Глава 5

ПРОЕКТИРОВАНИЕ КЛАССОВ

Шаблоны проектирования GRASP

При создании классов, распределении обязанностей и способов взаимодействия объектов перед программистом возникает серьезная проблема моделирования взаимодействия классов и объектов. Представляются широкие возможности выбора. Неоптимальный выбор может сделать системы и их отдельные компоненты непригодными для поддержки, восприятия, повторного использования и расширения. Систематизация приемов программирования и принципов организации классов получила название шаблона.

Основные принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах GRASP (General Responsibility Assignment Software Patterns).

При этом были сформулированы основные принципы и общие стандартные решения, придерживаясь которых можно создавать хорошо структурированный и понятный код.

Шаблон Expert

При проектировании классов на первом этапе необходимо определить общий принцип распределения обязанностей между классами проекта, а именно: в первую очередь определить кандидатов в информационные эксперты – классы, обладающие информацией, требуемой для выполнения своей функциональности.

Простые эксперты определять достаточно просто, но часто возникает необходимость дополнять класс атрибутами, и в этом случае необходимо сделать правильный выбор. Например, в подсистеме прохождения назначенного теста некоторому классу необходимо знать число вопросов, на которые получен ответ на текущий момент времени в процессе тестирования. Какой класс должен отвечать за знание количества вопросов, на которые дан ответ на текущий момент времени при прохождении теста, если определены следующие классы?

```
/*пример # 1 : шаблон Expert : LineRequestQuest.java :Test.java : Quest.java */
      public class Test {//информационный эксперт
            private int idTest;
            private int numberQuest;
            private String testName;
            // реализация конструкторов и методов
      }
      public class LineRequestQuest {
            private int questID;
            // реализация конструкторов и методов
      }
```

```
public class Quest {//информационный эксперт
        private int idQuest;
        private int testID;
        // реализация конструкторов и методов
}
```

Необходимо узнать количество вопросов из теста, на которые дан ответ, то есть число созданных объектов класса **LineRequestQuest**. Такой информацией обладает лишь экземпляр объекта **Test**, так как этот класс ответствен за знание общего количества вопросов в тесте. Следовательно, с точки зрения шаблона Expert объект **Test** подходит для выполнения этой обязанности, т.е. является информационным экспертом.

```
/*пример # 2 : шаблон Expert : Test.java */
class Test {
        private int idTest;
        private int numberQuest;
        private String testName;
        private int currentNumberQuest;

        public int getCurrentNumberQuest() {
                // реализация
        }
        // реализация конструкторов и методов
}
```

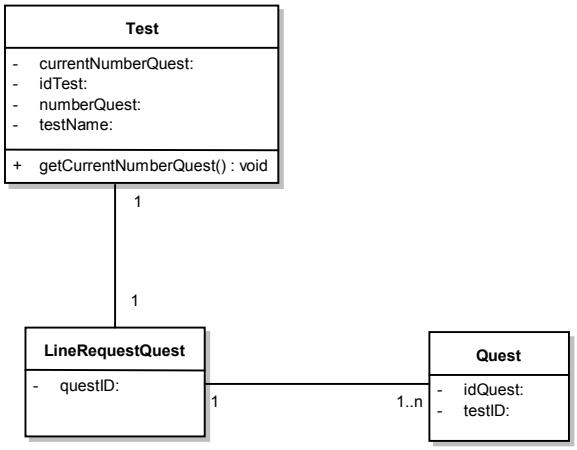

Рис. 5.1. Применение шаблона Expert

Преимущества следования шаблону Expert:

- сохранение инкапсуляции информации при назначении ответственности классам, которые уже обладают необходимой информацией для обеспечения своей функциональности;
- уклонение от новых зависимостей способствует обеспечению низкой степени связанности между классами (Low Coupling);
- добавление соответствующего метода способствует высокому зацеплению (Highly Cohesive) классов, если класс уже обладает информацией для обеспечения необходимой функциональности.

Однако назначение чрезмерно большого числа ответственностей классу при использовании шаблона Expert может привезти к получению слишком сложных классов, которые перестанут удовлетворять шаблонам Low Coupling и High Cohesion.

Шаблон Creator

Существует большая вероятность того, что класс проще, если он будет большую часть своего жизненного цикла ссылаться на создаваемые объекты.

После определения информационных экспертов следует определить классы, ответственные за создание нового экземпляра некоторого класса. Следует назначить классу **B** обязанность создавать экземпляры класса **A**, если выполняется одно из следующих условий:

- класс **B** агрегирует (aggregate) объекты **A**;
- класс **B** содержит (contains) объекты **A**;
- класс **B** записывает или активно использует (records or closely uses) экземпляры объектов **A**;
- классы **B** и **A** относятся к одному и тому же типу, и их экземпляры составляют, агрегируют, содержат или напрямую используют другие экземпляры того же класса;
- класс **B** содержит или получает данные инициализации (has the initializing data), которые будут передаваться объектам **A** при его создании.

Если выполняется одно из указанных условий, то класс **B** – создатель (creator) объектов **A**.

Инициализация объектов – стандартный процесс. Грамотное распределение обязанностей при проектировании позволяет создать слабо связанные независимые простые классы и компоненты.

В соответствии с шаблоном необходимо найти класс, который должен отвечать за создание нового экземпляра объекта **Quest** (агрегирующий экземпляры объектов **Quest**).

Поскольку объект **LineRequestQuest** использует объект **Quest**, согласно шаблону Creator он является кандидатом для выполнения обязанности, связанной с созданием экземпляров объектов **Quest**. В этом случае обязанности могут быть распределены следующим образом:

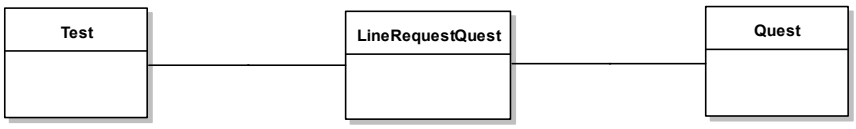

Рис. 5.2. Пример реализации шаблона Creator

```
/* пример # 3 : шаблон Creator: Qest.java: LineRequestQuest.java :Test.java */
        public class Test {
           private int idTest;
           private int numberQuest;
           private String testName;
           private int currentNumberQuest;
           // реализация конструкторов и методов
           }
```

```
public class LineRequestQuest {
   private int questID;

   public void answerQuest() {
   // реализация
   Vector q = new Vector();
   q.add(makeRequest(параметры));
   //
   }
   public Quest makeRequest(параметры) {
         // реализация
         return new Quest(параметры);
   }
}
public class Quest{
   private int idQuest;
   private int testID;

   public Quest() {}
   // реализация конструкторов и методов
}
```

Шаблон Creator способствует низкой зависимости между классами (Low Coupling), так как экземпляры класса, которым необходимо содержать ссылку на некоторые объекты, должны создавать эти объекты. При создании некоторого объекта самостоятельно класс тем самым перестает быть зависимым от класса, отвечающего за создание объектов для него. Распределение обязанностей выполняется в процессе создания диаграммы взаимодействия классов.

Шаблон Low Coupling

Степень связанности классов определяет, насколько класс связан с другими классами и какой информацией о других классах он обладает. При проектировании отношений между классами следует распределить обязанности таким образом, чтобы степень связанности оставалась низкой.

Наличие классов с высокой степенью связанности нежелательно, так как:

- изменения в связанных классах приводят к локальным изменениям в данном классе;
- затрудняется понимание каждого класса в отдельности;
- усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

Пусть требуется создать экземпляр класса **Quest** и связать его с объектом **Test**. В предметной области регистрация объекта **Test** выполняется объектом **Course**.

