【软件构造】第五章第三节 可复用的设计模式

第五章第三节 可复用的设计模式

除了Framework，5-2节所讨论的其他技术都过于“基础”和“细小”，有没有办法做更大规模的复用设计？
本节将介绍几种典型的“面向复用”的设计模式，设计模式更强调多个类/对象之间的关系和交互过程—比接口/类复用的粒度更大。

Outline

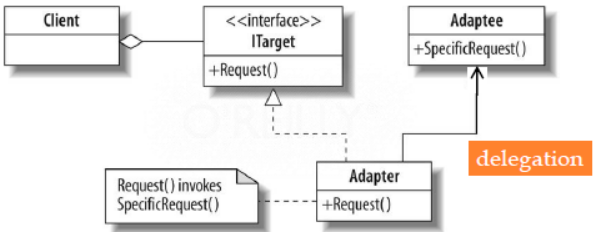
- 结构型模式：Structural patterns
  - 适配器模式（Adapter）
  - 装饰器模式（Decorator ）
  - 外观模式（Facade）
- 行为类模式：Behavioral patterns
  - 策略模式（Strategy）
  - 模板方法模式（Template method）
  - 迭代器模式（ Iterator）

Notes

结构型模式：**Structural patterns**

【适配器模式（**Adapter**）】

- 目的：将某个类/接口转换为用户期望的其他形式。
- 含义：适配器模式是作为两个互不相容的接口的桥梁，将某个类/接口转换为client期望的其他形式。适配器模式使得原本由于接口不兼容而不能一起工作的的那些类/接口可以一起工作。
- 用途：主要解决在软件系统中，需要将现存的类放到新的环境中，而环境要求的接口是现对象不能满足的。
- 实现方法：通过增加一个新的接口，将已存在的子类封装起来，client直接面向接口编程，从来隐藏了具体子类。适配器继承或依赖已有的对象，实现想要的目标接口。
- 对象：将旧组件重用到新系统（也称为“包装器”）。
- 模型：

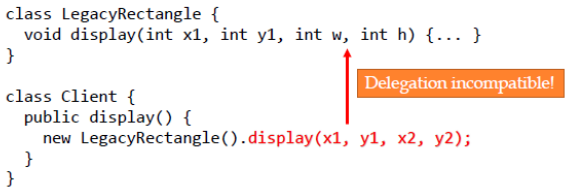


- 实例：

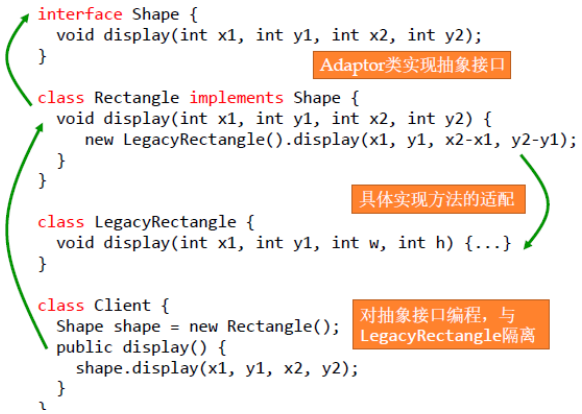
问题描述：其中LegacyRectangle是已有的类（需要传入矩形的一个顶点、长和宽），但是与client要求的接口不一致（需要给出对角线两个顶点坐标），我们此时建立一个新的接口Shape以供客户端调用，用户通过Shape接口传入两个点的坐

标。Rectangle作为Adapter，实现该抽象接口，通过具体的方法实现适配。

在不适用适配器时：会发生委派不相容。

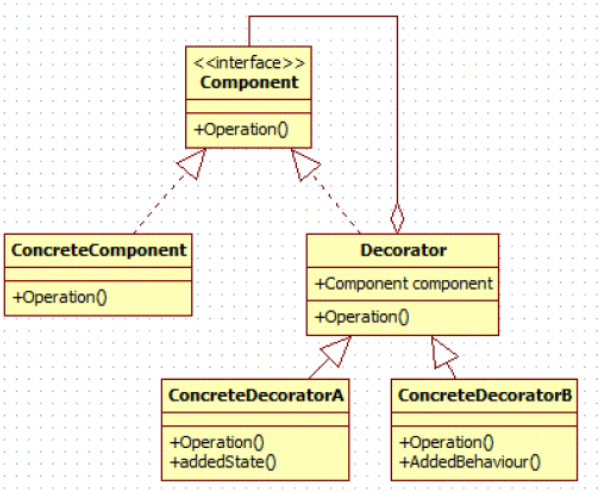


使用了适配器后就能够解决上述问题：



【装饰器模式（**Decorator**）】

- 含义：
  - 装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构，它是作为一个现有的类的一个包装。
  - 这种模式创建了一个装饰类，用来包装原有的类，并在保证类方法签名完整性的前提下，提供额外的功能。
  - 装饰模式是继承的一个代替模式，装饰模式可以动态地给一个对象添加一些额外的职能。就增加功能来说，装饰器模式比生成子类更为灵活。
- 主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。
- 实现方法：将具体功能职责划分，对每一个特性构造子类，通过委派机制增加到对象上。
- 用途：为对象增加不同侧面的特性。
- 优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。
- 缺点：多层装饰比较复杂，客户端需要一个具有多特性的object，需要一层一层的装饰来实现
- 模型：



