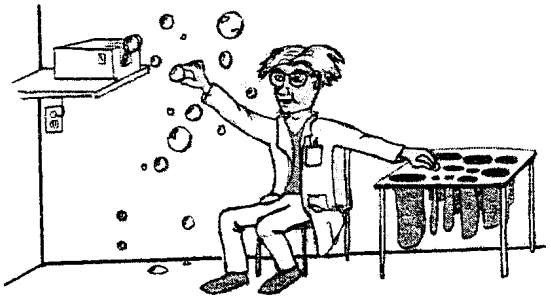
Выбор рассматриваемого примера мотивируется его простотой, а также тем, что в этом случае поддерживается хорошая платформа, которая позволяет увидеть механику метода `Template Method` в действии. Хотя на самом деле разрабатывать приложения, подобные `ftoc`, не рекомендуется.

Рассматриваемый пример демонстрирует пагубные последствия, к которым может привести злоупотребление шаблонами. Использование в данном случае шаблона `Template Method` является нецелесообразным, поскольку это ведет к усложнению программы и к увеличению объема кода. Инкапсуляция цикла main при разработке любого приложения на первый взгляд кажется неплохой идеей, хотя на практике эта идея может оказаться "бесплодной".

Проектные шаблоны обеспечивают разработчикам целый ряд возможностей, а также обеспечивают разрешение многих проблем, возникающих в процессе проектирования. Но отсюда вовсе не следует, что эти шаблоны должны применяться повсеместно. В данном случае, несмотря на то, что шаблон `Template Method` может применяться в целях устранения тех или иных проблем, его использование все же не рекомендуется. Накладные расходы, связанные с применением шаблона, превышают возможную прибыль.

Ну а теперь обратите внимание на более сложный пример. (листинг 14.4.)

Сортировка методом "пузырька"[3]



Листинг 14.4. BubbleSorter.java

```
public class BubbleSorter
{
  static int operations = 0;
  public static int sort(int[] array)
  {
  operations = 0;
  if (array.length <= 1)
    return operations;

  for (int nextToLast = array.length-2;
       nextToLast >= 0; nextToLast--)
    for (int index = 0; index <= nextToLast; index++)
      compareAndSwap(array, index);

  return operations;

  private static void swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void compareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1]) swap(array, index);
      operations++;
  }
}
```

[3]Как и в случае с Application, приложение Bubble Sort можно считать идеальным для учебных целей. Не следует применять его для сортировки больших объемов данных, поскольку в этом случае применяются более совершенные алгоритмы.

Класс BubbleSorter "знает" о том, каким образом следует сортировать целочисленный массив с помощью алгоритма сортировки "пузырьком". Метод сортировки BubbleSorter использует алгоритм, включающий метод сортировки "пузырьком". Два вспомогательных метода, swap и compareAndSwap, реализуют детали, связанные с обработкой целых чисел и массивов, а также механику, требуемую алгоритмом сортировки.

С помощью шаблона Template Method можно выделять алгоритм сортировки "пузырьком" в виде абстрактного базового класса, именуемого BubbleSorter. Этот класс включает реализацию функции сортировки, которая вызывает абстрактный метод outOfOrder и другой метод — swap. Метод outOfOrder выполняет сравнивание двух смежных элементов массива. При этом возвращается значение true, если элементы неупорядочены. Метод swap выполняет перестановку двух смежных элементов массива.

Метод sort ничего не "знает" ни о самом массиве, ни о том, как следует обрабатывать объекты, находящиеся в этом массиве. Он просто вызывает функцию outOfOrder с различными индексами массива, а также определяет, была ли выполнена перестановка индексов (листинг 14.5).

Листинг 14.5. BubbleSorter.java

```
public abstract class BubbleSorter
{
  private int operations = 0;
  protected int length = 0;

  protected int doSort()
  {
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (outOfOrder(index))
          swap(index);
        operations++;
      }

    return operations;
  }

  protected abstract void swap(int index);
  protected abstract boolean outOfOrder(int index)
}
```

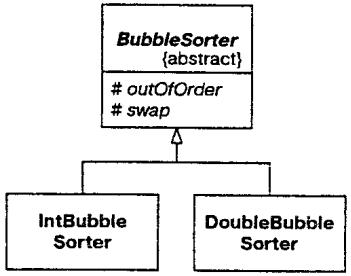


Рис. 14.1. Структура класса Bubble-Sorter

Располагая `BubbleSorter`, можно создавать простые производные методы, которые могут применяться для сортировки любых объектов. Например, можно сформировать класс `intBubbleSorter`, обеспечивающий сортировку целочисленных массивов, а также класс `DoubleBubbleSorter`, применяемый для сортировки массивов, содержащих вещественные числа удвоенной точности. (Рис. 14.1, листинг 14.6 и листинг 14.7.)

Листинг 14.6. `IntBubbleSorter.java`

```
public class IntBubbleSorter extends BubbleSorter
{
  private int[] array = null;
  public int sort(int[] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```


Листинг 14.7. `DoubleBubbleSorter.java`

```
public class DoubleBubbleSorter extends BubbleSorter
{
  private double[] array = null;
  public int sort(double[] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    double temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

Шаблон `Template Method` демонстрирует одну из классических форм метода повторного использования, применяемого в объектно-ориентированном программировании. Обобщенные алгоритмы размещаются в базовом классе, а затем наследуются в различных детализированных контекстах. Но использование этой техники предполагает определенные затраты. Это связано с тем, что между производными и базовыми классами устанавливается очень сложная взаимосвязь.

Например, функции `outOfOrder` и `swap` из класса `intBubbleSorter` представляют необходимый "минимальный набор", применяемый другими видами алгоритмов сортировки. И все же не существует способа повторного применения `outOfOrder` и `swap` другими алгоритмами сортировки. Реализовав наследование с помощью `BubbleSorter`, мы "обречем" `IntBubbleSorter` на вечное ограничение рамками `BubbleSorter`. В этом случае дополнительные возможности предлагаются шаблоном `Strategy`.

Шаблон Strategy

Благодаря использованию шаблона `Strategy` разрешается проблема инвертирования зависимостей между обобщенным алгоритмом и его детализированной реализацией. Причем в этом случае могут применяться самые различные методы.

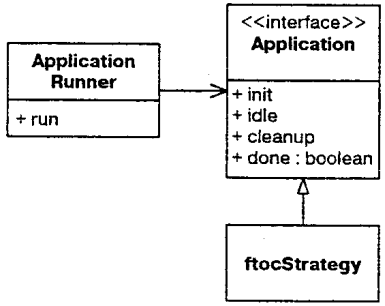


Рис. 14.2. Шаблон Strategy в составе алгоритма Application

Снова вернемся к проблеме, связанной с неоправданным применением шаблонов в классе `Application`.

Вместо того чтобы обобщенный алгоритм приложения включать в абстрактный базовый класс, в данном случае он будет размещен в *конкретном* классе, именуемом `ApplicationRunner`. В интерфейсе `Application` будут определены абстрактные методы, которые должен вызывать обобщенный алгоритм. На базе этого интерфейса будет наследоваться `ftocStrategy`, а затем передаваться классу `ApplicationRunner`. Затем именно `ApplicationRunner` будет делегирован для данного интерфейса. (Рис. 14.2, а также листинги с 14.8 по 14.10.)

Листинг 14.8. `ApplicationRunner.java`

```
public class ApplicationRunner
{
  private Application itsApplication = null;

  public ApplicationRunner(Application app)
  {
    itsApplication = app;
  }
  public void run()
  {
    itsApplication.init();
    while (!itsApplication.done())
    itsApplication.idle();
    itsApplication.cleanup();
  }
}
```

Листинг 14.9. `Application.java`

```
public interface Application
{
```



```
  public void init();
  public void idle();
  public void cleanup();
  public boolean done();
}
```

Листинг 14.10. `ftocStrategy.java`

```
import java.io.*;
public class ftocStrategy implements Application
{
  private InputStreamReader isr;
  private BufferedReader br;
  private boolean isDone = false;

  public static void main(String[] args) throws Exception
  {
    (new ApplicationRunner(new ftocStrategy())).run();
  }

  public void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  public void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() == 0)
      isDone = true;
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.printin("F=" + fahr + ", C=" + celcius);
    }
  }

  public void cleanup()
  {
    System.out.printin("ftoc exit");
  }

  public boolean done()
  {
    return isDone;
  }

  private String readLineAndReturnNullIfError()
```

```
    {
      String s;
      try
      {
        s = br.readLine();
      }
      catch(lOException e)
      {
        s = null;
      }
      return s;
    }
}
```

Очевидно, что применяемая в данном случае структура обеспечивает преимущества, а также снижает уровень накладных расходов (относительно структуры `Template Method`). Шаблон `Strategy` вовлекает более полные классы, а также носит косвенный характер (по сравнению с `Template Method`) Предоставление указателя при выполнении `ApplicationRunner` связано с несколько большими накладными расходами (если за шкалу отсчета брать время выполнения и объем, занимаемый хранимыми данными), чем в случае наследования. С другой стороны, если требуется запустить на выполнение большое количество приложений, можно использовать повторно *экземпляр* `ApplicationRunner`, осуществляя передачу параметров в нескольких экземплярах `Application`. В результате уменьшается степень связывания между обобщенным алгоритмом и контролируемыми им деталями.

Ни один из видов упомянутых преимуществ и накладных расходов не перекрывает "соседа". Как правило, наибольшее беспокойство приносит дополнительный класс, требуемый шаблону `Strategy`. Ну а теперь перейдем к рассмотрению других вопросов.

И снова сортировка

Обратимся к рассмотрению сортировки методом "пузырька", реализуемой с помощью шаблона `Strategy`. (Листинги с 14.11 по 14.13.)

Листинг 14.11. `BubbleSorter.java`

```
public class BubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public BubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
          itsSortHandle.swap(index);
        operations++;
      }

    return operations;
  }
}
```


Листинг 14.12. `SortHandle.java`

```
public interface SortHandle
{
  public void swap(int index);
  public boolean outOfOrder(int index);
  public int length();
  public void setArray(Object array);
}
```

Листинг 14.13. Листинг 14.13. `IntSortHandle.java`

```
public class IntSortHandle implements SortHandle
{
  private int[] array = null;

  public void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  public void setArray(Object array)
  {
    this.array = (int[])array;
  }
```

```
  public int length()
  {
    return array.length;
  }

  public boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

Обратите внимание на то, что класс `IntSortHandle` "ничего не знает" о классе `BubbleSorter`. Не существует зависимости независимо от реализации метода сортировки "пузырьком". В этом случае шаблон `Template Method` неприемлем. Снова обратите внимание на листинг 14.6, после чего вы наверняка заметите, что класс `intBubbleSorter` непосредственно зависит от класса `BubbleSorter`, включающего алгоритм сортировки методом "пузырька".

На самом деле подход с применением шаблона `Template Method` приводит к частичному нарушению принципа DIP. Это связано с тем, что реализованы методы `swap` и `outOfOrder`, которые непосредственно зависят от сортировки методом "пузырька". Подобная зависимость не характерна в случае применения подхода с шаблоном `Strategy`. Благодаря этому можно применять класс `IntSortHandle` вместе с реализациями `Sorter` вместо того, что применять `BubbleSorter`.

Например, можно разработать вариант алгоритма сортировки методом "пузырька", который завершает свою работу досрочно в случае, если массив будет упорядочен (листинг 14.14). Алгоритм `QuickBubbleSorter` также может использовать класс `IntSortHandle` или любой другой класс, производный от класса `SortHandle`.

Листинг 14.14. `QuickBubbleSorter.java`

```
public class QuickBubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public QuickBubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
```



```
    operations = 0;
    if (length <= 1)
      return operations;

    boolean thisPassInOrder = false;
    for (int nextToLast = length-2; nextToLast >= 0 &&
         !thisPassInOrder; nextToLast--)
    {
      thisPassInOrder = true; // потенциально.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
        {
          itsSortHandle.swap(index);
          thisPassInOrder = false;
        }
        operations++;
      }
    }
    return operations;
  }
}
```

Как видите, шаблон `Strategy` обеспечивает одно дополнительное преимущество по сравнению с шаблоном `Template Method`. В то врем, как шаблон `Template Method` позволяет сформировать обобщенный алгоритм, который позволяет манипулировать несколькими возможными детализированными реализациями, шаблон `Strategy` полностью совместим с принципом DIP. Благодаря этому каждая детализированная реализация может обрабатываться каждым из различных обобщенных алгоритмов.

Резюме

Оба шаблона, `Template Method` и `Strategy`, позволяют отделять алгоритмы высокого уровня от низкоуровневых подробностей. Также обеспечивается повторное использование алгоритмов высокого уровня, причем исключается зависимость от деталей. Шаблон `Strategy` также позволяет организовать использование деталей независимо от алгоритма высокого уровня. Это достигается за счет незначительного усложнения, затрат несколько большего количества памяти, а также времени выполнения.

Литература

1. Gamma и др. *Design Patterns.* Reading, MA: Addison-Wesley. 1995.
2. Martin. Robert C. и др. *Pattern Languages of Program Design 3.* Reading, MA: Addison-Wesley. 1998.

15

Шаблоны Facade и Mediator

А сейчас мы приступим к рассмотрению шаблонов общего назначения. Они применяются для установки некоторого рода политики по отношению к другой группе объектов. Шаблон `Facade` устанавливает политику в направлении "сверху вниз", а шаблон `Mediator` — в направлении "снизу вверх". Эффект применения `Facade` очевиден, хотя весьма ограничен. В случае с шаблоном `Mediator` эффект будет невидимым, а политика его применения носит рекомендательный характер.

Шаблон Facade

Шаблон `Facade` применяется в том случае, если требуется установить простой, но в то же время специфичный интерфейс для группы объектов, для которых выбран сложный родовой интерфейс. Например, обратите внимание на класс

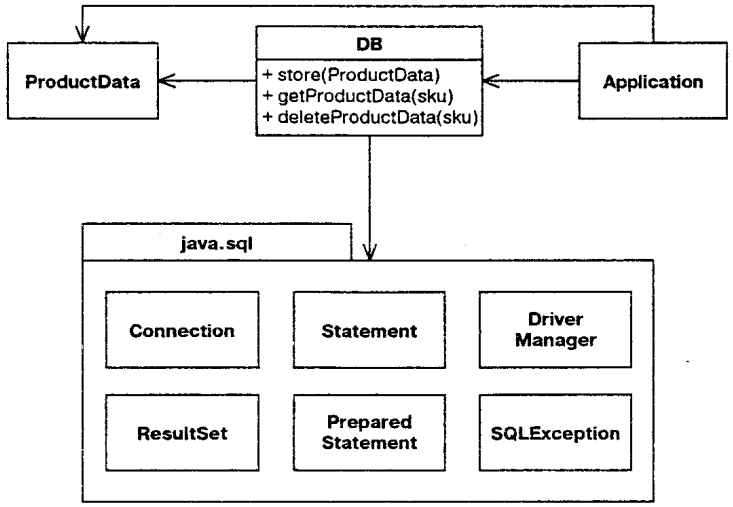


Рис. 15.1. Шаблон DB Facade

`DB.java`, код которого включен в листинг 26.9 (глава 26). Этот класс применяет очень простой интерфейс, который является специфичным для `productData` относительно сложных родовых интерфейсов классов в пакете `java.sql`. На рис. 15.1 приводится соответствующая структура.

Обратите внимание, что класс DB предотвращает потребность `Application` в том, чтобы знать детали реализации пакета `Java.sql`. Он скрывает сложность и обобщенный характер `java.sql`, маскируя их очень простым, но в то же время специфичным интерфейсом.

Шаблон `Facade`, как и DB, устанавливает применение политики в пакете `java.sql`. Он "знает", каким образом следует инициализировать и закрывать подключение к базе данных. Также известно, каким образом следует транслировать элементы `ProductData` в поля базы данных (и наоборот). Также известно, каким образом следует формировать подходящие запросы и команды, позволяющие выполнять обработку информации в базе данных. В этом случае излишняя сложность скрывается от пользователей. С "точки зрения" `Application`, `java.sql` не существует; поскольку он сокрыт за `Facade`.

В процессе использования шаблона `Facade` разработчикам следует адаптировать соглашение таким образом, чтобы все вызовы базы данных направлялись через DB. Если же какая-то часть кода `Application` направляется непосредственно `java.sql` вместо того, чтобы использовать `Facade`, в этом случае соглашение будет нарушено. Следовательно, `Facade` устанавливает политики для приложения. Согласно соглашению, DB выступает в качестве единственного брокера возможностей `java.sql`.

Шаблон Mediator

Шаблон Mediator также применяется для установки политики. В то время как Facade устанавливает политику, используя видимый и ограничивающий метод, Mediator устанавливает свои политики скрытым и не ограничивающим способом. Например, класс QuickEntryMediator, код которого приводится в листинге 15.1, находится "за кулисами", а также связан с полями ввода текста в списке. Если вы вводите данные в поле ввода текста, выделяется первый элемент в списке, который соответствует вводимому тексту. Благодаря этому облегчается ввод аббревиатур, а также обеспечивается быстрый вывод элементов из списка.

Листинг 15.1. QuickEntryMediator.java

```
package utility;

import javax.swing.*;
import javax.swing.event.*;

/**
QuickEntryMediator. Этот класс использует JTextField и
JList. Пользователь вводит символы в JTextField,
предваряемые записями в JList. Автоматически
выбирается первый элемент в Jlist, который
соответствует текущему префиксу JTextField.