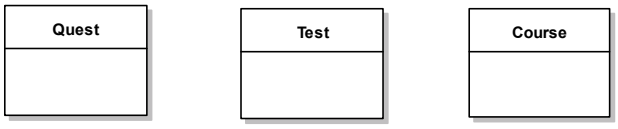

Рис. 5.3. Классы, которые необходимо связать

Далее экземпляр объекта **Course** должен передать сообщение **makeQuest()** объекту **Test**. Это значит, что в текущем тесте были получены идентификаторы всех вопросов, составляющих тест и становится возможным создание объектов типа **Quest** и наполнение собственно теста.

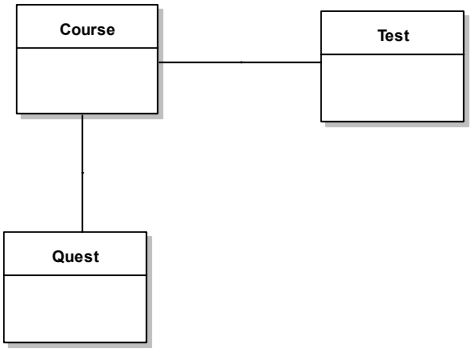

Рис. 5.4. Пример плохой реализации шаблона Low Coupling

```
/* пример # 4 : шаблон Low Coupling : Qest.java : Test.java : Course.java */
        public class Course {
           private int id;
           private String name;

           public void makeTest(){
                  Test test = new Test(параметры);
                  // реализация
           while(условие){
                  Quest quest = new Quest(параметры);
                  // реализация
                  test.addTest(quest);
                  }
                  // реализация
           }
        }
        public class Test {
           // поля
           public void addTest(Quest quest){
                  // реализация
           }
        }
        public class Quest {
           // поля и методы
        }
```

При таком распределении обязанностей предполагается, что класс **Course** связывается с классом **Quest**.

Второй вариант распределения обязанностей с уклонением класса **Course** от создания объектов вопросов представлен на рисунке 5.5.

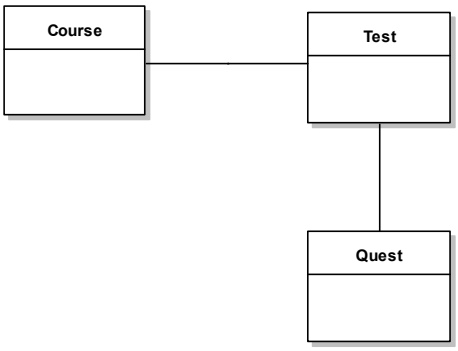

Рис. 5.5. Пример правильной реализации шаблона Low Coupling

```
/* пример # 5 : шаблон Low Coupling: Qest.java: Test.java: Course.java */
public class Course {
   private int id;
   private String name;

   public void makeTest() {
         Test test = new Test(параметры);
         // реализация
         test.addTest();
         // реализация
   }
}
public class Test {
   // поля
   public void addTest() {
         // реализация
     while(условие) {
       Quest quest = new Quest(параметры);
         // реализация
         }
   }
}
public class Quest {
   // поля и методы
}
```

Какой из методов проектирования, основанный на распределении обязанностей, обеспечивает низкую степень связанности?

В обоих случаях предполагается, что объекту **Test** должно быть известно о существовании объекта **Quest**.

При использовании первого способа, когда объект **Quest** создается объектом **Course**, между этими двумя объектами добавляется новая связь, тогда как второй способ степень связывания объектов не увеличивает. Более предпочтителен второй способ, так как он обеспечивает низкую связываемость.

В ООП имеются некоторые стандартные способы связывания объектов **A** и **B**:

- объект **A** содержит атрибут, который ссылается на экземпляр объекта **B**;

- объект **A** содержит метод, который ссылается на экземпляр объекта **B**, что подразумевает использование **B** в качестве типа параметра, локальной переменной или возвращаемого значения;
- класс объекта **A** является подклассом объекта **B**;
- **B** является интерфейсом, а класс объекта **A** реализует этот интерфейс.

Шаблон Low Coupling нельзя рассматривать изолированно от других шаблонов (Expert, Creator, High Cohesion). Не существует *абсолютной меры* для определения слишком высокой степени связывания.

Преимущества следования шаблону Low Coupling:

- изменение компонентов класса мало сказывается на других классах;
- принципы работы и функции компонентов можно понять, не изучая другие классы.

Шаблон High Cohesion

С помощью этого шаблона можно обеспечить возможность управления сложностью, распределив обязанности, поддерживая высокую степень зацепления.

Зацепление – мера сфокусированности класса. При высоком зацеплении обязанности класса тесно связаны между собой, и класс не выполняет работ непомерных объёмов. Класс с низкой степенью зацепления выполняет много разнородных действий или не связанных между собой обязанностей.

Возникают проблемы, связанные с тем, что класс:

- труден в понимании, так как необходимо уделять внимание несвязным (неродственным) идеям;
- сложен в поддержке и повторном использовании из-за того, что он должен быть использован вместе с зависимыми классами;
- ненадежен, постоянно подвержен изменениям.

Классы со слабым зацеплением выполняют обязанности, которые можно легко распределить между другими классами.

Пусть необходимо создать экземпляр класса **Quest** и связать его с заданным тестом. Какой класс должен выполнять эту обязанность? В предметной области сведения о вопросах на текущий момент времени при прохождении теста записываются в объекте **Course**, согласно шаблону для создания экземпляра объекта **Quest** можно использовать объект **Course**. Тогда экземпляр объекта **Course** сможет отправить сообщение **makeTest()** объекту **Test**. За прохождение теста отвечает объект **Course**, т.е. объект **Course** частично несет ответственность за выполнение операции **makeTest()**. Однако если и далее возлагать на класс **Course** обязанности по выполнению все новых функций, связанных с другими системными операциями, то этот класс будет слишком перегружен и будет обладать низкой степенью зацепления.

Этот шаблон необходимо применять при оценке эффективности каждого проектного решения.

Виды зацепления:

1. *Очень слабое зацепление*. Единоличное выполнение множества разнородных операций.

```
/*пример # 6 : очень слабое зацепление : Initializer.java */
        public class Initializer {
```

```
        public void createTCPServer(String port) {
                // реализация
        }
        public int connectDataBase(URL url) {
                // реализация
        }
        public void createXMLDocument(String name) {
                // реализация
        }
    }
```

2. *Слабое зацепление*. Единоличное выполнение сложной задачи из одной функциональной области.

```
/*пример # 7 : слабое зацепление : NetLogicCreator.java */
    public class NetLogicCreator {
        public void createTCPServer() {
                // реализация
        }
        public void createTCPClient() {
                // реализация
        }
        public void createUDPServer() {
                // реализация
        }
        public void createUDPClient() {
                // реализация
        }
    }
```

3. *Среднее зацепление*. Несложные обязанности в нескольких различных областях, логически связанных с концепцией этого класса, но не связанных между собой.

```
/*пример # 8 : среднее зацепление : TCPServer.java */
  public class TCPServer {
        public void createTCPServer() {
                // реализация
        }
        public void receiveData() {
                // реализация
        }
        public void sendData() {
                // реализация
        }
        public void compression() {
                // реализация
        }
        public void decompression() {
                // реализация
        }
    }
```

4. *Высокое зацепление.* Среднее количество обязанностей из одной функциональной области при взаимодействии с другими классами.

```
/*пример # 9 : высокое зацепление : TCPServerCreator.java : DataTransmission.java
: CodingData.java */
        public class TCPServerCreator {
           public void createTCPServer() {
                 // реализация
           }
        }
        public class DataTransmission {
           public void receiveData() {
                 // реализация
           }
           public void sendData() {
                 // реализация
           }
        }
        public class CodingData {
           public void compression() {
                 // реализация
           }
           public void decompression() {
                 // реализация
           }
        }
```

Если обнаруживается, что используется слишком негибкий дизайн, который сложен в поддержке, следует обратить внимание на классы, которые не обладают свойством зацепления или зависят от других классов. Эти классы легки в узнавании, поскольку они сильно взаимосвязаны с другими классами или содержат множество неродственных методов. Как правило, классы, которые не обладают сильной зависимостью с другими классами, обладают свойством зацепления и наоборот. При наличии таких классов необходимо реорганизовать их структуру таким образом, чтобы они по возможности не являлись зависимыми и обладали свойством зацепления.

Шаблон Controller

Одной из базовых задач при проектировании информационных систем является определение класса, отвечающего за обработку системных событий. При необходимости посылки внешнего события прямо объекту приложения, которое обрабатывает это событие, как минимум один из объектов должен содержать ссылку на другой объект, что может послужить причиной очень негибкого дизайна, если обработчик событий зависит от типа источника событий или источник событий зависит от типа обработчика событий.

В простейшем случае зависимость между внешним источником событий и внутренним обработчиком событий заключается исключительно в передаче событий. Довольно просто обеспечить необходимую степень независимости между источником событий и обработчиком событий, используя интерфейсы. Интер-

фейсов может оказаться недостаточно для обеспечения поведенческой независимости между источником и обработчиком событий, когда отношения между этими источником и обработчиком достаточно сложны.

Можно избежать зависимости между внешним источником событий и внутренним обработчиком событий путем введения между ними дополнительного объекта, который будет работать в качестве посредника при передаче событий. Этот объект должен быть способен справляться с любыми другими сложными аспектами взаимоотношений между объектами.

Согласно шаблону Controller, производится делегирование обязанностей по обработке системных сообщений классу, если он:

- представляет всю организацию или всю систему в целом (внешний контроллер);
- представляет активный объект из реального мира, который может участвовать в решении задачи (контроллер роли);
- представляет искусственный обработчик всех системных событий прецедента и называется `Прецедент`**`Handler`** (контроллер прецедента) .

Для всех системных событий в рамках одного прецедента используется один и тот же контролер.

Controller – это класс, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Использование объекта-контроллера обеспечивает независимость между внешними источниками событий и внутренними обработчиками событий, их типом и поведением. Выбор определяется зацеплением и связыванием.

Раздутый контроллер (волшебный сервлет) выполняет слишком много обязанностей. Признаки:

- в системе имеется единственный класс контроллера, получающий все системные сообщения, которых поступает слишком много (внешний или ролевой контроллер);
- контроллер имеет много полей (информации) и методов (ассоциаций), которые необходимо распределить между другими классами.

Шаблоны проектирования GoF

Шаблоны проектирования GoF – это многократно используемые решения широко распространенных проблем, возникающих при разработке программного обеспечения. Многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме.

«Любой шаблон описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново». (Кристофер Александер).

В общем случае шаблон состоит из четырех основных элементов:

1. *Имя.* Точное имя предоставляет возможно сразу понять проблему и определить решение. Уровень абстракции при проектировании повышается.
2. *Задача.* Область применения в рамках решения конкретной проблемы.

3. *Решение.* Абстрактное описание элементов дизайна задачи проектирования и способа ее решения с помощью обобщенного набора классов.
4. *Результаты.*

Шаблоны классифицируются по разным критериям, наиболее распространенным из которых является назначение шаблона. Вследствие этого выделяются порождающие шаблоны, структурные шаблоны и шаблоны поведения.

Порождающие шаблоны

Порождающие шаблоны предназначаются для организации процесса создания объектов.

К порождающим шаблонам относятся:

Abstract Factory (**Абстрактная Фабрика**) – предоставляет интерфейс для создания связанных между собой объектов семейств классов без указания их конкретных реализаций;

Factory (**Фабрика**) – создает объект из иерархического семейства классов на основе передаваемых данных (частный случай Abstract Factory);

Builder (**Строитель**) – создает объект класса различными способами;

Singleton (**Одиночка**) – гарантирует существование только одного экземпляра класса;

Prototype (**Прототип**) – применяется при создании сложных объектов. На основе прототипа объекты сохраняются и воссоздаются, н-р путем копирования;

Factory Method (**Фабричный Метод**) – определяет интерфейс для создания объектов, при этом объект класса создается с помощью методов подклассов.

Шаблон Factory

Необходимо определить механизм создания объектов по заданному признаку для классов, находящихся в иерархической структуре. Основной класс шаблона представляет собой класс, который имеет методы для создания одного объекта из нескольких возможных на основе передаваемых данных.

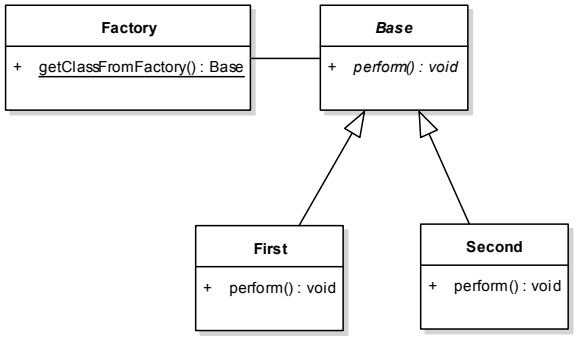

Рис. 5.6. Пример реализации шаблона Factory

Решением проблемы может быть создание класса **`ClassFactory`** с одним методом **`getClassFromFactory(String id)`**, возвращаемым значением которого будет ссылка на класс-вершину **`Base`** иерархии создаваемых классов. В качестве параметра метода передается некоторое значение, в соответствии с

которым будет осуществляться инициализация объекта одного из подклассов класса **Base**.

Программная реализация может быть представлена в общем виде следующим образом.

```
/*пример # 10 : создание объектов с помощью шаблона Factory : Base.java :
First.java : Second.java : ClassFactory.java : Main.java */
package chapt05.factory;
public abstract class Base {
      public abstract void perform();
}
package chapt05.factory;
public class First extends Base {
      public void perform() {
            System.out.println("First");
    }
}
package chapt05.factory;
public class Second extends Base {
    public void perform() {
            System.out.println("Second");
    }
}
package chapt05.factory;
public class ClassFactory {
  private enum  Signs {FIRST, SECOND}
    public static Base getClassFromFactory(String id) {
      Signs sign = Signs.valueOf(id.toUpperCase());
      switch(sign){
       case FIRST  : return new First();
       case SECOND : return new Second();
       default : throw new EnumConstantNotPresentException(
             Signs.class, sign.name());
      }
   }
}
package chapt05.factory;
public class Main {
    public static void main(String args[]) {
        Base ob1 =
            ClassFactory.getClassFromFactory("first");
        Base ob2 =
            ClassFactory.getClassFromFactory("second");
        ob1.perform();
        ob2.perform();
    }
}
```

Один из примеров применения данного шаблона уже был рассмотрен в примере # 5 предыдущей главы.

Шаблон AbstractFactory

Необходимо создавать объекты классов, не имеющих иерархической связи, но логически связанных между собой. Абстрактный класс-фабрика определяет общий интерфейс таких фабрик. Его подклассы обладают конкретной реализацией методов по созданию разных объектов.

Предложенное решение изолирует конкретные классы. Так как абстрактная фабрика реализует процесс создания классов-фабрик и саму процедуру инициализации объектов, то она изолирует приложение от деталей реализации классов.