Если JTextField - нуль или его префикс не совпадает
с элементами в JList, выбор JList устраняется.

Отсутствуют методы, применяемые для вызова этого
объекта. Можно просто создать объект и забыть о
нем. (Но не удаляйте его вместе с ''мусором''...)

Пример:

JTextField t = new JTextField();
JList 1 = new JList();

QuickEntryMediator qem = new QuickEntryMediator(t, 1);
// а вот и сама программа.

савтор Роберт К. Мартин, Роберт С. Косс
@дата 30 июня, 1999 г, 2113 (SLAC)
*/

public class QuickEntryMediator {
  public QuickEntryMediator(JTextField t, Jlist 1) {
    itsTextField = t;
    itsList = 1;
```



```
    itsTextField.getDocument().addDocumentListener(
      new DocumentListener() {
        public void changedUpdate(DocumentEvent e){
          textFieldChanged();
        }

        public void insertUpdate(DocumentEvent e) {
          textFieldChanged();
        }

        public void removeUpdate(DocumentEvent e) {
          textFieldChanged();
        }
      } // новый DocumentListener );
    }// addDocumentListener
  }//QuickEntryMediator()

  private void textFieldChanged() {
    String prefix = itsTextField.getText();

    if (prefix.length() == 0) {
      itsList.clearSelection();
      return;
      }

      ListModel m = itsList.getModel();
      boolean found = false;
      for (int i = 0; found == false && i < m.getSize(); i++) {
        Object o = m.getElementAt(i);
        String s = o.toString();
        if (s.startsWith(prefix)){
          itsList.setSelectedValue(o, true);
          found = true;
        }
      }

    if (!found) {
      itsList.clearSelection();
    }
  } // textFieldChanged

  private JTextField itsTextField;
  private JList itsList;
} // класс QuickEntryMediator
```

Структура QuickEntryMediator продемонстрирована на рис. 15.2. Экземпляр QuickEntryMediator конструируется на основе JList и JTextField. Объект QuickEntryMediator регистрирует анонимный DocumentListener вместе с JTextField. Этот слушатель вызывает метод textFieldChanged

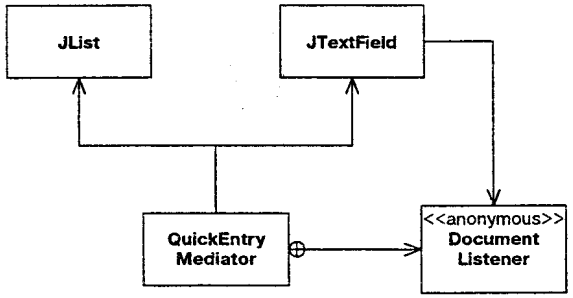


Рис. 15.2. Объект QuickEntryMediator

при наличии каких-либо изменений в тексте. При вызове этого метода производится поиск элементов в `JList`, которые предваряются текстом, с последующим их выделением.

Пользователи, работающие с `JList` и `JTextField`, не подозревают о существовании `Mediator`. Этот шаблон "ведет себя тихо", формируя политику по отношению к этим объектам. При этом не остаются какие-либо видимые "следы".

Резюме

Если политика должна быть яркой и заметной, следует воспользоваться услугами шаблона `Facade`. Если же во главу угла ставится скрытность и аккуратность, более подходящим является шаблон `Mediator`. Как правило, фасады (Facade) являются наиболее заметными. Любой из нас согласится работать с фасадами вместо объектов, скрываемых под ним. Медиатор (Mediator) скрыт от пользователей. Предлагаемая политика обычно является негласной и не регулируется какими-либо соглашениями.

Литература

1. Gamma и др. *Design Patterns*. Reading. MA: Addison-Wesley, 1995.

16

Шаблоны Singleton и Monostate

Бесконечно блаженство существования! Именно оно и ничего кроме него.

Эдвин А. Эбботт

Между классами и их экземплярами обычно устанавливаются взаимосвязи "один ко многим". Для большинства классов можно создавать несколько экземпляров. Экземпляры создаются по мере необходимости, а их применение реализуется в том случае, если это приносит какую-либо пользу. Они появляются в потоке действий, реализующих распределение и очистку памяти.

Некоторые классы обладают лишь одним экземпляром. Этот экземпляр активизируется в том случае, если программа запускается на выполнение. После завершения выполняемой программы экземпляр класса разрушается. Подобные

объекты иногда выступают в качестве своего рода "корней" приложения. Начиная с "корней", можно отслеживать многие другие объекты в системе. Иногда в качестве этих объектов применяются так называемые фабрики, с помощью которых можно создавать другие объекты в системе. Порой эти объекты являются диспетчерами, несущими ответственность за отслеживание некоторых других объектов, а также реализующими управление ими.

Какими бы ни были рассматриваемые объекты, возможны серьезные логические ошибки в случае, если создается более одного объекта. Если было создано более одного "корня", доступ к объектам приложения будет зависеть от выбранного "корня". Программисты, не подозревающие о существовании более чем одного "корня", могут столкнуться с ситуацией, когда приходится иметь дело с поднабором объектов приложения, о которых ничего не известно. Если существует несколько фабрик, "опека" созданных объектов может быть скомпрометирована. Если существует несколько диспетчеров, последовательные действия могут быть преобразованы в параллельную форму.

Возможно, все дело в том, что механизмы, придающие особенность этим объектам, являются избыточными. Ведь в процессе инициализации приложения создается один из таких механизмов, который используется в дальнейшей работе. На самом деле подобный алгоритм действий является наилучшим. Следует избегать применения механизма, если в этом нет насущной потребности. Также ожидается, что наш код будет взаимодействовать с содержимым. Если механизм, порождающий особенности, устроен просто, преимущества, обеспечиваемые коммуникациями, превышают затраты, связанные с эксплуатацией механизма.

Материал этой главы описывает два шаблона, призванных придавать некие особенности объектам. Этим шаблонам присущи различные соотношения "выгода-цена". Во многих случаях затраты достаточно малы, чтобы негативно влиять на положительный эффект выразительности, связанный с их применением.

Шаблон Singleton[1]

Шаблон `Singleton` устроен очень просто. Его функционирование демонстрируется с помощью тестового случая, код которого приводится в листинге 16.1. Первая тестовая функция демонстрирует технологию доступа к экземпляру `Singleton` с помощью общедоступного статического метода `Instance`. Здесь также показано, что, если метод `Instance` вызывается несколько раз, возвращается ссылка на один и тот же интерфейс. Во втором тестовом случае показано, что если класс `Singleton` не включает общедоступные конструкторы, невозможно создать экземпляр, не воспользовавшись методом `Instance`.

[1][GOF95], с. 127.

Листинг 16.1. Тестовый случай для шаблона Singleton

```
import junit.framework.*;
import Java.lang.reflect.Constructor;

public class TestSimpleSingleton extends TestCase
{
  public TestSimpleSingleton(String name)
  {
    super(name);
  }

  public void testCreateSingleton()
  {
    Singleton s = Singleton.Instance();
    Singleton s2 = Singleton.Instance();
    assertSame(s, s2);
  }

  public void testNoPublicConstructors() throws Exception
  {
    Class singleton = Class.forName("Singleton");
    Constructor[] constructors = singleton.getConstructors();
    assertEquals("public constructors.", 0, constructors.length);
  }
}
```

Этот тестовый случай представляет собой спецификацию шаблона `Singleton`. Логическим следствием этого тестового случая является код, приведенный в листинге 16.2. Этот код очень прост для понимания, поскольку он является ничем иным, как экземпляром класса `Singleton`, который находится в области действия статической переменной `Singleton.theInstance`.

Листинг 16.2. Реализация класса Singleton

```
public class Singleton
{
  private static Singleton theInstance = null;
  private Singleton() {}

  public static Singleton Instance()
  {
    if (theInstance == null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

Преимущества шаблона Singleton

- **Перекрестные платформы**. Подходящее промежуточное ПО (например, RMI) позволяет расширять возможности класса `Singleton` таким образом, что обеспечивается его использование на многих JVM, а также на различных типах компьютеров.
- **Применимость к любому классу**. Можно преобразовать любой класс таким образом, что будет создан класс `Singleton`. Для этого достаточно преобразовать конструкторы класса так, чтобы они стали частными, а также добавить соответствующие статические функции и переменные.
- **Возможность создания производного класса**. При наличии какого-либо класса можно создать его подкласс, который и будет соответствовать шаблону `Singleton`.
- **"Ленивое" развитие**. Если класс `Singleton` никогда не использовался, он и не будет создан.

Недостатки шаблона Singleton

- **Неопределенное разрушение**. Не существует однозначного способа, позволяющего разрушить или зарезервировать класс `Singleton`. Если был добавлен метод `decommission`, обнуляющий `outtheInstance`, другие модули системы могут сохранять установленную ссылку на экземпляр `Singleton`. Последовательные вызовы `Instance` приводят к созданию других экземпляров, причем в этом случае могут одновременно существовать два экземпляра. Эта проблема особенно остро стоит в C++, где экземпляр объекта *может уничтожаться,* приводя к возможному "разыменованию" устраняемого объекта.
- **Отсутствие наследования**. Класс, наследуемый от `Singleton`, не будет обладать свойствами исходного класса. Если требуется, чтобы он имел тип `Singleton`, необходимо добавить статическую функцию и переменную.
- **Эффективность**. Каждый вызов `Instance` сопровождается вызовом конструкции `if`. При большинстве вызовов конструкция `if` бесполезна.
- **Непрозрачность**. Пользователи `Singleton` знают о том, что работают с этим классом, поскольку им приходится вызывать метод `Instance`.

Пример использования шаблона Singleton

Предположим, что мы имеем дело с Web-системой, которая открывает доступ пользователям к защищенным областям Web-сервера. Подобные системы снабжены базами данных, включающими имена пользователей, пароли, а также описания других пользовательских атрибутов. Более того, предположим, что доступ к базе данных осуществляется с помощью API от независимых производителей. Можно получить доступ к базе данных непосредственно из каждого модуля, который должен быть написан (и прочтен) пользователем. В результате API от независимых производителей будут "рассеяны" по всему коду, в результате чего не останется места для соглашений о структуре или методе доступа.

Один из лучших способов, используемых в данном случае, представляет шаблон `Facade`, а также класс `UserDatabase`, поддерживающий методы, которые применятся для чтения и записи объектов `User`. Эти методы обеспечивают доступ API от независимых производителей к базе данных, выполняя операцию трансляции между объектами `User` и таблицами (строками) базы данных. Находясь в `UserDatabase`, можно устанавливать соглашения о структуре и доступе. Например, можно сделать так, что ни одна из записей `User` не будет фиксироваться до тех пор, пока не указано имя пользователя. Либо можно реализовать последовательную форму доступа к записи `User`, в результате чего исключается возможность одновременного чтения и записи для двух модулей.

Код, приведенный в листингах 16.3 и 16.4, демонстрирует шаблон `Singleton` в действии. Класс `Singleton` в данном случае называется `UserDatabaseSource`. Он реализует интерфейс `UserDatabase`. Обратите внимание, что статический метод `instance()` не включает традиционную конструкцию `if`, предохраняющую от создания нескольких экземпляров. Вместо этого используются возможности инициализации, предоставляемые Java.

Листинг 16.3. Интерфейс `UserDatabase`

```
public interface UserDatabase
{
  User readUser(String userName);
  void writeUser(User user);
}
```

Листинг 16.4. `UserDatabaseSource Singleton`

```
public class UserDatabaseSource implements UserDatabase
{
  private static UserDatabase theInstance =
    new UserDatabaseSource();

  public static UserDatabase instance()
  {
    return theInstance;
  }

  private UserDatabaseSource()
  {
  }
```

```
  public User readUser(String userName)
  {
    // реализация
    return null; // осталось скомпилировать
  }

  public void writeUser(User user)
  {
    // реализация
  }
}
```

Итак, вы ознакомились с примером максимально обобщенного использования шаблона `Singleton`. Применение этого примера на практике гарантирует, что доступ к базе данных осуществляется через единственный экземпляр `UserDatabaseSource`. В результате облегчается установка проверок, счетчиков и "замков" в `UserDatabaseSource`, который формирует упомянутые ранее соглашения по доступу и структуре.

Шаблон Monostate[2]

Другой метод установки особенностей объектов обеспечивает шаблон `Monostate`. Но он использует совершенно другой рабочий механизм. Наглядная демонстрация работы этого механизма производится в тестовом случае `Monostate`, код которого приводится в листинге 16.5.

Первая тестовая функция просто описывает объект, для переменной `x` которого может устанавливаться и считываться значение. Во втором тестовом случае показаны два экземпляра одного и того же класса, которые ведут себя *как единое целое*. Если переменной `x`, соответствующей одному экземпляру, было присвоено то или иное значение, можно считывать это значение, получив доступ к переменной `x` из другого экземпляра. Получается, что два экземпляра являются носителями различных имен одного и того же объекта.

Листинг 16.5. Тестовый случай `Monostate`

```
import junit.framework. *;

public class TestMonostate extends TestCase
{
  public TestMonostate(String name)
  {
    super(name);
  }
```

[2][BALL2000].


```
  public void testInstance()
  {
    Monostate m = new Monostate();
    for (int x = 0; x<10; x++)
    {
      m.setX(x);
      assertEquals(x, m2.getX());
    }
  }

  public void testInstancesBehaveAsOne()
  {
    Monostate m1 = new Monostate();
    Monostate m2 = new Monostate();

    for (int x = 0; x<10; x++)
    {
      m1.setX(x);
      assertEquals(x, m2.getX());
    }
  }
}
```

Если вы включили класс `Singleton` в этот тестовый случай и заменили конструкциями `new Monostate` вызовы `Singleton.Instance`, тестовый случай по-прежнему будет выполняться. Это связано с тем, что данный тестовый случай описывает *поведение* `Singleton`, не учитывая ограничения единственного экземпляра!

Каким же образом два экземпляра могут вести себя так, как будто речь идет об одном объекте? В данном случае это означает то, что два объекта должны совместно использовать одни и те же переменные. Это легко достигается в том случае, если все переменные будут статическими. Листинг 16.6 демонстрирует реализацию `Monostate`, который проходит указанный выше тестовый случай. Обратите внимание, что переменная `itsX` является статической. Также обратите внимание на то, что *ни один из методов не является статическим*. Это обстоятельство играет важную роль в дальнейшем.