Одна из возможных реализаций шаблона предложена в следующем примере. Классы фабрики создаются по тому же принципу, по которому в предыдущем шаблоне создавались объекты.

```
/*пример # 11 : создание классов-фабрик по заданному признаку :
AbstractFactory.java */
package chapt05.abstractfactory;
public class AbstractFactory {
      enum Color { BLACK, WHITE };
      public static BaseFactory getFactory(String data) {
            Color my = Color.valueOf(data.toUpperCase());
            switch (my){
                  case BLACK : return new BlackFactory();
                  case WHITE : return new WhiteFactory();
                  default : throw new
EnumConstantNotPresentException(Signs.class, sign.name());
            }
      }
}
```

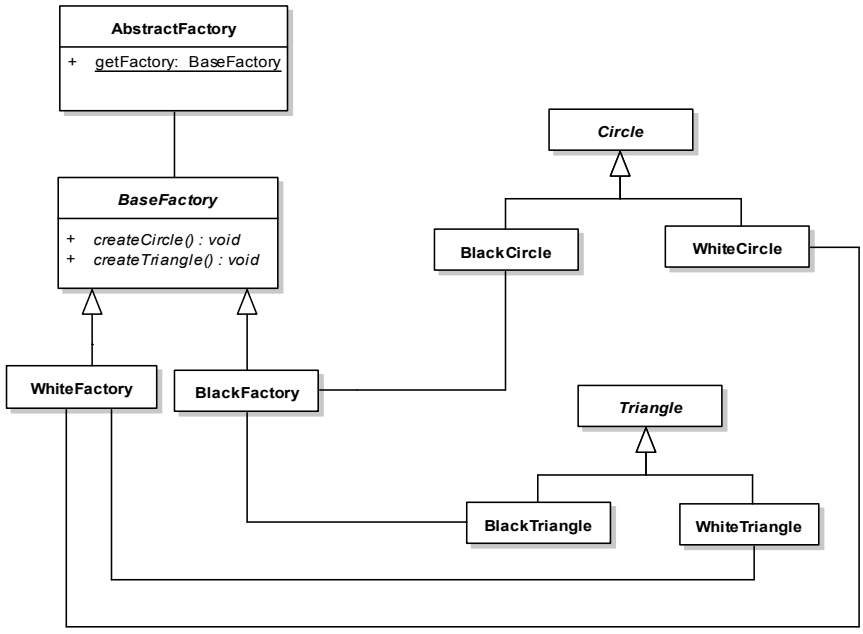

Рис. 5.7. Пример реализации шаблона AbstractFactory

Производители объектов реализуют методы по созданию не связанных иерархическими зависимостями объектов. Класс **BaseFactory** – абстрактная фабрика, а классы **BlackFactory** и **WhiteFactory** конкретные производители объектов, наследуемые от нее. Конкретные фабрики могут создавать черные или белые объекты-продукты.

```
/*пример # 12 : классы-фабрики по созданию несвязанных объектов :
BaseFactory.java : BlackFactory.java : WhiteFactory.java */
package chapt05.abstractfactory;
public abstract class BaseFactory {
      public abstract Circle createCircle(double radius);
      public abstract Triangle createTriangle(double a,
                                                double b);
}
package chapt05.abstractfactory;
public class BlackFactory extends BaseFactory {
      public Circle createCircle(double radius) {
            return new BlackCircle(radius);
      }
      public Triangle createTriangle(double a, double b){
            return new BlackTriangle(a,b);
      }
}
package chapt05.abstractfactory;
public class WhiteFactory extends BaseFactory {
      public Circle createCircle(double radius) {
            return new WhiteCircle(radius);
      }
      public Triangle createTriangle(double a, double b){
            return new WhiteTriangle(a, b);
      }
}
```

Рассматриваются два вида классов-продуктов: **Circle**, **Triangle**. Каждый из них может быть представлен в одном из двух цветов: белом или черном.

```
/*пример # 13 : классы-продукты : Circle.java : Triangle.java */
package chapt05.abstractfactory;
public abstract class Circle {
      protected double radius;
      protected String color;
      public abstract void square();
}
package chapt05.abstractfactory;
public class BlackCircle extends Circle {
      public BlackCircle(double radius){
            this.radius = radius;
            color = "Black";
      }
```

```
      public void square(){
            double s = Math.PI * Math.pow(radius, 2);
            System.out.println(color + " Circle"
                              + "  Square =  " + s);
      }
}
package chapt05.abstractfactory;
public class WhiteCircle extends Circle{
      public WhiteCircle(double radius){
            this.radius = radius;
            color = "White";
      }
      public void square(){
            double s = Math.PI * Math.pow(radius, 2);
            System.out.println(color + " Circle  "
                                    + "Square =  " + s);
      }
}
package chapt05.abstractfactory;
public abstract class Triangle {
      protected double a, b;
      protected String color;
      public abstract void square();
}
package chapt05.abstractfactory;
public class BlackTriangle extends Triangle {
      public BlackTriangle (double a, double b){
            this.a = a;
            this.b = b;
            color = "Black";
      }
      public void square(){
            double s = a * b / 2;
            System.out.println(color + " Triangle"
                                    + "  Square =  " + s);
      }
}
package chapt05.abstractfactory;
public class WhiteTriangle extends Triangle {
      public WhiteTriangle (double a,double b) {
            this.a = a;
            this.b = b;
            color = "White";
      }
      public void square(){
            double s = 0.5 * a * b;
            System.out.println(color + " Triangle"
                                    + "  Square =  " + s);
      }
}
```

Ниже будут созданы объекты всех классов и всех цветов.

```
/*пример # 14 : демонстрация работы шаблона AbstractFactory : Main.java */
package chapt05.abstractfactory;
public class Main {
      public static void main(String[] args) {
            BaseFactory factory1 =
                  AbstractFactory.getFactory("black");
            BaseFactory factory2 =
                  AbstractFactory.getFactory("white");
            Circle ob1 = factory1.createCircle(1.232);
            Circle ob2 = factory2.createCircle(1);
            Triangle ob3 = factory1.createTriangle(12,5);
            Triangle ob4 = factory2.createTriangle(1,7);

            ob1.square();
            ob2.square();
            ob3.square();
            ob4.square();
            }
}
```

Шаблон Builder

Необходимо задать конструирование сложного объекта, определяя для него только тип и содержимое. Детали построения объекта остаются скрытыми.

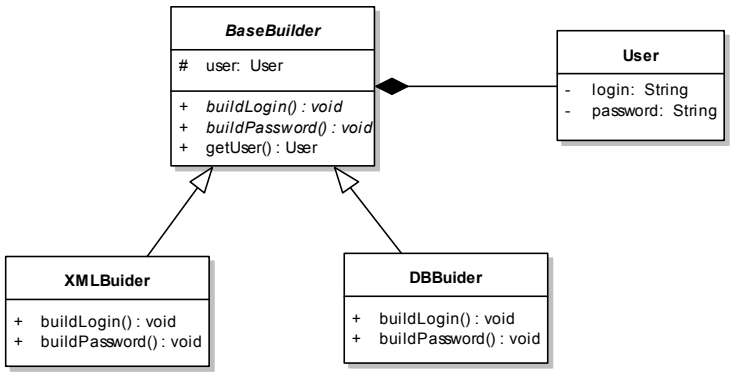

Рис. 5.8. Пример реализации шаблона Builder

Класс **BaseBuilder** определяет абстрактный интерфейс для создания частей объекта сложного класса **User**. Классы **XMLBuilder** и **DBBuilder** конструируют и собирают вместе части объекта класса **User**, а также представляет внешний интерфейс для доступа к нему. В результате объекты-строители могут работать с разными источниками, определяющими содержимое, не требуя при этом никаких изменений. При использовании этого шаблона появляется возможность контролировать пошагово весь процесс создания объекта-продукта.

Простая реализация шаблона Builder приведена ниже.

```
/*пример # 15 : «сложный» для построения объект : User.java */
package chapt05.builder;
public class User {
    private String login = "Guest";
    private String password = "Kc";

    public String getLogin() {
        return login;
    }
    public void setLogin(String login) {
        this.login = login;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Класс **BaseBuilder** – абстрактный класс-строитель, объявляющий в качестве поля ссылку на создаваемый объект и абстрактные методы его построения. Классы **XMLBuilder** и **DBBuilder** – наследуемые от него классы, реализующие специальные способы создания объекта. Таким образом, используя один класс **User** можно создать или администратора или модератора.

```
/*пример # 16 : разные способы построения объекта : BaseBuilder.java :
XMLBuilder.java: DBBuilder.java */
package chapt05.builder;
public abstract class BaseBuilder {
    protected User user = new User();

    public User getUser() {
        return user;
    }
    public abstract void buildLogin();
    public abstract void buildPassword();
}
package chapt05.builder;
public class XMLBuilder extends BaseBuilder {

    public void buildLogin() {
      //реализация
        user.setLogin("Admin");
    }
    public void buildPassword() {
      //реализация
        user.setPassword("Qu");
    }
}
```

```
package chapt05.builder;
public class DBBuilder extends BaseBuilder {

    public void buildLogin() {
      //реализация
        user.setLogin("Moderator");
    }
    public void buildPassword() {
      //реализация
        user.setPassword("Ku");
    }
}
```

Процесс создания объектов с использованием одного принципа реализован ниже.

```
/*пример # 17 : тестирование процесса создания объекта : Main.java */
package chapt05.builder;
public class Main {
    private static User buildUser(BaseBuilder builder) {
        builder.buildLogin();
        builder.buildPassword();
        return builder.getUser();
    }
    public static void main(String args[]) {
        User e1 = buildUser(new XMLBuilder());
        User e2 = buildUser(new DBBuilder());

        System.out.println(e1.getLogin());
        System.out.println(e1.getPassword());
        System.out.println(e2.getLogin());
        System.out.println(e2.getPassword());
    }
}
```

Шаблон Singleton

Необходимо создать объект класса таким образом, чтобы гарантировать невозможность инициализации другого объекта того же класса. Обычно сам класс контролирует наличие единственного экземпляра и он же предоставляет при необходимости к нему доступ.

```
/*пример # 18 : реализация шаблона «Одиночка» : Singleton.java */
package chapt05.singleton;
public class Singleton {

    private static Singleton instance = null;
    private SingletonTrust() {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            System.out.println("Creating Singleton");
            instance = new Singleton();
        }
```

```
        return instance;
    }
}
```

Класс объявляет метод **getInstance()**, который позволяет клиентам получать контролируемый доступ к единственному экземпляру. Этот шаблон позволяет уточнять методы через подклассы, а также разрешить появление более чем одного экземпляра.

Структурные шаблоны

Структурные шаблоны отвечают за композицию объектов и классов.

К структурным шаблонам относятся:

Proxy (**Заместитель**) – подменяет сложный объект более простым и осуществляет контроль доступа к нему;

Composite (**Компоновщик**) – группирует объекты в иерархические структуры и позволяет работать с единичным объектом так же, как с группой объектов;

Adapter (**Адаптер**) – применяется при необходимости использовать вместе несвязанные классы. Поведение адаптируемого класса при этом изменяется на необходимое;

Bridge (**Мост**) – разделяет представление класса и его реализацию, позволяя независимо изменять то и другое;

Decorator (**Декоратор**) – представляет способ изменения поведения объекта без создания подклассов. Позволяет использовать поведение одного объекта в другом;

Facade (**Фасад**) – создает класс с общим интерфейсом высокого уровня к некоторому числу интерфейсов в подсистеме.

Шаблон Bridge

Необходимо отделить абстракцию (Abstraction) от ее реализации (Implementor) так, чтобы и то и другое можно было изменять независимо. Шаблон Bridge используется в тех случаях, когда существует иерархия абстракций и соответствующая иерархия реализаций. Причем не обязательно точное соответствие между абстракциями и реализациями. Обычно абстракция определяет операции более высокого уровня, чем реализация.

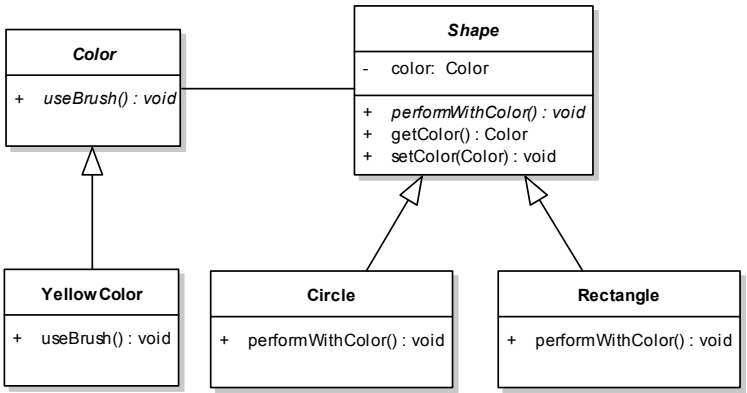

Рис. 5.9. Пример реализации шаблона Bridge

```
/*пример # 18 : Implementor и его подкласс : Color.java: YellowColor.java */
package chapt05.implementor;
public abstract class Color { //implementor
    public abstract void useBrush();
}
package chapt05.implementor;
public class YellowColor extends Color {
    public void useBrush() {
        System.out.println("BrushColor - Yellow");
    }
}
```

Класс **Color** – абстрактный, реализующий Implementor. Класс **YellowColor** – уточняющий подкласс класса **Color**.

```
/*пример # 19 : абстракция и ее уточнения : Shape.java : Circle.java :
Rectangle.java */
package chapt05.abstraction;
import chapt05.implementor.*;
public abstract class Shape { //abstraction
    protected Color color;

    public Shape (){
      color = null;
    }
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }
    public abstract void performWithColor();
}
package chapt05.abstraction;
import chapt05.implementor.*;
public class Circle extends Shape {
    public Circle(Color color) {
        setColor(color);
    }
    public void performWithColor() {
       System.out.println("Performing in Circle class");
       color.useBrush();
    }
}
package chapt05.abstraction;
import chapt05.implementor.*;
public class Rectangle extends Shape {
    public Rectangle(Color color) {
        setColor(color);
    }
```

```
    public void performWithColor() {
       System.out.println("Performing in Rectangle class");
       color.useBrush();
    }
}
```

Класс **Shape** – абстракция, классы **Circle** и **Rectangle** – уточненные абстракции.

```
/*пример # 20 : использование шаблона Bridge : Main.java */
package chapt05.bridge;
import chapt05.abstraction.*;
import chapt05.implementor.*;
public class Main {
    public static void main(String args[]) {
        YellowColor color = new YellowColor();
        new Rectangle(color).performWithColor();
        new Circle(color).performWithColor();
    }
}
```

Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно динамически изменять и конфигурировать во время выполнения. Иерархии классов Abstraction и Implementor независимы и поэтому могут иметь любое число подклассов.

Шаблон Decorator

Необходимо расширить функциональные возможности объекта, используя прозрачный для клиента способ. Расширяемый класс реализует тот же самый интерфейс, что и исходный класс, делегируя исходному классу выполнение базовых операций. Шаблон Decorator даёт возможность динамического изменения поведения объектов в процессе выполнения приложения.

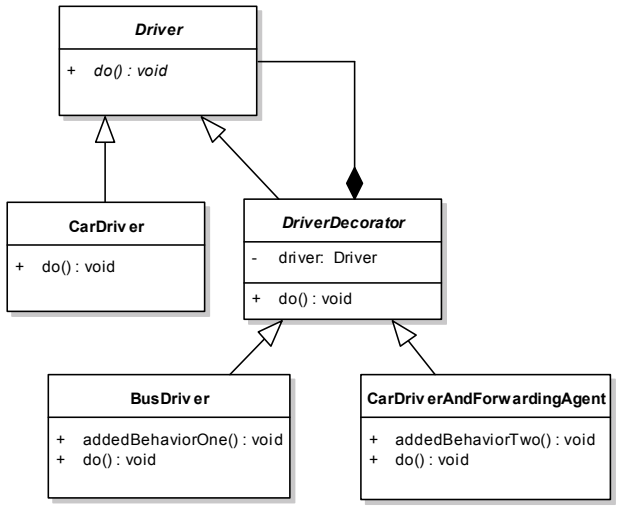

Рис. 5.10. Пример реализации шаблона Decorator

```
/*пример # 21 : определение интерфейса для компонентов : Driver.java */
package chapt05.decorator;
public abstract class Driver {
      public abstract void do();
   }
```

Класс **DriverDecorator** определяет для набора декораторов интерфейс, соответствующий интерфейсу класса **Driver**, и создает необходимые ссылки.