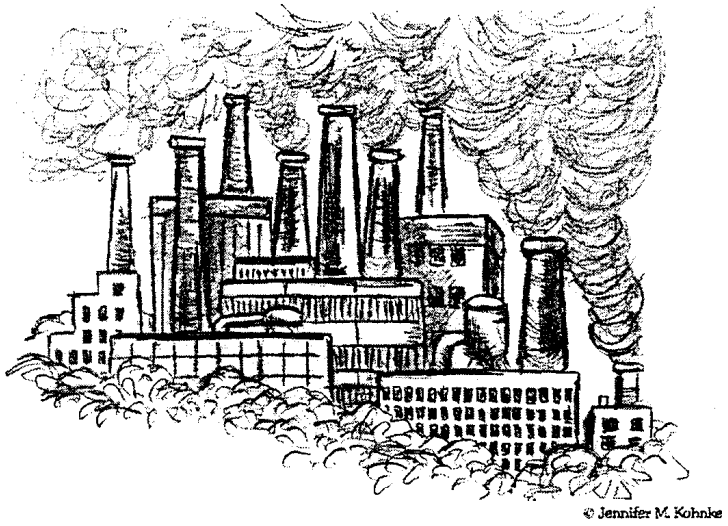
Листинг 16.6. Реализация шаблона `Monostate`

```
public class Monostate
{
  private static int itsX = 0;
  public Monostate() {}

  public void setX(int x)
  {
    itsX = x;
  }
```

21

Шаблон Factory



Человек, строящий фабрику, сооружает храм.

Кэлвин Кулидж

Принцип инверсии зависимостей (DIP[1], Dependency-Inversion Principle) гласит, что следует отдавать предпочтение зависимостям от абстрактных классов, избегая зависимости от конкретных классов, особенно, если такие классы не относятся к семейству неизменных. Следующий фрагмент кода вступает в явное противоречие с этим принципом.

```
Circle c = new Circle(origin, 1);
```

[1]Смотрите описания принципа инверсии зависимостей в гл. 11.

В данном случае `Circle` является конкретным классом. Следовательно, принцип DIP будут нарушать те модули, которые создают экземпляры класса `Circle`. Действительно, любая строка кода, в которой используется ключевое слово `new`, противоречит принципу DIP.

На практике имеет место ряд ситуаций, когда нарушение принципа DIP относительно безопасно[2]. Чем чаще изменяется конкретный класс, тем больше вероятность того, что зависимость от такого класса приведет к появлению определенных проблем.

Например, создание экземпляров класса `String` вряд ли приведет к появлению каких-либо проблем. Это связано с тем, что вероятность изменения этого класса в будущем практически равна нулю.

С другой стороны, в процессе активной разработки приложений мы сталкиваемся с большим количеством классов, которым присущ сверхвысокий уровень изменчивости. В случае появления зависимости от таких классов возникают определенные трудности. В целях предотвращения возможных неприятностей рекомендуется воспользоваться абстрактным интерфейсом.

Шаблон `Factory` позволяет создавать экземпляры конкретных объектов, причем в этом случае сохраняется зависимость от абстрактных интерфейсов. Следовательно, данный шаблон может в значительной мере пригодиться в процессе активной разработки приложений, при которой конкретные классы обладают высоким уровнем изменчивости.

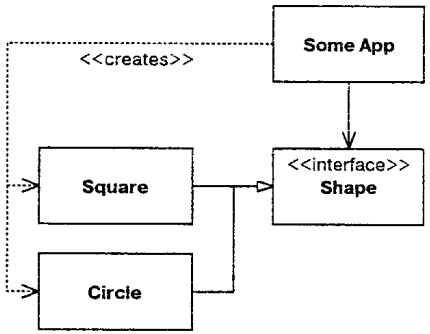


Рис. 21.1. Класс App, нарушающий принцип DIP в процессе создания конкретных классов

На рис. 21.1. представлен некоторый "проблематичный" сценарий. Здесь вы видите класс `SomeApp`, который зависит от интерфейса `Shape`. Этот класс использует экземпляры `Shape` исключительно с помощью интерфейса `Shape`. В данном случае не применяются методы, присущие классам `Square` или `Circle`.

[2]Соответствующие примеры можно найти в данной книге.

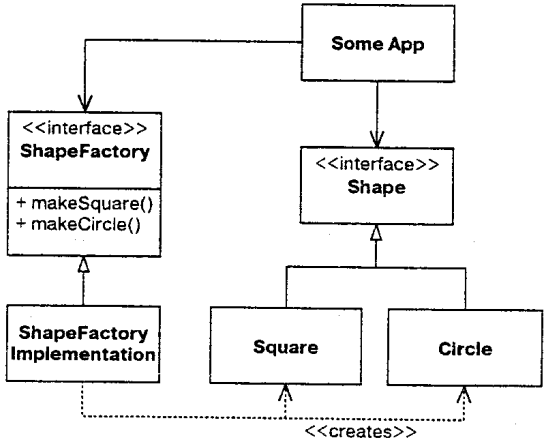


Рис. 21.2. Интерфейс ShapeFactory

Класс SomeApp также создает экземпляры классов `Square` и `Circle` и, следовательно, зависит от конкретных классов.

Описанную проблему можно решить, применив по отношению к классу SomeApp шаблон `Factory` так, как это изображено на рис. 21.2. Здесь демонстрируется интерфейс `ShapeFactory`, включающий два метода, `makeSquare` и `makeCircle`. Метод `makeSquare` возвращает экземпляр класса `Square`, а метод `makeCircle` возвращает экземпляр класса `Circle`. Обе функции возвращают тип данных `Shape`.

В листинге 21.1 приводится код интерфейса `ShapeFactory`, а в листинге 21.2 — код реализации `ShapeFactory`.

Листинг 21.1. `ShapeFactory.java`

```
public interface ShapeFactory
{
  public Shape makeCircle();
  public Shape makeSquare();
}
```

Листинг 21.2. `ShapeFactoryImplementation.Java`

```
public class ShapeFactoryImplementation implements ShapeFactory
{
  public Shape makeCircle()
  {
    return new Circle();
  }

  public Shape makeSquare()
  {
    return new Square();
  }
}
```


Обратите внимание, что использованный в данном случае прием полностью устраняет проблему зависимости от конкретных классов. Код приложения уже больше не зависит от класса `Circle` или `Square`, обеспечивая в то же время создание экземпляров этих классов. При этом манипулирование экземплярами осуществляется посредством интерфейса `Shape`, и в этом случае не используются специфичные для классов `Square` и `Circle` методы.

Итак, проблема, связанная с зависимостью от конкретного класса, устранена. По-прежнему следует создавать класс `ShapeFactory-Implementation`, но зато не нужно формировать классы `Square` и `Circle`. Вероятней всего, создание `ShapeFactoryImplementation` будет осуществляться с помощью модуля `main` или функции инициализации, связанной с этим модулем.

Цикл зависимости

Некоторые читатели, наверное, уже заметили, что с использованием описанной формы шаблона `Factory` связаны некоторые проблемы. Для каждого из производных модулей `Shape` в классе `ShapeFactory` предусмотрен отдельный метод. В результате возникает цикл зависимости, который усложняет процесс добавления в `Shape` новых производных модулей. В случае добавления каждого нового производного модуля `Shape` в интерфейс `ShapeFactory` необходимо включать соответствующий метод. Как правило, это означает необходимость рекомпиляции и повторного развертывания пользовательского интерфейса `ShapeFactory`[3].

Можно избавиться от подобного цикла в ущерб безопасности использования типов. Вместо того чтобы передавать в распоряжение интерфейса `ShapeFactory` по одному методу для каждого производного модуля `Shape`, можно сопоставить ему всего лишь одну функцию `make`, для которой определен тип данных `String`. Пример применения подобной методики демонстрируется в листинге 21.3. В этом случае требуется, чтобы в `ShapeFactoryImplementation` использовалась цепь `if/else`, с помощью которой определяется производный модуль класса `Shape`, для которого и формируется экземпляр. Именно эта методика проиллюстрирована в листингах 21.4 и 21.5.

[3] При работе с языком Java этой рекомендации можно не следовать. Можно изменить интерфейс, не выполняя рекомпиляцию и повторное развертывание клиентов, хотя это чревато разного рода

Листинг 21.3. Фрагмент кода, создающий цикл

```
public void testCreateCircle() throws Exception
{
  Shape s = factory.make("Circle");
  assert(s instanceof Circle);
}
```

Листинг 21.4. ShapeFactory.java

```
public interface ShapeFactory
{
  public Shape make(String shapeName) throws Exception;
}
```

Листинг 21.5. ShapeFactoryImplementation.java

```
public class ShapeFactoryImplementation implements ShapeFactory
{
  public Shape make(String shapeName) throws Exception
  {
    if (shapeName.equals("Circle"))
      return new Circle();
    else if (shapeName.equals("Square"))
      return new Square();
    else
      throw new Exception("ShapeFactory cannot create " + shapeName);
  }
}
```

Подобное решение чревато появлением проблем (ошибка времени выполнения). Данная ситуация аналогична неправильному указанию имени формы (shape) в качестве аргумента вызывающей программы, в результате чего отображается сообщение об ошибке времени компиляции. Эта проблема достаточно неприятна. Однако, если воспользоваться достаточным количеством модульных тестов, а также выполнять разработку, основанную на тестировании, можно будет перехватить ошибки времени выполнения задолго до того, как они превратятся в серьезные проблемы.

Замещаемые фабрики

Одно из наибольших преимуществ фабрик заключается в возможности выполнения подстановки одного экземпляра фабрики вместо другого. Благодаря этому можно изменять семейства объектов в рамках одного приложения.

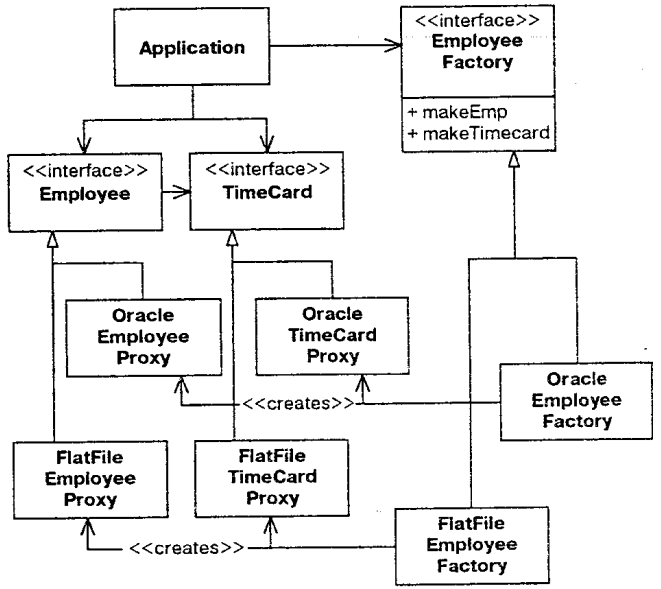


Рис. 21.3. Замещаемая фабрика

Например, представьте себе ситуацию, когда приходится адаптировать приложение с учетом множества реализаций баз данных. В нашем примере предполагается, что пользователи могут работать с простыми файлами таблиц либо воспользоваться конвертором Oracle™. В этом случае можно применить шаблон `Proxy`[4] в целях изоляции приложения от реализации базы данных. Также можно использовать фабрики для создания экземпляров прокси-объектов. Применяемая в этом случае структура представлена на рис. 21.3.

Обратите внимание, что существует две реализации `EmployeeFactory`. Одна из них создает прокси-объекты, которые работают с простыми табличными файлами, а вторая приводит к образованию прокси-объектов, предназначенных для работы с конвертором Oracle™. Также следует отметить, что самому приложению "не известно", какая именно из реализаций используется в данный момент времени. Также обратите внимание, что тип используемой реализации не играет решающей роли.

[4] Этот шаблон будет подробнее рассмотрен в гл. 26. Имейте в виду, что прокси представляет собой класс, который "знает" о правилах считывания конкретных объектов из определенных видов баз данных.

Применение фабрик для формирования схем тестов

В процессе разработки модульных тестов зачастую возникает потребность протестировать поведение определенного модуля отдельно от используемых им модулей. Например, предположим, что нам приходится работать с приложением `Payroll`, в котором используется база данных (рис. 21.4.) В данном случае следует протестировать функциональные свойства модуля `Payroll` отдельно от базы данных.

Рис. 21.4. Использование базы данных в модуле Payroll

Подобное тестирование можно выполнить путем использования абстрактного интерфейса для базы данных. Одна из реализаций этого интерфейса использует реальную базу данных. Другая реализация представляет собой тестовый код, задача которого заключается в имитации поведения базы данных и проверки правильности текущих вызовов базы данных. Подобная структура представлена на рис. 21.5. Модуль `PayrollTest` тестирует модуль `PayrollModule` путем его вызовов. Он также реализует интерфейс базы данных `Database`, обеспечивающий перехват вызовов базы данных, выполняемых модулем `Payroll`. В результате модуль `PayrollTest` может убедиться в "безупречном поведении" модуля `Payroll`. Это также позволяет модулю `PayrollTest` имитировать многие виды сбоев и проблем, возникающих в активной базе данных. Имитации подобного рода достаточно трудно добиться другими методами. Иногда подобный метод тестирования называют *спуфингом* (имитацией соединения).

В связи с вышеописанным возникает вопрос: каким образом модуль `Payroll` получает экземпляр модуля `PayrollTest`, используемый в качестве базы данных `Database`? Модуль `Payroll` не создает сам модуль `PayrollTest`. Также понятно, что модуль `Payroll` каким-то образом получает ссылку на реализацию базы данных `Database`, которую он будет использовать в будущем.

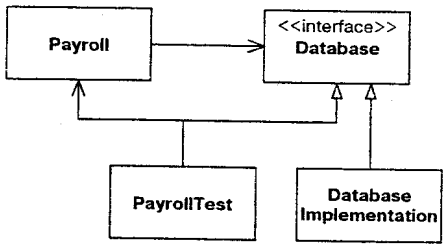


Рис. 21.5. Модуль PayrollTest имитирует соединение с базой данных

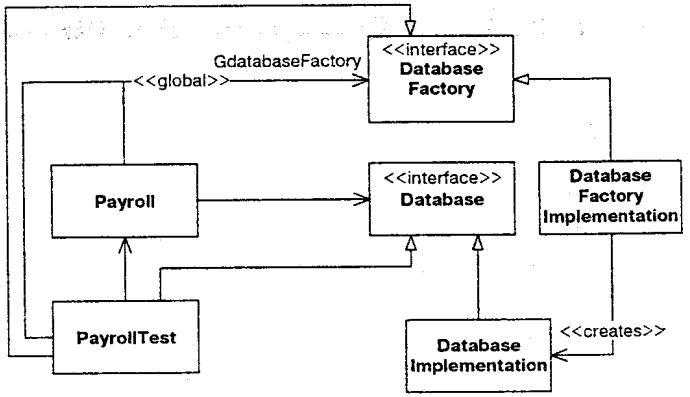


Рис. 21.6. Имитация соединения с Factory

В некоторых случаях `PayrollTest` естественным образом передает ссылку на базу данных `Database` модулю `Payroll`. В других случаях в модуле `PayrollTest` определяется глобальная переменная, которая будет устанавливать ссылку на базу данных `Database`. В ряде других случаев следует ожидать, что экземпляр `Database` будет создан модулем `Payroll`. В последнем случае, для того чтобы "вынудить" `Payroll` создать тестовую версию базы данных `Database`, можно воспользоваться шаблоном `Factory`, передав альтернативную фабрику модулю `Payroll`.

На рис. 21.6 показана возможная структура, реализующая применение данной методики на практике. Модуль `Payroll` получает фабрику посредством глобальной переменной (или статической переменной в глобальном классе) с именем `GdatabaseFactory`. Модуль `PayrollTest` реализует `DatabaseFactory`, а также устанавливает ссылку на самого себя в переменной `GdatabaseFactory`. Если `Payroll` использует фабрику для создания `Database`, модуль `PayrollTest` перехватывает вызов, а затем возвращает ссылку на самого себя. Таким образом, модуль `Payroll` удостоверяется в том, что был создан `PayrollDatabase`, но при этом `PayrollTest` вполне может сымитировать соединение с модулем `Payroll`, перехватывая при этом все вызовы базы данных.

Преимущества, связанные с использованием фабрик

Строгая интерпретация принципа DIP предполагает, что фабрики должны использоваться для каждого изменяемого класса системы. Более того, шаблон `Factory` обладает достаточно широкими возможностями. Эти два фактора в некоторых случаях могут побуждать разработчиков к использованию фабрик во всех возможных ситуациях. Конечно, это — крайность, поэтому автор книги категорически возражает против подобного решения.