```
/*пример # 22 : интерфейс-декоратор для класса Driver : DriverDecorator.java */
package chapt05.decorator;
public abstract class DriverDecorator extends Driver {
      protected Driver driver;

      public DriverDecorator(Driver driver) {
            this.driver = driver;
      }
      public void do() {
            driver.do();
      }
   }
```

Класс **CarDriver** определяет класс, функциональность которого будет расширена.

```
/*пример # 23 : класс просто водителя : CarDriver.java */
package chapt05.decorator;
public class CarDriver extends Driver {
      public void do() { //базовая операция
            System.out.println("I am a driver");
      }
   }
```

Класс **BusDriver** добавляет дополнительную функциональность **addedBehaviorOne()** необходимую для водителя автобуса, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 24 : класс водителя автобуса: BusDriver.java */
package chapt05.decorator;
public class BusDriver extends DriverDecorator {

      public BusDriver(Driver driver) {
            super(driver);
      }
      private void addedBehaviorOne() {
            System.out.println("I am bus driver");
      }
      public void do() {
            driver.do();
            addedBehaviorOne();
      }
   }
```

Класс **CarDriverAndForwardingAgent** добавляет дополнительную функциональность **addedBehaviorTwo()** необходимую для водителя-экспедитора, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 25 : класс водителя-экспедитора:CarDriverAndForwardingAgent.java*/
package chapt05.decorator;
public class CarDriverAndForwardingAgent
                              extends DriverDecorator {

      public CarDriverAndForwardingAgent(Driver driver){
            super(driver);
      }
      private void addedBehaviorTwo() {
            System.out.println("I am a Forwarding Agent");
      }
      public void do() {
            driver.do();
            addedBehaviorTwo();
      }
   }
```

Создав экземпляр класса **CarDriver** можно делегировать ему выполнение задач, связанных с водителем автобуса или водителя-экспедитора, без написания специализированных полновесных классов.

```
/*пример # 26 : использование шаблона Decorator : Main.java */
package chapt05.decorator;
public class Main {
      public static void main(String args[]){
            Driver carDriver = new CarDriver();
            Main runner = new Main();
            runner.doDrive(carDriver);

            runner.doDrive(new BusDriver(carDriver));
            runner.doDrive(
            new CarDriverAndForwardingAgent(carDriver));
            }
      public void doDrive(Driver driver){
            driver.do();
      }
   }
```

Шаблоны поведения

Шаблоны поведения характеризуют способы взаимодействия классов или объектов между собой.

К шаблонам поведения относятся:

Chain of Responsibility (Цепочка Обязанностей) – организует независимую от объекта-отправителя цепочку не знающих возможностей друг друга объектов-получателей, которые передают запрос друг другу;

Command (**Команда**) – используется для определения по некоторому признаку конкретного класса, которому будет передан запрос для обработки;

Iterator (**Итератор**) – позволяет последовательно обойти все элементы коллекции или другого составного объекта, не зная деталей внутреннего представления данных;

Mediator (**Посредник**) – позволяет снизить число связей между классами при большом их количестве, выделяя один класс, знающий все о методах других классов;

Memento (**Хранитель**) – сохраняет текущее состояние объекта для дальнейшего восстановления;

Observer (**Наблюдатель**) – позволяет при зависимости между объектами типа «один ко многим» отслеживать изменения объекта;

State (**Состояние**) – позволяет объекту изменять свое поведение за счет изменения внутреннего объекта состояния;

Strategy (**Стратегия**) – задает набор алгоритмов с возможностью выбора одного из классов для выполнения конкретной задачи во время создания объекта;

Template Method (**Шаблонный Метод**) – создает родительский класс, использующий несколько методов, реализация которых возложена на производные классы;

Visitor (**Посетитель**) – представляет операцию в одном или нескольких связанных классах некоторой структуры, которую вызывает специфичный для каждого такого класса метод в другом классе.

Шаблон Command

Необходимо создать объект-команду, метод которого может быть вызван, а сам объект может быть сохранен и передан в качестве параметра метода или возвращен как любой другой объект. Инкапсулирует запрос как объект.

Объект-источник запроса отделяется от команды, но от его типа зависит, какая из команд будет выполнена.

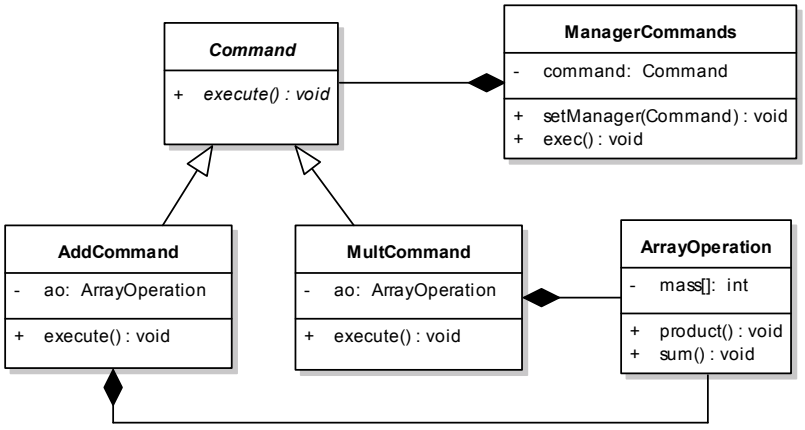

Рис. 5.11. Пример реализации шаблона Command

```
/*пример # 27 : описание команды и конкретные реализации : Command.java :
AddCommand.java : MultCommand.java */
package chapt05.command;
public abstract class Command {
      public abstract void execute();
}
package chapt05.command;
import chapt05.receiver.*;
public class MultCommand extends Command {
      private ArrayOperation ao;

      public MultCommand (ArrayOperation ao) {
            this.ao  = ao;
      }
      public  void execute () {
            this.ao.product();
      }
}
package chapt05.command;
import chapt05.receiver.*;
public class AddCommand extends Command {
      private ArrayOperation ao;

      public AddCommand (ArrayOperation ao) {
            this.ao  = ao;
      }
      public  void execute() {
            this.ao.sum();
      }
}
/*пример # 28 : класс Receiver (получатель) - располагает информацией о
способах выполнения операций : ArrayOperation.java */
package chapt05.receiver;
public class ArrayOperation {
      private int[] mass;

      public ArrayOperation(int[] mass) {
            this.mass = mass;
      }
      public void sum() {
            int sum = 0;
            for (int i : mass)
                  sum += i;
            System.out.println(sum);
      }
      public void product() {
            int mult = 1;
            for (int i : mass)
```

```
                mult *= i;
            System.out.println(mult);
        }
    }
```

*/*пример # 29 : класс Invoker (инициатор)-обращается к команде для выполнения запроса : ManagerCommands.java */*

```
package chapt05.invoker;
import chapt05.command.*;
public class ManagerCommands {
      private Command command;

      public ManagerCommands(Command command) {
            this.command = command;
      }
      public void setManager(Command command) {
            this.command = command;
      }
      public void exec() {
            command.execute();
      }
}
```

*/*пример # 30 : простое использование шаблона Command : Main.java */*