Не следует начинать с использования фабрик в самом начале процесса разработки. Их следует включать в состав системы только тогда, когда возникает настоятельная потребность в этом. Например, если применяется шаблон `Proxy`, скорее всего, потребуется воспользоваться фабрикой в целях создания постоянных объектов. Или же в процессе модульного тестирования может возникнуть ситуация, когда потребуется сымитировать соединение с создателем объекта. Скорее всего, и в этом случае будут задействованы фабрики. Но не стоит начинать разработку системы, ориентируясь изначально на использование исключительно одних фабрик.

Фабрики представляют достаточно сложные объекты, использования которых следует по возможности избегать, особенно на ранних фазах осуществления проекта. Если фабрики применяются повсеместно, в этом случае возможности по расширению проекта сводятся к минимуму. Для формирования нового класса могут потребоваться, как минимум, четыре новых класса. Эти классы включают два класса интерфейсов, которые представляют новый класс и его фабрику, а также два конкретных класса, которые реализуют эти интерфейсы.

Резюме

Фабрики относятся к мощным инструментальным средствам. Они могут принести огромную пользу в плане соблюдения принципа DIP. Благодаря им стратегические модули высшего уровня могут создавать экземпляры классов, причем в этом случае отсутствует зависимость от конкретных реализаций этих классов. Фабрики также позволяют осуществлять переходы между абсолютно различными семействами реализаций в рамках одной группы классов. Тем не менее, работа с фабриками влечет свои сложности, которых лучше избегать. Использование этих объектов во всех ситуациях не всегда является оправданным.

Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

22

Практическое занятие: программа расчета зарплаты (часть 2)

Если что-либо представляется разумным и совершенным, остерегайтесь — вероятнее всего, это — самообман.

Дональд Э. Норман (The Design of Everyday Things, Donald A. Norman, Doubleday, 1990)

23

Шаблон Composite

Шаблон Composite является очень простым, но его применение влечет за собой серьезные последствия. Фундаментальная структура шаблона Composite показана на рис. 23.1. Здесь можно видеть иерархию, основанную на формах. Базовый класс Shape включает две производные формы Circle и Square. Третья производная форма является составной. Форма CompositeShape включает список нескольких экземпляров Shape. При вызове метода draw() для

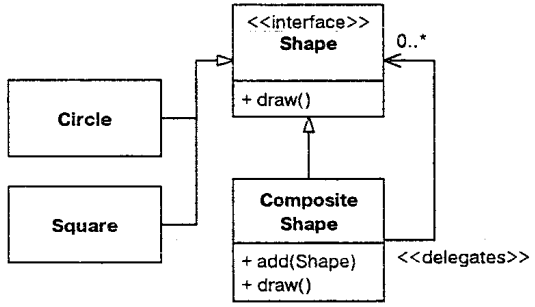


Рис. 23.1. Шаблон Composite

CompositeShape производится его делегирование всем экземплярам Shape, находящимся в списке.

Таким образом, экземпляр CompositeShape появляется в системе в качестве одинарной Shape. Этот экземпляр может быть передан любой функции или объекту, которые используют Shape, и будет вести себя как Shape. Однако в действительности это прокси-объект[1] для группы экземпляров Shape.

В листингах 23.1 и 23.2 показана одна возможная реализация CompositeShape.

Листинг 23.1. Shape.java

```
public interface Shape
{
  public void draw();
}
```

Листинг 23.2. CompositeShape.java

```
import Java.util.Vector;

public class CompositeShape implements Shape
{
  private Vector itsShapes = new Vector();
  public void add(Shape s)
  {
    itsShapes.add(s);
  }

  public void draw()
  {
    for (int i = 0; i < itsShapes.size(); i++)
    {
```

[1] Обратите внимание на схожесть со структурой в шаблоне Proxy.

```
      Shape shape = (Shape) itsShapes.elementAt(i);
      shape.draw();
    }
  }
}
```

Пример: составные команды

Обратимся к рассмотрению объектов Sensors и Command, о которых уже говорилось в главе 13. Там на рис. 13.3 приводился класс Sensor, использующий класс Command. Когда Sensor обнаруживает сигнал вызова, в Command вызывается метод do().

В приведенном ранее рассмотрении не было упомянуто, что существует много случаев, когда Sensor должен выполнять более чем одну Command. Например, когда бумага достигает определенного места в механизме протяжки, она должна отключить оптический датчик. Этот датчик затем останавливает один двигатель, запускает другой двигатель и включает захваты.

Сначала было принято во внимание, что каждый класс Sensor должен обслуживать список объектов Command (рис. 23.2). Однако вскоре было обнаружено, что всякий раз, когда Sensor нуждался в выполнении более чем одной Command, он всегда обращался с этими объектами Command идентично. То есть, Sensor только итерировал по списку и вызывал метод do() для каждой Command. И это было идеально в случае использования шаблона Composite.

Таким образом, класс Sensor был оставлен и создан CompositeCommand, как показано на рис. 23.3.

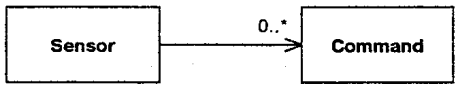


Рис. 23.2. Sensor включает несколько команд

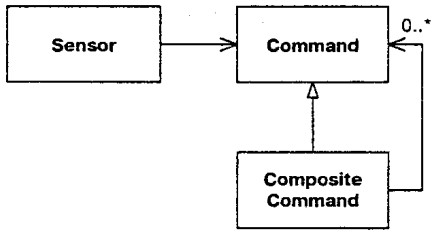


Рис. 23.3. Составная команда

Это значит, что не требуется изменять Sensor или Command. Можно добавить множественность Commands к Sensor без их изменения. Это приложение соответствует принципу OCP.

Множественное и единственное число

А теперь мы приходим к интересному выводу. Была возможность определять поведение Sensors так, как будто бы они содержали много Commands, причем без необходимости изменять Sensors. Существует множество других подобных ситуаций в обычном программном проекте, т.е. должны быть случаи, когда можно использовать шаблон Composite вместо того, чтобы формировать список или вектор объектов.

Это можно объяснить по-другому. Взаимосвязь между Sensor и Command является однозначной. Изменив эту взаимосвязь на "один ко многим", мы неожиданно получили способ имитировать поведение взаимосвязи "один ко многим" без фактического наличия этого типа связи. Однозначная взаимосвязь проще для понимания, программирования и поддержки, чем взаимосвязь "один ко многим". Т.е. мы получили идеальный проект. Таким образом шаблон Composite позволяет заменять в проекте взаимосвязи "один ко многим" однозначными взаимосвязями.

Конечно, не все взаимосвязи "один ко многим" могут быть заменены однозначными взаимосвязями с помощью шаблона Composite. Только те, в которых каждый объект в списке обрабатывается идентично, являются "кандидатами" на подобное изменение. Например, пусть поддерживается список работников, в котором следует определить тех, день выплаты зарплаты которых настал сегодня. В этом случае шаблон Composite непригоден, поскольку различные работники трактуются различным образом.

Тем не менее, существует довольно много взаимосвязей "один ко многим", которые могут быть преобразованы с помощью шаблона Composite. И получаемые в этом случае преимущества существенны. Вместо дублированной обработки списка и выполнения итерационного кода для каждого из клиентов, достаточно выполнить требуемый код один раз в составном классе.

24

Обратно к шаблонам: Observer

В процессе написания этой главы преследовались особые цели. Здесь рассматривается шаблон `Observer`[1], но описание этого шаблона не является в данном случае главной задачей. Данная глава посвящена проблеме применения шаблона в процессе разработки и в кодировании.

В предыдущих главах уже было описано достаточное количество шаблонов, но вопрос о предварительной обработке кодов почти не обсуждался. В этой ситуации может показаться, что готовые шаблоны просто включаются в код или проект. Но именно так поступать вовсе не рекомендуется. Желательно постепенно адаптировать код в соответствии с изменяющимися требованиями. В процессе рефакторинга кода с учетом связывания, упрощения и повышения действенности, обнаруживается, что окончательный вариант становится похожим на определенный шаблон. Если имеет место подобная ситуация, следует изменить названия классов и переменных с учетом названия шаблона, модифицировать структуру

[1] [GOF95], с. 293

кода, что позволит применять шаблон более регулярным образом. Таким образом, код "возвращается к шаблону".

В данной главе на примере несложной проблемы показано, каким образом совершенствуются проект и код. Результатом эволюционных преобразований является переход к шаблону `Observer`.

Электронные часы

Предположим, что мы имеем дело с объектом, моделирующим часы. Этот объект воспринимает миллисекундные прерывания операционной системы (известные как такты) и преобразовывает их в соответствующее показание часов. Объект определяет секунды на основе миллисекунд, минуты — на основе секунд, часы — на основе минут, дни — на основе часов и т.д. Также он располагает информацией о количестве дней в месяце и количестве месяцев в году. Имеются данные о високосных годах, а также о времени их наступления. Этот объект полностью "осведомлен" о текущем времени (рис. 24.1).

Наша задача заключается в создании электронных часов, отображающихся на рабочем столе, которые показывали бы текущее время. Для этого можно воспользоваться следующим кодом.

```
public void DisplayTime
{
  while(1)
  {
    int sec = clock.getSeconds();
    int min = clock.getMinutes();
    int hour = clock.getHours();
    showTime(hour,min,sec);
  }
}
```

Очевидно, что подобное решение не является оптимальным. В случае его реализации все свободные циклы ЦПУ будут использоваться для отображения текущего времени. В большинстве случаев это является излишним, поскольку показания часов не изменяются столь часто. Конечно, подобную методику можно использовать в случае "конструирования" каких-нибудь настенных цифровых

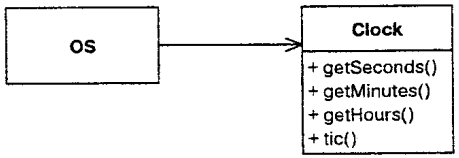


Рис. 24.1. Объект Clock

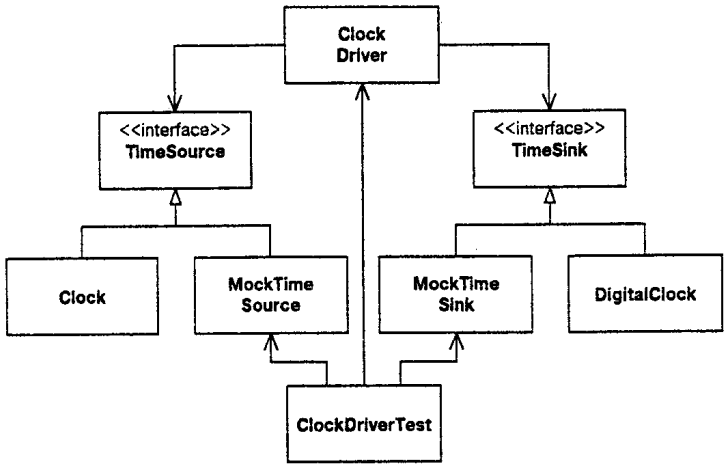


Рис. 24.2. Тестирование объекта DigitalClock

часов. Но для формирования изображений часов на рабочем столе использовать циклы ЦПУ весьма нежелательно.

Основная проблема состоит в реализации эффективного перенаправления данных от `Clock` к `DigitalClock`. Предполагается, что существует как объект `Clock`, так и объект `DigitalClock`. Между этими объектами необходимо установить связь. Это соединение можно протестировать, что позволит удостовериться, что данные, получаемые из `Clock`, совпадают с данными, пересылаемыми объекту `DigitalClock`.

Проще всего в процессе написания этого теста создать один интерфейс, претендующий на отображение функций `Clock`, а также другой интерфейс, имеющий отношение к `DigitalClock`. Затем определяются специальные тестовые объекты, реализующие эти интерфейсы, а также удостоверяющие корректную поддержку установленных между ними связей (рис. 24.2).

Объект `ClockDriverTest` связывает `ClockDriver` с двумя фиктивными объектами с помощью интерфейсов `TimeSource` и `TimeSink`. Затем производится проверка каждого из фиктивных объектов, чтобы удостовериться в том, что `ClockDriver` контролирует передачу времени из источника целевому объекту. При необходимости `ClockDriverTest` также гарантирует сохранение приемлемого уровня эффективности.

Представляет интерес включение интерфейсов в проект после завершения тестирования. Чтобы протестировать модуль, необходимо изолировать его от других модулей системы, подобно тому, как объект `ClockDriver` изолируется от `Clock` и `Digitalclock`. Рассматриваемые тесты помогают минимизировать количество связей в проектах.

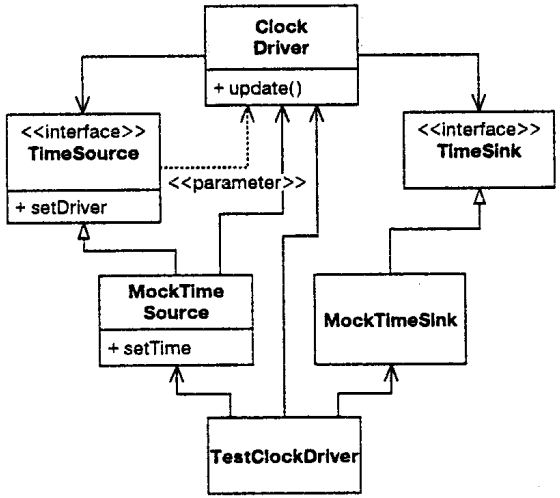


Рис. 24.3. Получение значения TimeSource для обновления ClockDriver

Каким же образом функционирует `ClockDriver`? В целях повышения степени эффективности `ClockDriver` должен проверять каждое изменение значения времени, получая доступ к объекту `TimeSource`. И только тогда значение времени передается объекту `TimeSink`. Каким же образом `ClockDriver` узнает о том, что значение времени изменилось? Можно запрашивать об этом объект `TimeSource`, но это приведет к повторному возникновению проблемы с "громоздкими" циклами ЦПУ.

Проще всего сведения об изменении времени направлять объекту `ClockDriver` с помощью объекта `Clock`. Объект `ClockDriver` можно связать с `Clock` с помощью интерфейса `TimeSource`, тогда при изменении значения времени `Clock` обновит значение `ClockDriver`. Объект же `ClockDriver`, в свою очередь, установит значение времени для `ClockSink` (рис. 24.3).

Обратите внимание на зависимость, возникающую между `TimeSource` и `ClockDriver`. Ее причина заключается в том, что аргументом метода `setDriver` является `ClockDriver`. Это очень удобно, поскольку объекты `TimeSource` должны в любом случае использовать объекты `ClockDriver`. Но следует с осторожностью относиться к любым преобразованиям этой зависимости, чтобы не потерять функциональные свойства программы.

Код из листинга 24.1 демонстрирует тестовый случай для `ClockDriver`. Обратите внимание, что в данном случае создается `ClockDriver`, а `MockTimeSource` и `MockTimeSink` привязываются к нему. Затем в исходном объекте устанавливается значение времени и ожидается, что оно "магическим образом продвинется дальше". Остальная часть кода демонстрируется в листингах 24.2–24.6.

Листинг 24.1. ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
  public ClockDriverTest(String name)
  {
    super(name);
  }

  public void testTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    ClockDriver driver = new ClockDriver(source,sink);
    source.setTime(3,4,5);
    assertEquals(3, sink.getHours());
    assertEquals(4, sink.getMinutes());
    assertEquals (5, sink.getSeconds ());

    source.setTime(7,8,9);
    assertEquals(7, sink.getHours());
    assertEquals(8, sink.getMinutes());
    assertEquals(9, sink.getSeconds());
  }
}
```

Листинг 24.2. TimeSource.java

```
public interface TimeSource
{
  public void setDriver(ClockDriver driver);
}
```

Листинг 24.3. TimeSink.java

```
public interface TimeSink
{
  public void setTime(int hours, int minutes, int seconds);
}
```

Листинг 24.4. ClockDriver.java

```
public class ClockDriver
{
  private TimeSink itsSink;

  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.setDriver(this);
    itsSink = sink;
  }

  public void update(int hours, int minutes, int seconds)
  {
    itsSink.setTime(hours, minutes, seconds);
  }
}
```


Листинг 24.5. MockTimeSource.java

```
public class MockTimeSource implements TimeSource
{
  private ClockDriver itsDriver;

  public void setTime(int hours, int minutes, int seconds)
  {
    itsDriver.update(hours, minutes, seconds);
  }

  public void setDriver(ClockDriver driver)
  {
    itsDriver = driver;
  }
}
```

Листинг 24.6. MockTimeSink.java

```
public class MockTimeSink implements TimeSink
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public int getSeconds()
  {
    return itsSeconds;
  }

  public int getMinutes()
  {
    return itsMinutes;
  }

  public int getHours()
  {
```

```
    return itsHours;
  }

  public void setTime(int hours, int minutes, int seconds)
  {
    itsHours = hours;
    itsMinutes = minutes;
    itsSeconds = seconds;
  }
}
```

Теперь попробуем усовершенствовать разработанный код. Сначала следует устранить зависимость `TimeSource` от `ClockDriver`, поскольку интерфейс `TimeSource` должен применяться любыми объектами, а не только объектами `ClockDriver`. Для этого создадим интерфейс, применяющий `TimeSource`, который сможет реализовать `ClockDriver`. Этот интерфейс назовем `ClockObserver`. Обратитесь к листингам 24.7-24.10. Измененный код выделен с помощью полужирного шрифта.

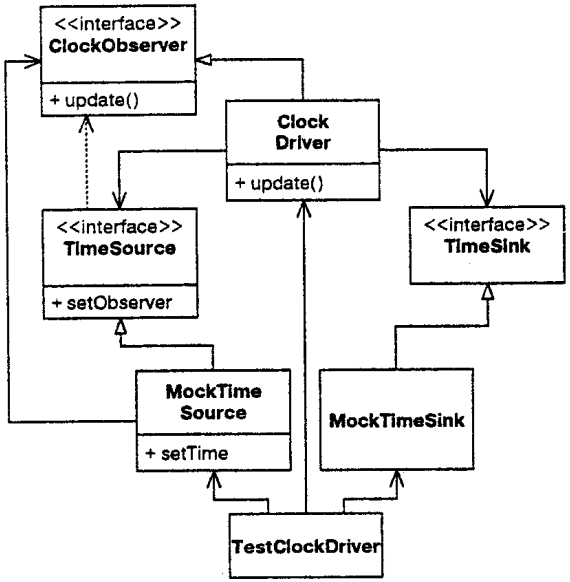


Рис. 24.4. Устранение зависимости между TimeSource и ClockDriver

Листинг 24.7. `ClockObserver.java`

```
public interface ClockObserver
{
  public void update(int hours, int minutes, int seconds);
}
```

Листинг 24.8. `ClockDriver.java`

```
public class ClockDriver implements ClockObserver
{
  private TimeSink itsSink;

  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.setObserver(this);
    itsSink = sink;
  }
  public void update(int hours, int minutes, int seconds)
  {
    itsSink.setTime(hours, minutes, seconds);
  }
}
```

Листинг 24.9. `TimeSource.java`

```
public interface TimeSource
{
  public void setObserver(ClockObserver observer);
}
```

Листинг 24.10. `MockTimeSource.java`

```
public class MockTimeSource implements TimeSource
{
  private ClockObserver itsObserver;

  public void setTime(int hours, int minutes, int seconds)
  {
    itsObserver.update(hours, minutes, seconds);
  }
  public void setObserver(ClockObserver observer)
  {
    itsObserver = observer;
  }
}
```

Получилось значительно лучше. Никто не сможет воспользоваться объектом TimeSource. Для этого следует создать копию ClockObserver и вызвать SetObserver, передавая в качестве аргумента значение.

Желательно для отсчета времени использовать более одного объекта TimeSink. Одно значение используется электронными часами. Другое значение применяется для поддержки времени службы напоминания. Еще одно предназначено для запуска службы ночного резервного копирования. В общем, один объект TimeSource должен поддерживать значение времени для нескольких объектов TimeSink.

Итак, изменим конструктор ClockDriver, обращающийся к объекту TimeSource, затем включим метод addTimeSink, который позволит в любое удобное для вас время создавать экземпляры TimeSink.

В этом случае также получаются два нежелательных следствия. При вызове setObserver необходимо указывать TimeSource для объекта ClockObserver. Также необходимо указывать ClockDriver для экземпляров TimeSink. Можно ли исключить эти моменты?

Рассматривая ClockObserver и TimeSink, замечаем, что они применяют метод setTime. То есть TimeSink образует копию ClockObserver. Если выполнить это, тестовая программа создает MockTimeSink и вызывает setObserver для TimeSource. Можно избавиться сразу и от ClockDriver (и от TimeSink)! Код из листинга 24.11 демонстрирует изменения в ClockDriverTest.

Листинг 24.11. ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
  public ClockDriverTest(String name)
  {
    super(name);
  }

  public void testTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    source.setObserver(sink);

    source.setTime(3,4,5);
    assertEquals(3, sink.getHours());
    assertEquals(4, sink.getMinutes());
    assertEquals(5, sink.getSeconds());

    assertEquals(7, sink.getHours());
    assertEquals(8, sink.getMinutes());
    assertEquals(9, sink.getSeconds());
  }
}
```


Следовательно, MockTimeSink должен создавать копию ClockObserver вместо TimeSink. Обратите внимание на листинг 24.12. Изменения оказались вполне функциональными. Почему ClockDriver необходимо разместить на первом месте? На рис. 24.5 показана соответствующая UML-диаграмма.

Листинг 24.12. MockTimeSink.java

```
public class MockTimeSink implements ClockObserver
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public int getSeconds()
  {
    return itsSeconds;
  }

  public int getMinutes()
  {
    return itsMinutes;
  }

  public int getHours()
  {
    return itsHours;
  }

  public void update(int hours, int minutes, int seconds)
  {
    itsHours = hours;
    itsMinutes = minutes;
    itsSeconds = seconds;
  }
}
```

Очевидно, что данный вариант кода значительно проще.

Теперь несколько объектов TimeSink можно обрабатывать путем изменения функции setObserver на registerObserver. При этом гарантируется, что все зарегистрированные экземпляры ClockObserver отобразятся в списке, а также модифицируются соответствующим образом. Теперь в тестовую программу следует внести другие изменения. Код из листинга 24.13 демонстрирует эти

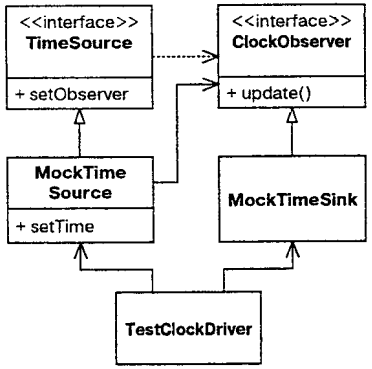


Рис. 24.5. Результат устранения ClockDriver и TimeSink

изменения. Также в тестовой программе выполнен небольшой рефакторинг, что значительно сократит ее размеры и облегчит просмотр кода.

Листинг 24.13. ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
  private MockTimeSource source;
  private MockTimeSink sink;

  public ClockDriverTest(String name)
  {
    super(name);
  }

  public void setUp()
  {
    source = new MockTimeSource();
    sink = new MockTimeSink();
    source.registerObserver(sink);
  }

  private void assertSinkEquals(
      MockTimeSink sink, int hours, int minutes, int seconds)
  {
    assertEquals(hours, sink.getHours());
    assertEquals(minutes, sink.getMinutes());
    assertEquals(seconds, sink.getSeconds());
  }

  public void testTimeChange()
  {
    source.setTime(3,4,5);
    assertSinkEquals(sink, 3,4,5);
    source.setTime(7,8,9);
    assertSinkEquals(sink, 7,8,9);
  }

  public void testMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.registerObserver(sink2);

    source.setTime(12,13,14);
    assertSinkEquals(sink, 12,13,14);
    assertSinkEquals(sink2, 12,13,14);
  }
}
```


Для выполнения рассматриваемых заданий следует внести небольшие изме нения. Преобразуем MockTimeSource так, чтобы в него вошли все зарегистри рованные наблюдатели Vector. Затем, когда изменится значение времени, вы полним итерацию с использованием Vector и вызовем update для всех зареги стрированных ClockObservers. Код из листингов 24.14 и 24.15 демонстрируе эти изменения. На рис. 24.6 показана соответствующая UML-диаграмма.

Листинг 24.14. TimeSource.java

```
public interface TimeSource
{
  public void registerObserver(ClockObserver observer);
}
```

Листинг 24.15. MockTimeSource.java

```
import java.util.*;

public class MockTimeSource implements TimeSource
{
  private Vector itsObservers = new Vector();

  public void setTime(int hours, int minutes, int seconds)
  {
    Iterator i = itsObservers.iterator();
    while (i.hasNext())
    {
      ClockObserver observer = (ClockObserver) i.next();
      observer.update(hours, minutes, seconds);
    }
```

```
  public void registerObserver(ClockObserver observer)
  {
    itsObservers.add(observer);
  }
}
```

Все отлично, но программиста не устраивает, что MockTimeSource выполняет регистрацию и обновление. Отсюда следует, что Clock и все остальные производные классы TimeSource должны дублировать код, относящийся к регистрации и обновлению. Автор полагает, что Clock не должен иметь отношения к регистрации и обновлению. Также не вполне уместно дублирование кода. Итак, желательно переместить весь подготовительный материал в TimeSource. Естественно, TimeSource превратится из интерфейса в класс.

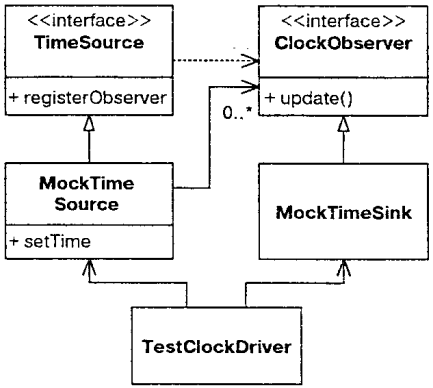


Рис. 24.6. Обработка нескольких объектов TimeSink

В результате получим резкое сокращение размеров MockTimeSource. Имеющие место изменения отображаются в листингах 24.16 и 24.17, а также на рис. 24.7.

Листинг 24.16. TimeSource.java

```
import java.util.*;

public class TimeSource
{
  private Vector itsObservers = new Vector();

  protected void notify(int hours, int minutes, int seconds)
  {
    Iterator i = itsObservers.iterator();
    while (i.hasNext())
    {
```



```
      ClockObserver observer = (ClockObserver) i.next();
      observer.update(hours, minutes, seconds);
    }
  }

  public void registerObserver(ClockObserver observer)
  {
    itsObservers.add(observer);
  }
}
```

Листинг 24.17. MockTimeSource.java

```
public class MockTimeSource extends TimeSource
{
  public void setTime(int hours, int minutes, int seconds)
  {
    notify(hours, minutes, seconds);
  }
}
```

Это уже небольшое достижение. Теперь можно создавать производные классы на основе TimeSource. Для этого необходимо только получить наблюдателей, обновленных с учетом обращения к notify. Но именно этого желательно избегать. MockTimeSource наследуется непосредственно из TimeSource. Значит, Clock также должен быть производным классом от TimeSource. Почему Clock обязательно должен зависеть от процессов регистрации и обновления? Он является лишь классом, который получает информацию о времени. Зависимость от TimeSource кажется обязательной и в то же время нежелательной.

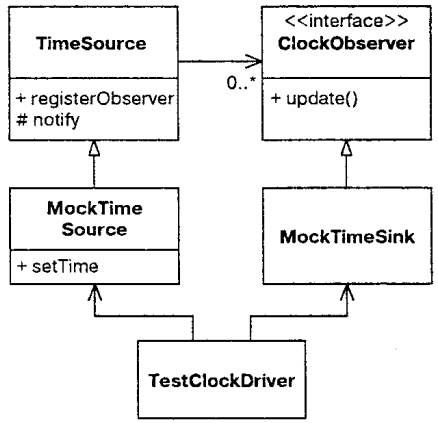


Рис. 24.7. Перемещение процедур регистрации и обновления в TimeSource

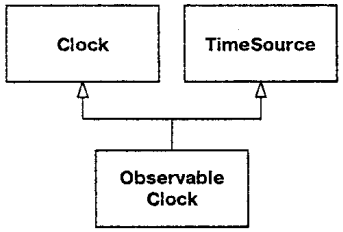


Рис. 24.8. Применение множественного наследования в C++ для отделения Clock от TimeSource

Известен способ разрешения этих затруднений при работе в C++. Как для `TimeSource`, так и для `Clock` создается подкласс под названием `ObservableClock`. Перекрывается использование `tic` и `setTime` в `ObservableClock` для вызова `tic`, либо `setTime` в `Clock` и затем вызывается `notify` из `TimeSource`. Обратите внимание на листинг 24.18 и рис. 24.8.

Листинг 24.18. `ObservableClock.cc (C++)`

```
class ObservableClock : public Clock, public TimeSource
{
  public:
    virtual void tic()
    {
      Clock::tic();
      TimeSource::notify(getHours(), getMinutes(), getSeconds())
    }

    virtual void setTime(int hours, int minutes, int seconds)
    {
      Clock::setTime(hours, minutes, seconds);
      TimeSource::notify(hours, minutes, seconds);
    }
};
```

К сожалению, подобная возможность отсутствует в Java, поскольку в этом языке не используется множественное наследование классов. Итак, при работе в Java либо оставим все без изменения, либо применим прием делегирования. Реализация этого метода показана в листингах 24.19–24.21 и на рис. 24.9.

Листинг 24.19. `TimeSource.java`

```
public interface TimeSource
{
  public void registerObserver(ClockObserver observer);
}
```


Листинг 24.20. `TimeSourceImplementation.java`

```
import java.util.*;

public class TimeSourceImplementation
{
  private Vector itsObservers = new Vector();
  public void notify(int hours, int minutes, int seconds)
  {
    Iterator i = itsObservers.iterator();
    while (i.hasNext())
    {
      ClockObserver observer = (ClockObserver) i.next();
      observer.update(hours, minutes, seconds);
    }
  }

  public void registerObserver(ClockObserver observer)
  {
    itsObservers.add(observer);
  }
}
```

Листинг 24.21. `MockTimeSource.java`