```
package chapt05;
import chapt05.invoker.*;
import chapt05.receiver.*;
import chapt05.command.*;
public class Main {
      public static void main(String[] args) {
            int mass[] = {-1, 71, 45, -20, 48, 60, 19};
      /*класс получатель(Receiver)-располагают информацией о способах
                                                выполнения операций*/
      ArrayOperation receiver = new ArrayOperation (mass);
      //инициализация команды
      Command operation1 = new MultCommand(receiver);
      Command operation2 = new AddCommand(receiver);
//класс инициатор (Invoker)-обращается к команде для выполнения запроса
ManagerCommands manager = new ManagerCommands(operation1);
            manager.exec();
            manager.setManager(operation2);
            manager.exec();
      }
   }
```

Объект-команда получен прямой инициализацией на основе переданного параметра. На практике данный объект создается или извлекается из коллекции на основе признака вызываемой команды. Объект **ManagerCommands** инициализируется командой и обладает простым интерфейсом для выполнения специализируемой задачи. В этом случае появляется возможность изменять реакцию приложения на запрос команды простой заменой объекта-управления.

Шаблон Strategy

Необходимо определить семейство алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

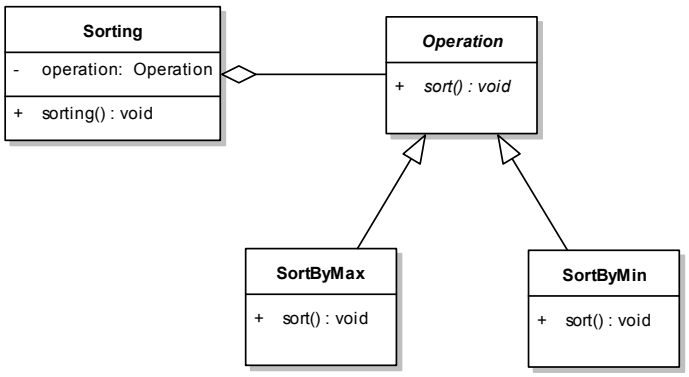

Рис. 5.12. Пример реализации шаблона Strategy

Класс **Operation** объявляет общий для всех поддерживаемых алгоритмов интерфейс, которым пользуется класс **Sorting** для вызова конкретного алгоритма, определенного в классе **SortByMax** или **SortByMin**. Класс **Sorting** конфигурируется объектом класса **SortByXxx**, объявляет ссылку на объект класса **Operation** и может определять интерфейс, позволяющий объекту **Operation** получить доступ к информации, в данном случае для сортировки массива.

Использование шаблона позволяет отказаться от условных операторов при выборе нужного поведения. Стратегии могут предлагать различные реализации одного и того же поведения. Класс-клиент вправе выбирать подходящую стратегию в зависимости от своих требований.

```
/*пример # 31 : общий интерфейс и классы конкретных стратегий :
Operation.java : SortByMax.java : SortByMin.java */
package chapt05.strategy;
public abstract class Operation {
      public abstract void sort(int mass[]);
}
package chapt05.strategy;
public class SortByMax extends Operation {
      public void sort(int mass[]) {
            for (int i = 0; i < mass.length; ++i) {
                  for (int j = i; j < mass.length; ++j) {
                        if (mass[j] > mass[i]){
                              int m = mass[i];
                              mass[i] = mass[j];
                              mass[j] = m;
                        }
                  }
            }
```

```
            System.out.print("SortByMax : ");
            for (int i : mass)
                  System.out.print(i + " ");
            System.out.println('\n');           }
}
package chapt05.strategy;
public class SortByMin extends Operation {
      public void sort(int mass[]) {
            for (int i = 0; i < mass.length; ++i) {
                  for (int j = i; j < mass.length; ++j) {
                        if (mass[j] < mass[i]){
                              int m = mass[i];
                              mass[i] = mass[j];
                              mass[j] = m;
                        }
                  }
            }
            System.out.print("SortByMin : ");
            for (int i : mass)
                  System.out.print(i + " ");
            System.out.println('\n');}
}
/*пример # 32 : класс выбора стратегии : Sorting.java */
package chapt05.strategy;
public class Sorting {
    private Operation operation = null;
    public Sorting(int operation){
        switch(operation) {
            case 1: this.operation = new SortByMax();
                        break;
            case 2: this.operation = new SortByMin();
                        break;
            default: System.out.println(
                        "Такая операция отсутствует");
                  }
     }
    public void sorting(int[] mass) {
      if (operation != null) operation.sort(mass);
      else return;
   }
}
/*пример # 33 : использование шаблона Strategy: Main.java */
package chapt05.strategy;
public class Main {
    public static void main(String args[]) {
      int mass[] = {28, 9, 71, 8, 35, 5, 51};
      Sorting cont1 = new Sorting(1);
      Sorting cont2 = new Sorting(2);
```

```
        cont1.sorting(mass);
        cont2.sorting(mass);
      }
}
```

Шаблон Observer

Требуется определить связь «один ко многим» между объектами таким образом, чтобы при изменении состояния одного объекта все связанные с ним объекты оповещались об этом и автоматически изменяли свое состояние. В языке Java этот шаблон используется под названием Listener.

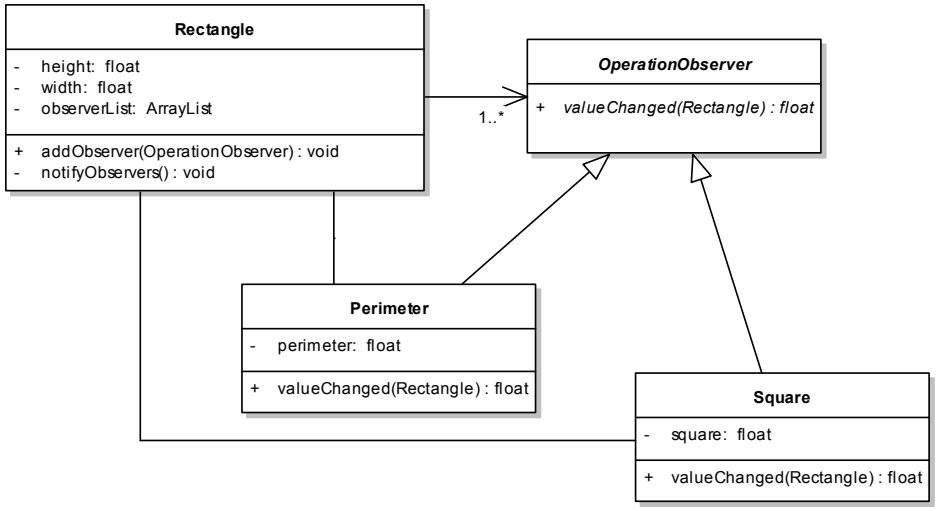

Рис. 5.13. Пример реализации шаблона Observer

Класс **Rectangle** (субъект) располагает информацией о своих наблюдателях и предоставляет интерфейс для регистрации и уведомления наблюдателей. Класс **OperationObserver** (наблюдатель) определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта. Класс **Perimeter** (конкретный наблюдатель) хранит или получает ссылку на объект класса **Rectangle**, сохраняет данные и реализует интерфейс обновления, определенный в классе **OperationObserver** для поддержки согласованности с субъектом.

Шаблон обеспечивает автоматическое уведомление всех подписавшихся на него объектов. Кроме этого, применение шаблона Observer абстрагирует связь субъекта и наблюдателя. Субъект имеет информацию только о том, что у него есть некоторое число наблюдателей, каждый из которых подчиняется интерфейсу абстрактного класса-наблюдателя.

```
/*пример # 34 : класс-субъект, за которым следят все классы-наблюдатели :
Rectangle.java */
package chapt05.observer;
import java.util.*;
```

```
public class Rectangle {
      private float width;
      private float height;
      private ArrayList<OperationObserver> observerList =
                        new ArrayList<OperationObserver>();
      public Rectangle(float width, float height) {
            this.width = width;
            this.height = height;
      }
      public void addObserver(OperationObserver observer) {
            observerList.add(observer);
      }
      public float getWidth() {
            return width;
      }
      public float getHeight() {
            return height;
      }
      public void setWidth(float width) {
            this.width = width;
            notifyObservers();
      }
      public void setHeight(float height) {
            this.height = height;
            notifyObservers();
      }
      private void notifyObservers() {
            Iterator it = observerList.iterator();
            while (it.hasNext()) {
      ((OperationObserver) it.next()).valueChanged(this);
            }
      }
      public String toString() {
            String s = "";
            Iterator it = observerList.iterator();
            while (it.hasNext()) {
                  s = s +
      ((OperationObserver) it.next()).toString() + '\n';
            }
            return s;
      }
}
```

Классы **Perimeter** и **Square** наследуются от абстрактного класса **OperationObserver** и являются наблюдателями. Как только субъект **Rectangle** изменяется, состояние этих объектов также подвергается изменению в соответствии с реализованным интерфейсом.

```
/*пример # 35 : классы-наблюдатели : OperationObserver.java : Square.java :
Perimeter.java */
package chapt05.observer;
public abstract class OperationObserver {
    public abstract float valueChanged(Rectangle observed);
}
package chapt05.observer;
public class Perimeter extends OperationObserver {
      private float perimeter;
      public float valueChanged(Rectangle observed) {
            return perimeter =
      2 * (observed.getWidth() + observed.getHeight());
      }
      public String toString() {
            return "P = " + perimeter;
      }
}
package chapt05.observer;
public class Square extends OperationObserver  {
      private float square;
      public float valueChanged(Rectangle observed) {
            return square =
            observed.getWidth() * observed.getHeight();
      }
      public String toString() {
            return "S = " + square;
      }
}
/*пример # 36 : использование шаблона Observer : Main.java */
package chapt05.observer;
public class Main {
    public static void main(String args[]) {
        Rectangle observed = new Rectangle(5, 3);
        System.out.println(observed.toString());
        observed.addObserver(new Square());
        observed.addObserver(new Perimeter());
        observed.setWidth(10);
        System.out.println(observed.toString());
        observed.setHeight(8);
        System.out.println(observed.toString());
    }
}
```

Антишаблоны проектирования

Big ball of mud. «Большой Ком Грязи» – термин для системы или просто программы, которая не имеет хоть немного различимой архитектуры. Как правило, включает в себя более одного антишаблона. Этим страдают системы, разработанные людьми без подготовки в области архитектуры ПО.

Software Bloat. «Распухание ПО» – пренебрежительный термин, используемый для описания тенденций развития новейших программ в направлении использования бóльших объемов системных ресурсов (место на диске, ОЗУ), чем предшествующие версии. В более общем контексте применяется для описания программ, которые используют больше ресурсов, чем необходимо.

Yo-Yo problem. «Проблема Йо-Йо» возникает, когда необходимо разобраться в программе, иерархия наследования и вложенность вызовов методов которой очень длинна и сложна. Программисту вследствие этого необходимо лавировать между множеством различных классов и методов, чтобы контролировать поведение программы. Термин происходит от названия игрушки йо-йо.

Magic Button. Возникает, когда код обработки формы сконцентрирован в одном месте и, естественно, никак не структурирован.

Magic Number. Наличие в коде многократно повторяющихся одинаковых чисел или чисел, объяснение происхождения которых отсутствует.

Gas Factory. «Газовый Завод» – необязательный сложный дизайн или для простой задачи.

Analiys paralisys. В разработке ПО «Паралич анализа» проявляет себя через чрезвычайно длинные фазы планирования проекта, сбора необходимых для этого артефактов, программного моделирования и дизайна, которые не имеют особого смысла для достижения итоговой цели.

Interface Bloat. «Распухший Интерфейс» – термин, используемый для описания интерфейсов, которые пытаются вместить в себя все возможные операции над данными.

Smoke And Mirrors. Термин «Дым и Зеркала» используется, чтобы описать программу либо функциональность, которая еще не существует, но выставляется за таковую. Часто используется для демонстрации финального проекта и его функционала.

Improbability Factor. «Фактор Неправдоподобия» – ситуация, при которой в системе наблюдается некоторая проблема. Часто программисты знают о проблеме, но им не разрешено ее исправить отчасти из-за того, что шанс всплытия наружу у этой проблемы очень мал. Как правило (следуя закону Мерфи), она всплывает и наносит ущерб.

Creeping featurism. Используется для описания ПО, которое выставляет напоказ вновь разработанные элементы, доводя до высокой степени ущербности по сравнению с ними другие аспекты дизайна, такие как простота, компактность и отсутствие ошибок. Как правило, существует вера в то, что каждая новая маленькая черта информационной системы увеличит ее стоимость.

Accidental complexity. «Случайная сложность» – проблема в программировании, которой легко можно было избежать. Возникает вследствие неправильного понимания проблемы или неэффективного планирования.

Ambiguous viewpoint. Объектно-ориентированные модели анализа и дизайна представляются без внесения ясности в особенности модели. Изначально эти модели обозначаются с точки зрения визуализации структуры программы. Двусмысленные точки зрения не поддерживают фундаментального разделения интерфейсов и деталей представления.

Boat anchor. «Корабельный Якорь» – часть бесполезного компьютерного «железа», единственное применение для которого – отправить на утилизацию. Этот термин появился в то время, когда компьютеры были больших размеров. В

настоящее время термин «Корабельный Якорь» стал означать классы и методы,, которые по различным причинам не имеют какого-либо применения в приложении и в принципе бесполезны. Они только отвлекают внимание от действительно важного кода.

Busy spin. Техника, при которой процесс непрерывно проверяет изменение некоторого состояния, например ожидает ввода с клавиатуры или разблокировки объекта. В результате повышается загрузка процессора, ресурсы которого можно было бы перенаправить на исполнения другого процесса. Альтернативным путем является использование сигналов. Большинство ОС поддерживают погружение потока в состояние «сон» до тех пор, пока ему отправит сигнал другой поток в результате изменения своего состояния.

Caching Failure. «Кэширование Ошибки» – тип программного бага (bug), при котором приложение сохраняет (кэширует) результаты, указывающие на ошибку даже после того, как она исправлена. Программист исправляет ошибку, но флаг ошибки не меняет своего состояния, поэтому приложение все еще не работает.

Задания к главе 5

Вариант А

Выполнить описание логики системы и использовать шаблоны проектирования для определения организации классов разрабатываемой системы. Использовать объекты классов и подклассов для моделирования реальных ситуаций и взаимодействий объектов.

1. Создать суперкласс **Транспортное средство** и подклассы **Автомобиль**, **Велосипед**, **Повозка**. Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.
2. Создать суперкласс **Пассажироперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Задать правила выбора транспорта в зависимости от расстояния и наличия путей сообщения.
3. Создать суперкласс **Учащийся** и подклассы **Школьник** и **Студент**. Определить способы обучения и возможности его продолжения.
4. Создать суперкласс **Музыкальный инструмент** и классы **Ударный, Струнный**, **Духовой**. Определить правила организации и управления оркестром.
5. Создать суперкласс **Животное** и подклассы **Собака**, **Кошка**, **Тигр**, **Мустанг**, **Дельфин**. С помощью шаблонов задать способы обитания.
6. Создать базовый класс **Садовое дерево** и производные классы **Яблоня**, **Вишня**, **Груша**, **Слива**. Принять решение о пересадке каждого дерева в зависимости от возраста и плодоношения.

Тестовые задания к главе 5

Вопрос 5.1.

Какой шаблон создает объекты путем их копирования?

1) Factory;
2) Prototype;
3) Builder;
4) Singleton.

Вопрос 5.2.

Какие из шаблонов относятся к порождающим? (выберите два)

1) Factory;
2) Command;
3) Strategy;
4) Singleton.

Вопрос 5.3.

Какой шаблон позволяет обращаться к группе объектов таким же образом как и к одному?

1) Visitor;
2) Composite;
3) Prototype;
4) Observer.

Вопрос 5.4.

Какой шаблон реализует постоянную часть алгоритма, а реализацию изменяемой оставляет потомкам?

1) Strategy;
2) Decorator;
3) Template Method;
4) Visitor.

Вопрос 5.5.

Какой шаблон подменяет собой сложный объект и контролирует доступ к нему.

1) Adapter;
2) Decorator;
3) Proxy;
4) Bridge.