```
public class MockTimeSource implements TimeSource
{
  TimeSourceImplementation tsImp =
    new TimeSourceImplementation();

  public void registerObserver(ClockObserver observer)
  {
    tsImp.registerObserver(observer);
  }

  public void setTime(int hours, int minutes, int seconds)
  {
    tsImp.notify(hours, minutes, seconds);
  }
}
```

Обратите внимание, что класс `MockTimeSource` реализует `TimeSource` и содержит ссылку на экземпляр `TimeSourceImplementation`. Заметим также, что все вызовы, направляемые методу `registerObserver` из `MockTimeSource`, делегируются объекту `TimeSourceImplementation`. Итак, `MockTimeSource.setTime` вызывает `notify` с помощью экземпляра `TimeSourceImplementation`.

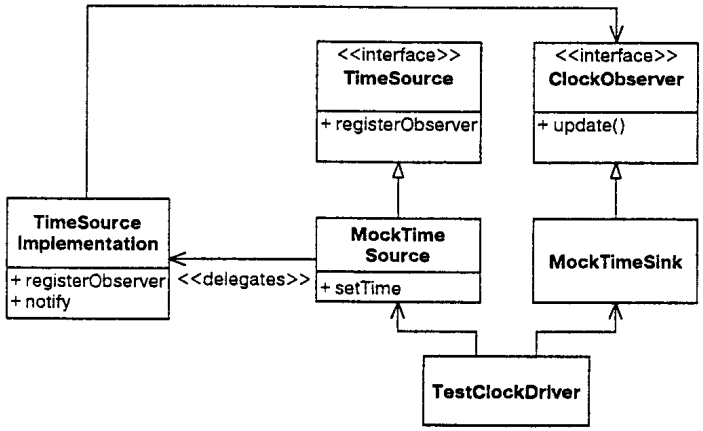


Рис. 24.9. Выполнение делегирования с помощью `Observer` в Java

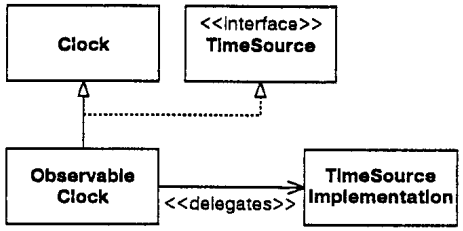


Рис. 24.10. Выполнение делегирования для `ObservableClock`

Хотя это и не совсем правильно, но преимущество в данном случае состоит в том, что `MockTimeSource` не расширяет класс. То есть при создании `ObservableClock` можно расширить `Clock`, реализовать `TimeSource` и делегировать его `TimeSourceImplementation` (рис. 24.10.) Подобный подход решает проблему с `Clock`, связанную с регистрацией и обновлением, но цена этого решения достаточно велика.

Итак, вернемся к тому "положению вещей", которое отображено на рис. 24.7, до формирования рассматриваемых представлений. Просто согласимся с тем, что `Clock` зависит от всего процесса регистрации и обновления.

При этом `TimeSource` — не совсем удачное название для данного класса. По сравнению с тем периодом, когда применялся `ClockDriver`, появилось множество изменений. Необходимо заменить название с учетом процессов регистрации и обновления. Шаблон `Observer` вызывает класс `Subject`. Поскольку следует учесть специфику задачи, назовем его `TimeSubject`, но это название не отвечает интуитивным представлениям. Можно применить известный моникер Java `Observable`, но неужели `TimeObservable` звучит гармоничнее? Ничуть.

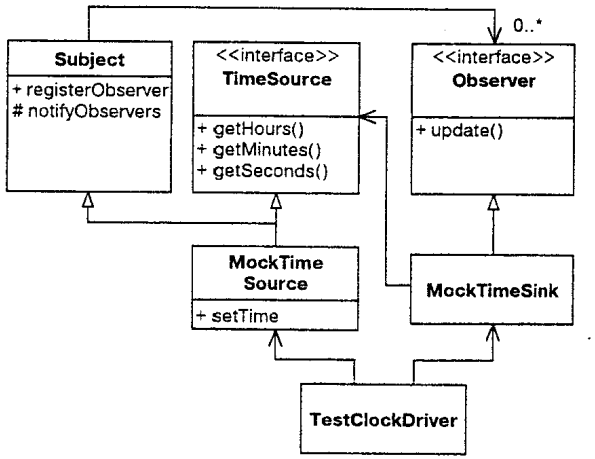


Рис. 24.11. Заключительная версия `Observer`, примененная к `MockTimeSource` и `MockTimeSink`

Возможно, затруднения вытекают из специфики наблюдателя "модели для толчка"[2]. При переходе к "модели для толчка" можно обобщить данный класс. Затем название `TimeSource` изменяется на `Subject`, и каждый, кто знаком с шаблоном `Observer`, будет лучше осведомлен о том, с чем он имеет дело.

Рассматриваемую возможность можно считать весьма неплохой. Вместо того чтобы передавать значение времени с помощью методов `notify` и `update`, можно устроить так, что `TimeSink` запросит `MockTimeSource` о показаниях времени. Нежелательно, чтобы `MockTimeSink` содержал сведения о `MockTimeSource`, поэтому создадим интерфейс, применяемый `MockTimeSink` для получения информации о времени. `MockTimeSource` (и `Clock`) реализует этот интерфейс. Данный интерфейс можно назвать `TimeSource`.

Заключительная версия кода и соответствующая UML-диаграмма представлены на рис. 24.11 и в листингах 24.22–24.27.

Листинг 24.22. `ObserverTest.java`

```
import junit.framework.*;

public class ObserverTest extends TestCase
{
```

[2]Наблюдатели "выталкивающей модели" перемещают данные от субъекта к наблюдателю, используя для этого методы `notify` и `update`. Наблюдатели "выталкивающей модели" не передают каких-либо сведений с помощью методов `notify` и `update`. Они зависят от объекта наблюдения, запрашивая наблюдаемый объект при получении обновления. См. [GOF95].

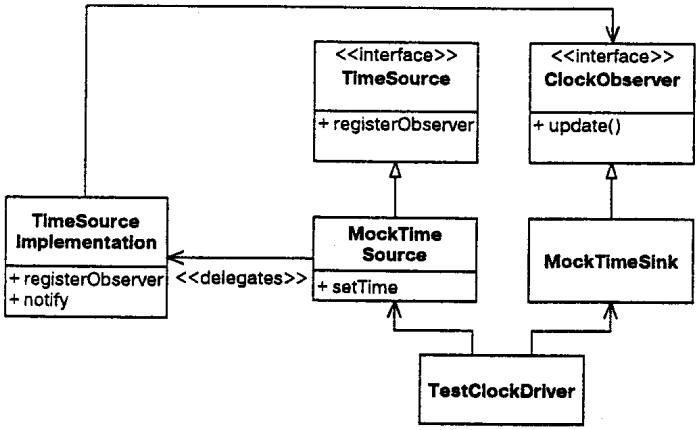


Рис. 24.9. Выполнение делегирования с помощью `Observer` в Java

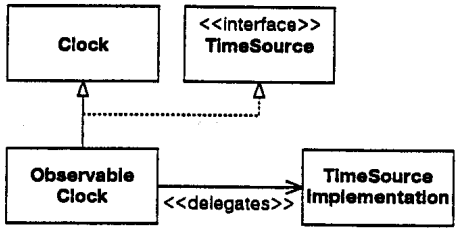


Рис. 24.10. Выполнение делегирования для `ObservableClock`

Хотя это и не совсем правильно, но преимущество в данном случае состоит в том, что `MockTimeSource` не расширяет класс. То есть при создании `ObservableClock` можно расширить `Clock`, реализовать `TimeSource` и делегировать его `TimeSourceImplementation` (рис. 24.10.) Подобный подход решает проблему с `Clock`, связанную с регистрацией и обновлением, но цена этого решения достаточно велика.

Итак, вернемся к тому "положению вещей", которое отображено на рис. 24.7, до формирования рассматриваемых представлений. Просто согласимся с тем, что `Clock` зависит от всего процесса регистрации и обновления.

При этом `TimeSource` — не совсем удачное название для данного класса. По сравнению с тем периодом, когда применялся `ClockDriver`, появилось множество изменений. Необходимо заменить название с учетом процессов регистрации и обновления. Шаблон `Observer` вызывает класс `Subject`. Поскольку следует учесть специфику задачи, назовем его `TimeSubject`, но это название не отвечает интуитивным представлениям. Можно применить известный моникер Java `Observable`, но неужели `TimeObservable` звучит гармоничнее? Ничуть.

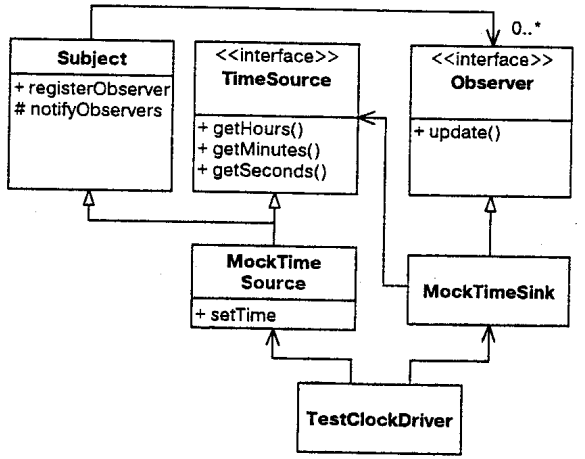


Рис. 24.11. Заключительная версия `Observer`, примененная к `MockTimeSource` и `MockTimeSink`

Возможно, затруднения вытекают из специфики наблюдателя "модели для толчка"[2]. При переходе к "модели для толчка" можно обобщить данный класс. Затем название `TimeSource` изменяется на `Subject`, и каждый, кто знаком с шаблоном `Observer`, будет лучше осведомлен о том, с чем он имеет дело.

Рассматриваемую возможность можно считать весьма неплохой. Вместо того чтобы передавать значение времени с помощью методов `notify` и `update`, можно устроить так, что `TimeSink` запросит `MockTimeSource` о показаниях времени. Нежелательно, чтобы `MockTimeSink` содержал сведения о `MockTimeSource`, поэтому создадим интерфейс, применяемый `MockTimeSink` для получения информации о времени. `MockTimeSource` (и `Clock`) реализует этот интерфейс. Данный интерфейс можно назвать `TimeSource`.

Заключительная версия кода и соответствующая UML-диаграмма представлены на рис. 24.11 и в листингах 24.22–24.27.

Листинг 24.22. `ObserverTest.java`

```
import junit.framework.*;

public class ObserverTest extends TestCase
{
```

[2]Наблюдатели "выталкивающей модели" перемещают данные от субъекта к наблюдателю, используя для этого методы `notify` и `update`. Наблюдатели "выталкивающей модели" не передают каких-либо сведений с помощью методов `notify` и `update`. Они зависят от объекта наблюдения, запрашивая наблюдаемый объект при получении обновления. См. [GOF95].

```
  private MockTimeSource source;
  private MockTimeSink sink;

  public ObserverTest(String name)
  {
    super(name);
  }

  public void setUp()
  {
    source = new MockTimeSource();
    sink = new MockTimeSink(source);
    source.registerObserver(sinkprivate void assertSinkEquals(
  }

  private void assertSinkEquals(
    MockTimeSink sink, int hours, int minutes, int seconds)
  {
    assertEquals(hours, sink.getHours());
    assertEquals(minutes, sink.getMinutes());
    assertEquals(seconds, sink.getSeconds());
  }

  public void testTimeChange()
  {
    source.setTime(3,4,5);
    assertSinkEquals(sink, 3,4,5);

    source.setTime(7,8,9);
    assertSinkEquals(sink, 7,8,9);
  }

  public void testMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink(source);
    source.registerObserver(sink2);

    source.setTime(12,13,14);
    assertSinkEquals(sink, 12,13,14);
    assertSinkEquals(sink2, 12,13,14);
  }
}
```

Листинг 24.23. Observer.java

```
public interface Observer
{
  public void update();
}
```


Листинг 24.24. Subject.java

```
import java.util.*;

public class Subject
{
  private Vector itsObservers = new Vector();

  protected void notifyObservers()
  {
    Iterator i = itsObservers.iterator();
    while (i.hasNext())
    {
      Observer observer = (Observer) i.next();
      observer.update ();
    }
  }

  public void registerObserver(Observer observer)
  {
    itsObservers.add(observer);
  }
}
```

Листинг 24.25. TimeSource.java

```
public interface TimeSource
{
  public int getHours();
  public int getMinutes();
  public int getSeconds();
}
```

Листинг 24.26. MockTimeSource.java

```
public class MockTimeSource extends Subject
      implements TimeSource
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public void setTime(int hours, int minutes, int seconds)
  {
    itsHours = hours;
    itsMinutes = minutes;
    itsSeconds = seconds;
    notifyObservers();
```

```
  }

  public int getHours()
  {
    return itsHours;
  }

  public int getMinutes()
  {
    return itsMinutes;
  }

  public int getSeconds()
  {
    return itsSeconds;
  }
}
```

Листинг 24.27. `MockTimeSink.java`

```
public class MockTimeSink implements Observer
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;
  private TimeSource itsSource;

  public MockTimeSink(TimeSource source)
  {
    itsSource = source;
  }

  public int getSeconds()
  {
    return itsSeconds;
  }

  public int getMinutes()
  {
    return itsMinutes;
  }

  public int getHours()
  {
    return itsHours;
  }

  public void update ()
  {
    itsHours = itsSource.getHours();
```



```
    itsMinutes = itsSource.getMinutes();
    itsSeconds = itsSource.getSeconds();
  }
}
```

Резюме

Мы начали рассмотрение с проблемы, возникшей при разработке, и в результате вполне обоснованной эволюции остановились на использовании "канонического" шаблона `Observer`. Читатель может удивиться, что обращение к шаблону было предопределено автором.

Если вы знакомы с шаблонами проектирования, то при появлении проблемы мысль об их использовании сразу же всплывает в вашем представлении. Вопрос в том, реализовать ли шаблон непосредственно или же предпочесть постепенное развитие в этом направлении. Данная глава подтверждает большую действенность второго варианта. Постепенно становится ясно, что код эволюционирует в направлении шаблона `Observer`, изменились названия, и код принял "каноническую" форму.

Применение диаграмм в данной главе

Часть диаграмм создана для удобства читателей.

Обычно эти диаграммы описывали промежуточные шаги. Имеет ли смысл детализировать подобные диаграммы? Если вам необходимо обосновать свое решение, как это делает автор данной книги, применение диаграмм весьма удобно. Но обычно нет необходимости документировать эволюционный путь, отнявший несколько рабочих часов. Как правило, эти диаграммы отражают "мимолетные состояния" и не сохраняются на долгое время, т.е. код выполняет роль документации. На более высоких уровнях разработки этот вывод не всегда справедлив.

Шаблон Observer

"Каноническая" форма этого шаблона показана на рис. 24.12. В этом примере `Clock` наблюдается с помощью `DigitalClock`. Объект `DigitalClock` поддерживает регистрацию с помощью интерфейса `Subject` из `Clock`. Объект `Clock` вызывает метод `notify` из `Subject`, когда бы не изменилось значение времени. Метод `notify` из `Subject` вызывает метод `update` для каждого зарегистрированного `Observer`. Поэтому `DigitalClock` получит и обновит сообщение при любом изменении значения времени. Используется возможность, состоящая в запросе `Clock` по поводу времени и отображении этого значения.

Шаблон `Observer` используется повсеместно. Наблюдателей можно регистрировать с помощью объектов всех видов, а не записывать эти объекты исклю-

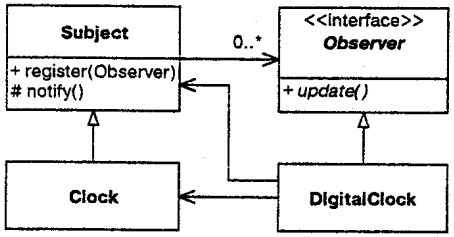


Рис. 24.12. "Каноническая" модель "вытягивания" Observer

чительно с применением вызовов. Это обстоятельство помогает управлять зависимостями, которые "легко переходят в крайности". Чрезмерное использование шаблона `Observer` приводит к усложнению систем, что затрудняет осмысление их функциональных возможностей и отслеживание их работы.

"Тяни-толкай". Шаблон `Observer` характеризуется двумя основными моделями. На рис. 24.13 показана *модель "вытягивания"* `Observer`. Свое название она получила на основании того факта, что после получения обновленного сообщения `DigitalClock` "вытягивает" сведения о времени из объекта `Clock`.

Преимуществом модели *"вытягивания"* является простота реализации, а также тот факт, что классы `Subject` и `Observer` могут быть стандартными, повторно используемыми элементами библиотеки. Но представьте, что запись служащего содержит тысячу полей, а вы только что получили сообщение `update`. В какое из тысяч полей внести это сообщение?

Когда `update` вызывается для `ClockObserver`, ответ очевиден. `ClockObserver` "тянет" значение времени из `Clock` и отображает его. Но если обновление вызывается на `EmployeeObserver`, ответ не столь очевиден. Неизвестно, как поступить. Возможно, изменилось имя служащего, или он получил свою зарплату. Может быть, он станет новым боссом. Или изменится счет в банке. Тут необходим дополнительный ориентир.

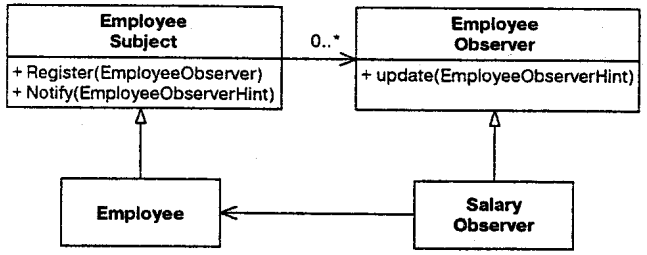


Рис. 24.13. "Толкающая" модель Observer

Такую помощь можно получить на основе формы модели "тяни-толкай" шаблона `Observer`. Структура модели "толчка" для наблюдателя показана на рис. 24.13. Обратите внимание, что аргумент получают оба метода: как `notify`, так и `update`. Аргумент передается из `Employee` к `SalaryObserver` с помощью методов `notify` и `update`. Также `SalaryObserver` получает подсказку об изменении записи `Employee`.

Аргумент `EmployeeObserverHint` методов `notify` и `update` может представлять собой определенное численное значение, являться строкой или более сложной структурой данных, включающей старое и новое значение некоторого поля. Какую бы природу не имел этот аргумент, он передается наблюдателю.

Выбор между двумя различными моделями `Observer` полностью зависит от уровня сложности наблюдаемого объекта. Если наблюдаемый объект имеет достаточно сложную природу, а наблюдатель должен располагать дополнительными возможностями, применяется модель "толчка". Если наблюдаемый объект достаточно прост, удобно обращаться к "тянущей" модели.

Каким образом шаблон Observer учитывает принципы ООП?

Шаблон `Observer` тесно связан с использованием принципа Open-Closed Principle (OCP). Обоснованием для применения шаблона служит тот факт, что новые наблюдаемые объекты можно добавлять без внесения изменений в исходный объект наблюдения. Объект наблюдения остается закрытым.

Возвращаясь к рис. 24.12, можно заключить, что `Clock` заменяется на `Subject`, а `DigitalClock` — на `Observer`. Поэтому соблюдается принцип подстановки Лискоу (Liskov Substitution Principle, LSP).

Класс `Observer` является абстрактным, причем от него зависит конкретный `DigitialClock`. Также от него зависят конкретные методы `Subject`. Следовательно, в этом случае применяется принцип инверсии зависимостей (DIP, Dependency-Inversion Principle). Можно предположить, что поскольку `Subject` не содержит абстрактных классов, зависимость между `Clock` и `Subject` нарушает принцип DIP. Но `Subject` является классом, на основе которого не формируются экземпляры. Он имеет смысл только в контексте производного класса. Поэтому `Subject` является *логически* абстрактным, даже если он не располагает абстрактными методами. Абстрактность `Subject` можно усилить, придавая ему чисто виртуальный деструктор в C++ или превращая конструкторы в защищенные.

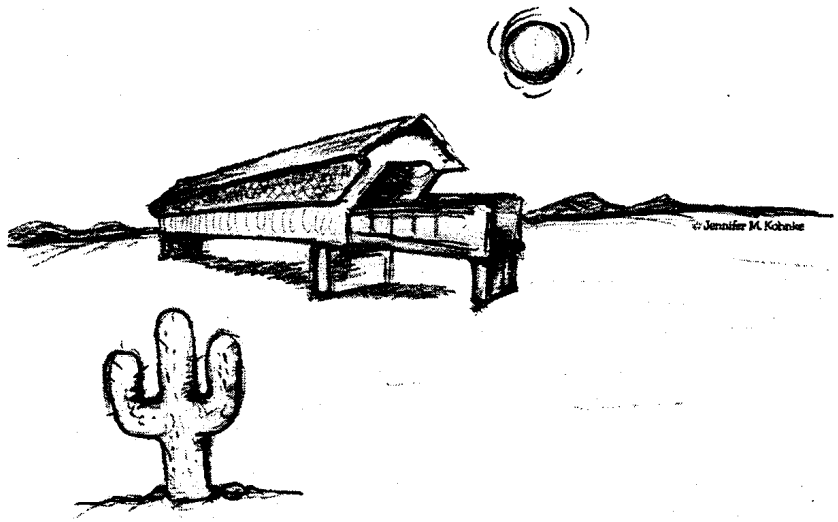
На рис. 24.11 приведены подсказки, связанные с принципом отделения интерфейса (ISP, Interface-Segregation Principle). Классы `Subject` и `TimeSource` производят разделение клиентов `MockTimeSource`, поддерживая для каждого из них специализированные интерфейсы.

Литература

1. Gamma и др. *Design Patterns.* Addison-Wesley, 1995.
2. Martin C. и др. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

25

Некоторые примеры из практики: шаблоны Abstract Server, Adapter и Bridge



Политики все одинаковы. Они обещают построить мост даже там, где нет реки.

Никита Хрущев

В середине 90-х годов прошлого века я принимал участие в дискуссиях, развернувшихся в группе новостей `comp.object`. Программисты, отправлявшие сообщения в эту группу новостей, вели оживленные споры о различных стратеги-

можно было бы оценить позицию каждого, кто принимал участие в этих дискуссиях. Итак, мы остановили свой выбор на простейшей проектной проблеме, чтобы представлять свои наиболее приемлемые решения.

Сама проблема была чрезвычайно простой. Мы решили разработать программу, моделирующую работу простой настольной лампы. Настольная лампа состоит из выключателя и лампочки. Можно сделать запрос выключателю о том, был ли он включен, а лампочке можно дать команду включиться или выключиться. Довольно простая проблема, не правда ли?

Дискуссии о методах решения данной проблемы длились несколько месяцев. Каждый из участников дебатов настаивал на превосходстве своего собственного стиля проекта. Некоторые предлагали простой подход, который заключался в наличии только объектов выключателя и лампочки. Другие полагали, что в качестве объекта выступает электричество. Один программист даже предложил ввести объект сетевого шнура.

Несмотря на нелепость некоторых из приведенных аргументов, модель проекта на самом деле представляет собой весьма интересный предмет исследования. Обратите внимание на рис. 25.1. Можно, конечно же, сделать этот проект действующим. Объект `Switch` опрашивает состояние текущего выключателя и отправляет соответствующие сообщения `turnOn` и `turnOff` объекту `Light`.

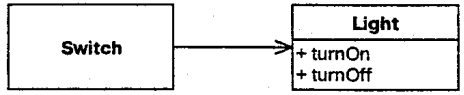


Рис. 25.1. Простая настольная лампа

Каковы недостатки описанного проекта?

Очевидно, что здесь нарушаются два принципа проектирования: принцип инверсии зависимостей (DIP) и принцип "открытия-закрытия" (OCP). Нарушение принципа инверсии зависимостей легко прослеживается; зависимость от `Switch` до `Light` на самом деле представляет собой зависимость от конкретного класса. В соответствии с принципом инверсии зависимостей, лучше выбирать зависимости из абстрактных классов. Нарушение принципа "открытия-закрытия" имеет немного опосредованный характер, но в то же время его последствия более существенны. Нас не устраивает этот проект в том плане, что приходится перетаскивать объект `Light` туда, где требуется объект `Switch`. Не так просто расширить объект `Switch` таким образом, чтобы управлять объектами, отличными от `Light`.

Шаблон Abstract Server

Может возникнуть мысль о возможности наследования подкласса из объекта `Switch`, который контролирует объекты, отличные от `Light`, как показано на

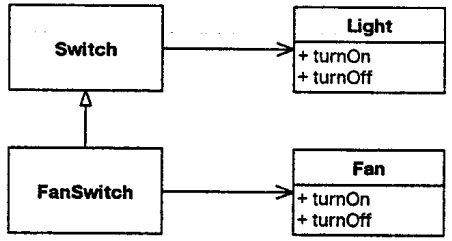


Рис. 25.2. Некорректный способ расширения объекта Switch

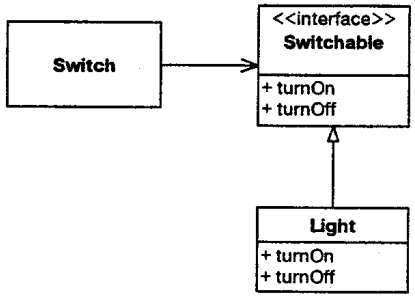


Рис. 25.3. Решение проблемы настольной лампы с помощью Abstract Server

рис. 25.3. Однако проблема этим не решается, поскольку `FanSwitch` все еще наследует зависимость от объекта `Light`. При любом использовании `FanSwitch` понадобится объект `Light`. В любом случае именно эти отношения зависимостей также приводят к нарушению принципа инверсии зависимостей.

В целях разрешения описанной проблемы активизируется простейший из всех шаблонов проектирования: `Abstract Server`. (рис. 25.2). Путем формирования интерфейса между объектами `Switch` и `Light` можно управлять всем, что реализует этот интерфейс. Благодаря этому сразу же достигается соответствие принципу инверсии зависимостей и принципу "открытия-закрытия".

Кому принадлежит интерфейс?

Обратите внимание, что интерфейс именуется в соответствии с применяемым клиентом. В данном случае он получил имя `Switchable`, а не `ILight`. Мы уже говорили об этом и, возможно, еще раз обратимся к этому вопросу. Интерфейсы принадлежат клиенту, а не производному классу. Логическая связь между клиентом и интерфейсом сильнее, чем логическая связь между интерфейсом и его производными классами. Она настолько сильна, что нет смысла развертывать `Switchable` без `Light`. Степень логической взаимосвязи превышает степень

физической взаимосвязи. Наследование обеспечивает более сильную физическую взаимосвязь, чем ассоциация.

В начале 90-х годов прошлого века физическая взаимосвязь обычно считалась доминирующей. Написанные в то время авторитетные книги рекомендовали размещать иерархии наследования вместе в одном и том же физическом пакете. Такое решение казалось оправданным, поскольку наследование представляет собой очень прочную физическую связь. Однако за последнее десятилетие мы узнали, что физическая сила наследования не настолько реальна, как казалось, и что иерархии наследования не должны упаковываться вместе. Вместо этого формируется тенденция упаковки клиентов вместе с контролируемыми ими интерфейсами.

Различная сила, присущая логическим и физическим связям, характерна для языков, обладающих статическими данными типа C++ и Java. Языки с динамическими данными типа Smalltalk, Python и Ruby не приводят к расхождениям, поскольку они не используют наследование в целях достижения полиморфного поведения.

Шаблон Adapter

На рис. 25.3 представлена небольшая проектная проблема. Эта проблема заключается в потенциальном нарушении принципа персональной ответственности (SRP). Мы связали вместе два объекта `Light` и `Switchable`, которые могут не изменяться в силу одних и тех же причин. Что случится, если мы не сможем добавить отношения зависимостей в `Light`? Что, если бы мы приобрели объект `Light` у стороннего производителя, не располагая при этом исходным кодом? Или что было бы, если бы какой-то другой класс контролировал объект `Switch`, но который невозможно наследовать от объекта `Switchable`? Рассмотрим шаблон `Adapter`[1].

На рис. 25.4 показаны возможные пути применения шаблона `Adapter` для разрешения проблемы. Причем `Adapter` является производным от `Switchable` и передается объекту `Light`. В результате проблема решается однозначно. Теперь любой объект может включаться или выключаться, контролируясь объектом `Switch`. Все, что необходимо сделать — создать соответствующий адаптер. На самом деле, объекту даже не нужны методы `turnOn` и `turnOff`, принадлежащие объекту `Switchable`. Адаптер может быть *приспособлен* к интерфейсу объекта.

Не так уж просто разрабатывать адаптеры. Прежде всего необходимо создать новый класс, создать экземпляр адаптера, связав с ним адаптированный объект. Затем при каждой активизации адаптера необходимо "платить дань" за время и пространство, необходимые для делегирования. Очевидно, что нет смысла по-

[1] Шаблон `Adapter` уже рассматривался (рис. 10.2 и 10.3).

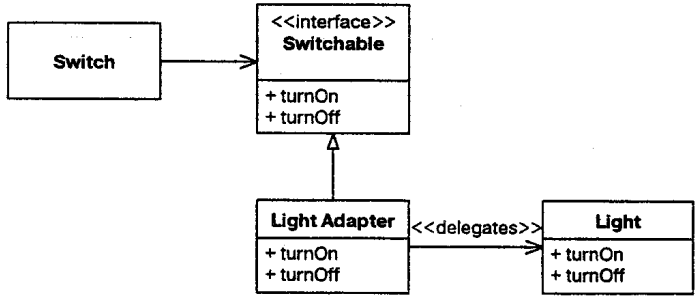


Рис. 25.4. Решение проблемы настольной лампы с помощью шаблона Adapter

стоянно прибегать к помощи адаптеров. Решение в виде `Abstract Server` вполне приемлемо во многих ситуациях. На самом деле, довольно неплохим является даже исходное решение, представленное на рис. 25.1, но только в том случае, если вы *не знаете* о существовании других объектов, контролирующих объект `Switch`.

Классовая форма шаблона Adapter

Класс `LightAdapter`, представленный на рис. 25.4, известен как *объектная форма адаптера*, представленного на рис. 25.5. В этой форме объект адаптера наследуется как из интерфейса `Switchable`, так и из класса `Light`. Эта форма немного более эффективна, чем объектная форма, а также проще в применении за счет использования высокой степени связывания для наследования.

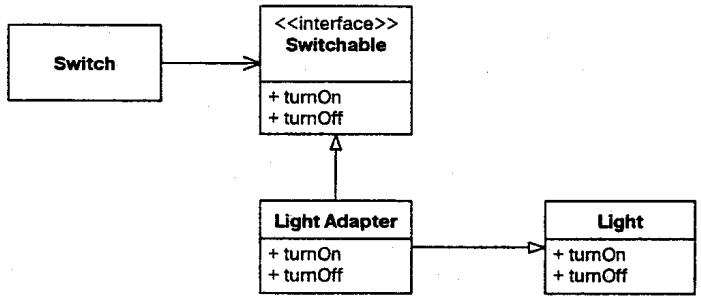


Рис. 25.5. Решение проблемы настольной лампы с помощью шаблона Adapter

Модемная проблема, шаблон Adapter и принцип LSP

Рассмотрим ситуацию, представленную на рис. 25.6. Мы имеем большое количество клиентов модемов, использующих интерфейс `Modem`. Интерфейс `Modem`

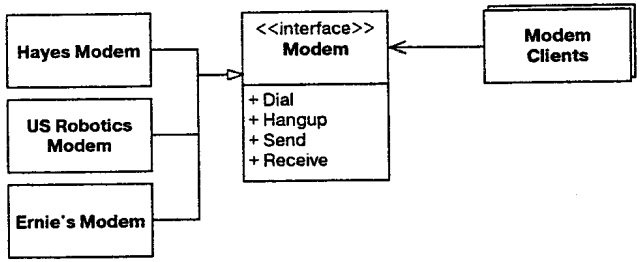


Рис. 25.6. Модемная проблема

применяется несколькими производными классами, включая `HayesModem`, `USRoboticsModem` и `EarniesModem`. Это довольно обычная ситуация. Она неплохо соответствует принципам "открытия-закрытия", наследования зависимостей и персональной ответственности (OCP, LSP, и DIP). Клиенты модемов, имея дело с новыми видами модемов, не подвергаются их влиянию. Предположим, сотни клиентов модемов успешно использовали интерфейс `Modem`.

Предположим теперь, что заказчики сформулировали новое требование. Существуют модемы без функции набора номера. Они называются выделенными, поскольку расположены на обоих концах выделенной линии[2]. Разработано несколько новых приложений, которые используют выделенные модемы, не поддерживая коммутируемый режим. Будем называть их `DedUsers`. Тем не менее, наши заказчики хотели бы, чтобы все текущие клиенты модемов могли использовать выделенные модемы. Заказчики объясняют, что не хотят изменять сотни приложений клиентских модемов, поэтому будет выделена отдельная команда дозвона по фиктивным телефонным номерам.

Если есть возможность выбора, мы можем воспользоваться ею после завершения проектирования системы, как показано на рис. 25.7. Мы могли бы воспользоваться принципом ISP, чтобы разделить функции набора номера и соединения на два отдельных интерфейса. Устаревший модем будет выполнять оба интерфейса, а клиенты модемов будут их использовать. Класс `DedUsers` будет использовать только интерфейс `Modem`, а `DedicatedModem` — лишь интерфейс `Modem`. К сожалению, в связи с этим нам потребуется внести изменения во все клиенты модемов — именно то, что запретили нам заказчики.

Как бы нам ни хотелось, мы не можем разделить интерфейсы, но можно предложить такой способ, чтобы все клиенты модемов могли пользоваться классом `DedicatedModem`. Одно из возможных решений — вывести `DedicatedModem`

[2]Все модемы могут использоваться в режиме выделенной линии. Этой способности не было у устаревших модемов, т.е. раньше применялись модемы, специально предназначенные для работы с выделенными линиями. Эти модемы могли работать в коммутируемом режиме только в случае подключения отдельного устройства, называемого автокоммутатором.

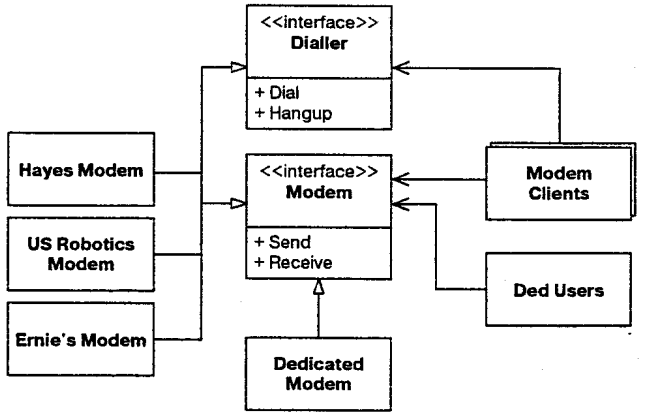


Рис. 25.7. Идеальное решение модемной проблемы

из интерфейса `Modem` и применить функции `dial` и `hangup` для выполнения следующих действий.

```
class DedicatedModem public : Modem
{
  public:
    virtual void dial(char phoneNumber[10]) {}
    virtual void hangup() {}
    virtual void send(char c)
    {...}
    virtual char receive()
    {...}
};
```

Дегенерация функций может означать то, что мы нарушаем принцип LSP. Пользователи основного класса могут ожидать, что функции `dial` и `hangup` значительно изменяют состояние модема. Дегенерация применений в `DedicatedModem` может не оправдать эти ожидания.

Допустим, что при создании клиентов модемов предполагалось, что их модемы должны находиться в состоянии ожидания до момента начала набора номера и вернуться в это состояние при вызове функции `hangup`. Другими словами, не предполагается, что модемы будут передавать какие-либо символы, не введенные ранее. Класс `DedicatedModem` приводит к обратным результатам. Он вернет символы до вызова функции `hangup` и будет продолжать возвращать их после вызова функции `hangup`. Таким образом, `Dedicated-Modem` может повредить некоторые из модемных клиентов.

Теперь читатель может справедливо полагать, что вся проблема заключается в клиентах модемов. Они разработаны недостаточно корректно, если повреждаются при непредвиденном вводе данных. Я бы согласился с этим утверждением.

Однако трудно будет убедить программистов, занимающихся поддержкой клиентов модемов, внести изменения в свои программы по причине добавления нового вида модема. Этим нарушается не только принцип "открытия-закрытия" (OCP), но и ставится под вопрос функционирование всей программы. К тому же, наши заказчики запретили вносить изменения в клиенты модемов.

Решение проблемы: клудж

Можно имитировать состояние соединения в методах `dial` и `hangup` объекта `DedicatedModem`. Можно не возвращать символы, если не вызвана функция `dial` или вызвана функция `hangup`. Прибегнув к такому решению, нам не придется изменять все клиенты модемов. *Все, что от нас требуется, — убедить* `DedUsers` *вызвать функции* `dial` *и* `hangup` (рис. 25.8).

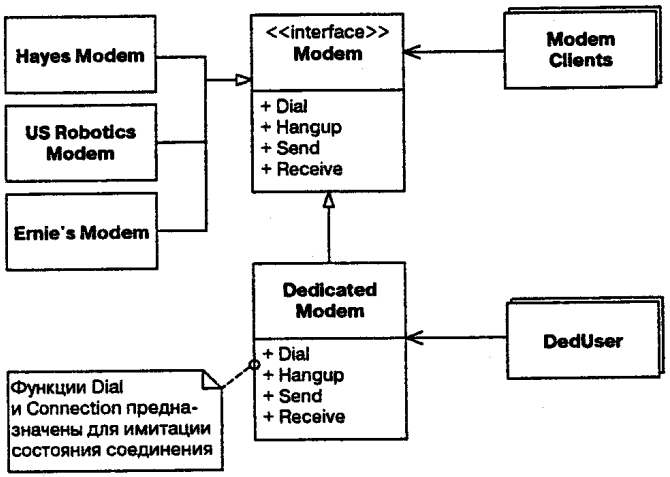


Рис. 25.8. Разрешение модемной проблемы путем включения клуджа DedicateModem для имитации состояния соединения

Представьте себе, что программисты, создающие `DedUsers`, находят такие действия довольно разрушающими. Они явно используют `DedicatedModem`. *Зачем же вызывать* `dial` *и* `hangup`? Тем не менее, они пока еще не создали свою программу, поэтому их проще убедить сделать то, чего хотим мы.

Запутанная сеть зависимостей

Спустя месяцы, когда появляются сотни `DedUsers`, наши заказчики предлагают внести новое изменение. Создается впечатление, что все эти годы нашим программистам не приходилось набирать международные телефонные номера.

Теперь мы отказались от `char[10]` в функции `dial`. Наши заказчики должны получать возможность набирать телефонные номера произвольной длины.

Понятно, что все клиенты модемов должны изменяться. Они создавались с расчетом на использование массива `char[10]` для набора телефонного номера. Заказчики разрешают внести изменения в модемные клиенты. Как известно, классы в модемной иерархии должны измениться с тем, чтобы подстроиться под новый формат телефонного номера. *К сожалению, нам теперь необходимо обратиться к разработчикам* `DedUsers` *с просьбой об изменении кода.* Можете только себе представить, как они "обрадуются"! Они не вызывали функцию `dial`, поскольку в этом нет необходимости. Вызов этой функции осуществлялся после нашего указания. А теперь им необходимо проделать дорогостоящую работу по поддержке, поскольку они выполнили наши указания.

Это пример неприятной путаницы зависимостей, с которой сталкиваются многие проекты. Клудж в одной части системы создает неприятную угрозу появления зависимости, которая фактически является причиной возникновения проблем в тех частях системы, которые должны быть полностью несвязанными.

На помощь придет шаблон Adapter

Можно было бы предотвратить эту неприятную ситуацию, если бы мы воспользовались шаблоном `Adapter` для разрешения исходной проблемы, как показано на рис. 25.9. В этом случае `DedicatedModem` не наследует из интерфейса `Modem`. Клиенты модемов используют `DedicatedModem` опосредованным образом через `DedicatedModemAdapter`. Этот адаптер использует функции `dial` и `hangup` для имитации состояния соединения. Он передает исходящие и входящие звонки `DedicatedModem`.

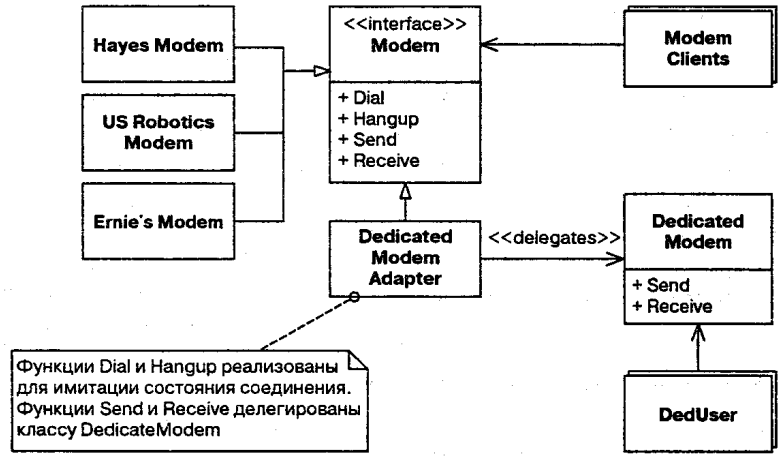


Рис. 25.9. Разрешение модемной проблемы с помощью шаблона Adapter

Обратите внимание, что благодаря такому решению устраняются все имевшиеся ранее трудности. Клиенты объекта Modem наблюдают ожидаемый тип поведения, а DedUsers не могут работать с функциями dial или hangup. При изменении требований к набору и формату телефонного номера DedUsers не изменяется. Таким образом, благодаря использованию адаптера устранились проблемы с нарушением принципов LSP и OCP.

Обратите внимание, что клудж пока еще существует. Адаптер по прежнему имитирует состояние соединения. Вам это покажется ужасным, и я, конечно же, соглашусь с вами. Тем не менее, обратите внимание, что все зависимости указывают направление, противоположное от адаптера. Клудж изолирован от системы и "скрыт" в адаптере, о котором едва кто-то знает. Устанавливается только жесткая зависимость до тех пор, пока адаптер не реализует здесь некоторую фабрику[3].

Шаблон Bridge

Сформулированную проблему можно рассмотреть и с другой точки зрения. Необходимость в выделенном модеме стимулирует добавление новой степени свободы в типовую иерархию Modem. Изначально тип Modem был задуман как интерфейс для набора различных аппаратных средств.

Известно, что HayesModem, USRModem и ErniesModem являлись производными классами от основного класса Modem. Оказывается, что есть еще один способ сокращения иерархии Modem. Можно было бы вывести DialModem и DedicatedModem из объекта Modem.

Слияние этих двух независимых иерархий можно выполнить по схеме, представленной на рис. 25.10. Каждый из уровней иерархии типов включает поведение dialup (коммутируемая линия) или dedicated (выделенная линия) в контролируемое им аппаратное обеспечение. Объект DedicatedHayesModem контролирует модем Hayes, применяющийся в выделенном контексте.

Эта структура не является идеальной. Всякий раз, добавляя что-то новое из аппаратного обеспечения, мы должны создать *два* новых класса — один для выделенного случая и один для коммутируемой линии. Каждый раз, добавляя новый тип соединения, следует создавать *три* новых класса — по одному для каждого отдельного аппаратного компонента. Если две степени свободы обеспечивают высшую степень изменчивости, мы можем развернуть большое количество производных классов (до определенного предела, конечно).

Шаблон Bridge часто помогает в таких ситуациях, когда типовая иерархия имеет более одной степени свободы. Вместо слияния иерархий можно разделить их, а затем соединить посредством моста.

[3] См. гл 21.

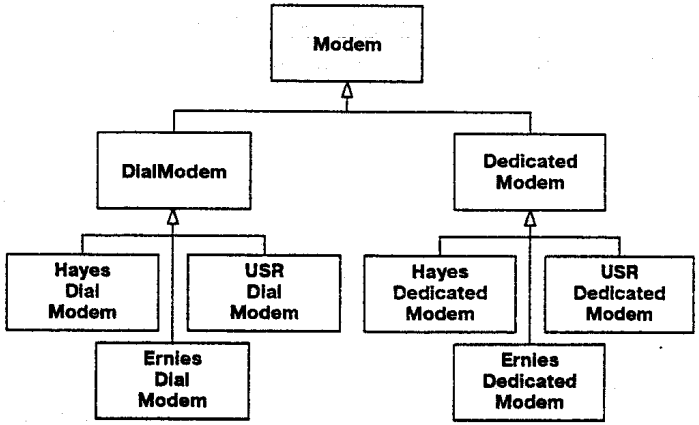


Рис. 25.10. Разрешение модемной проблемы путем слияния иерархии типов

На рис. 25.11 представлена рассматриваемая структура. Иерархия модема разделяется на две подчиненные иерархии. Одна представляет метод соединения, а другая — аппаратное обеспечение.

Пользователи модемов продолжают использовать интерфейс Modem. Объект ModemConnectionController использует интерфейс Modem. Производные классы для ModemConnectionController контролируют механизм соединения.

Объект DialModemController просто передает методы dial и hangup объектам основного класса dialImp и hangImp. Эти методы затем передаются классу ModemImplementation, где они разворачиваются в соответствующем контроллере аппаратного обеспечения. DedModemController использует методы dial и hangup для имитации состояния соединения. Он передает методы send и receive объектам sendImp и receiveImp, которые, как и раньше, передаются иерархии ModemImplementation.

Обратите внимание, что четыре функции imp в основном классе ModemConnectionController защищены. Это происходит в силу того, что они должны строго использоваться производными ModemConnectionController. Никто больше не должен их вызывать.

Описанная структура сложная, но интересная. Ее можно создать, не затрагивая интересы пользователей модемов, и, кроме того, она позволяет полностью отделить политики соединения от применения аппаратных средств. Каждый производный класс объекта ModemConnectionController представляет новую политику соединения. Эта политика может использовать методы sendimp, receiveimp, dialimp и hangimp. Можно создать новые функции imp, не затрагивая при этом интересы пользователей. Принцип ISP может применяться для

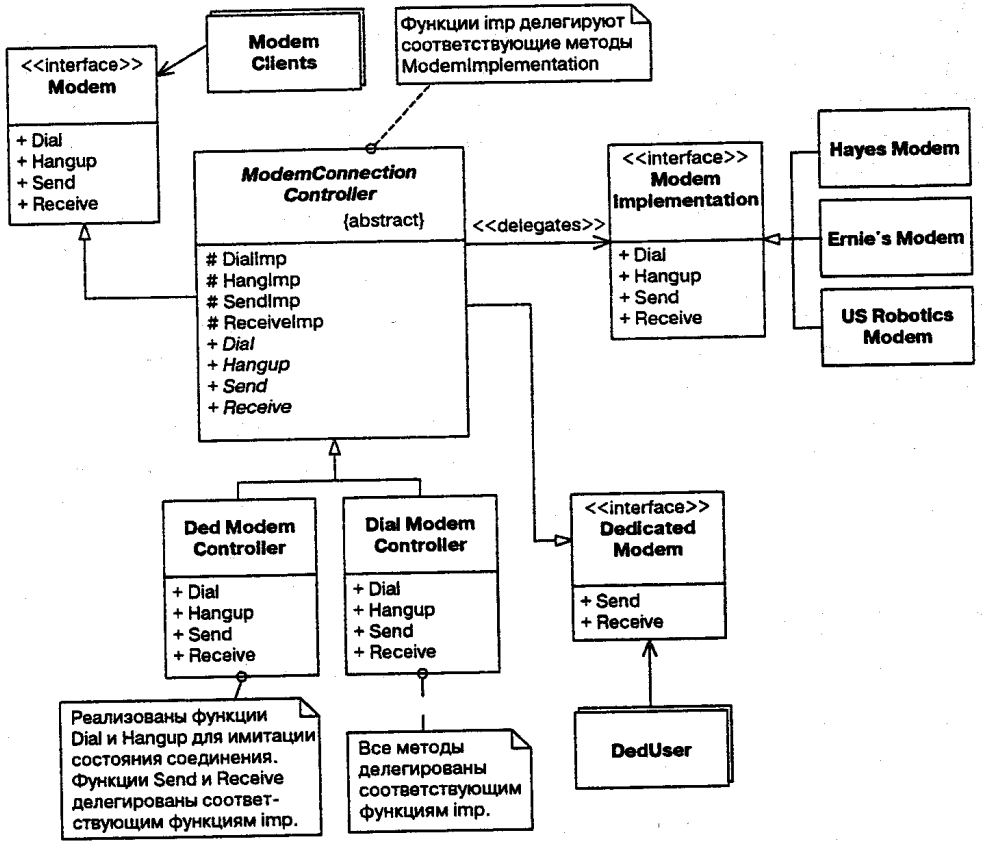


Рис. 25.11. Решение модемной проблемы с помощью шаблона Bridge

добавления новых интерфейсов в классы контроллеров соединений. Это приводит к образованию пути перемещения, по которому клиенты модема могут переходить на уровень API. Этот уровень находится выше того уровня, на котором находятся объекты `dial` и `hangup`.

Резюме

Может показаться, что настоящая проблема со сценарием `Modem` заключается в некорректной разработке исходного проекта. Разработчики должны были знать, что понятия соединения и коммуникации весьма различны. Проведя небольшой анализ, они обнаружили бы эту проблему и устранили ее. Поэтому вся суть проблемы заключается в недостаточном анализе.

Все это нонсенс! Нет такого понятия, как *достаточный* анализ. Не важно, сколько времени проходит в попытках создать идеальную структуру. Всегда об-

От этого никуда не денешься. Идеальных структур не существует. Есть только структуры, которые пытаются уравновесить текущие затраты и получаемые доходы. С течением времени эти структуры должны изменяться по мере изменений требований системы. Мастерство управления этим изменением состоит в поддержании системы как можно более простой и гибкой.

Решение в виде шаблона `Adapter` является простым и непосредственным. Благодаря этому решению все зависимости продолжают указывать правильное направление. Решение в виде шаблона `bridge` немного сложнее. Я не буду предлагать вам эту методику до тех пор, пока вы в достаточной степени не убедитесь в необходимости полностью разграничить политики соединения и коммуникации и добавить новые политики соединения.

Вытекающий из этого вывод, как всегда, состоит в том, что применение шаблона влечет за собой определенные затраты, но в то же время обеспечивает определенные преимущества. Следует использовать такие шаблоны, которые наилучшим образом соответствуют рассматриваемой проблеме.

Литература

1. Gamma и др. *Design Patterns*, Reading, MA: Addison-Wesley, 1995.