- 实例**1**：对于**Shape**接口和实现　参考：装饰器模式 菜鸟教程
- 实例**2**：
  - 我们需要对**Stack**数据结构的各种扩展：
    - UndoStack：一个允许你撤销先前的push或pop操作的栈
    - SecureStack：一个需要密码的栈
    - SynchronizedStack：一个串行化并发访问的栈
  - 我们可以采用继承的方式来解决。之后我们又需要任意可组合的扩展：
    - SecureUndoStack：需要密码的堆栈，并且还可以撤消以前的操作
    - SynchronizedUndoStack：一个堆栈，用于序列化并发访问，还可以让您撤销先前的操作
    - SecureSynchronizedStack：需要密码的堆栈，并用于序列化并发访问的操作
  - 我们又怎么处理呢？继承层次结构？多继承？但是会显得很麻烦，这时候装饰器模式就可以很好地解决这一问题。

```
interface Stack {
   void push(Item e);
   Item pop();
}

public abstract class AbstractStackDecorator implements Stack {
   protected final Stack stack;
   public AbstractStackDecorator(Stack stack) {
      this.stack = stack;
   }
   public void push(Item e) {
      stack.push(e);
   }
   public Item pop() {
      return stack.pop();
   }
   ...
}
```

Delegation (aggregation)

```
public class UndoStack
       extends AbstractStackDecorator
       implements Stack {

   private final UndoLog log = new UndoLog();
   public UndoStack(Stack stack) {
      super(stack);
   }
   public void push(Item e) {
      log.append(UndoLog.PUSH, e);
      super.push(e);
   }
   public void undo() {
      //implement decorator behaviors on stack
   }
   ...
}
```

增加了新特性

基础功能通过 delegation实现

增加了新特性

我们需要一层层具有多种特征的object，通过一层层的装饰来实现：

```
Stack s = new ArrayStack(); //构建一个普通的堆栈
UndoStack s = new UndoStack(new ArrayStack()); //构建撤消堆栈
```

```
SecureStack s = new SecureStack( new SynchronizedStack( new UndoStack(s)))//构建安全的同步撤消堆栈
```

- 装饰器 vs. 继承
  - 装饰器在运行时组成特征；继承在编译时组成特征。
  - 装饰器由多个协作对象组成；继承产生一个明确类型的对象。
  - 可以混合和匹配多个装饰；多重继承在概念上是困难的。
- java.util.Collections中也有一些装饰器模式：
- 将mutable list 变为 immutable list:

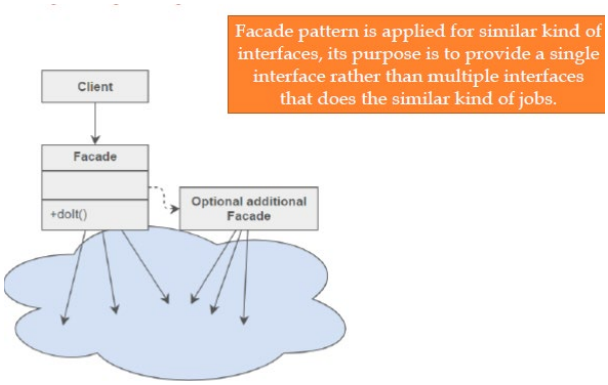
```
static List<T>  unmodifiableList(List<T> lst);
static Set<T>   unmodifiableSet( Set<T> set);
static Map<K,V> unmodifiableMap( Map<K,V> map);
```

- Similar for synchronization:

```
static List<T>  synchronizedList(List<T> lst);
static Set<T>   synchronizedSet( Set<T> set);
static Map<K,V> synchronizedMap( Map<K,V> map);
```

【外观模式（**Facade Pattern**）】

- 含义：外观模式隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。
- 意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
- 主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。
- 实现方法：提供一个统一的接口来取代一系列小接口调用，相当于对复杂系统做了一个封装，简化客户端使用。便于客户端学习，解耦 。
- 用途：为了解决客户端需要通过一个简化的接口来访问复杂系统内的功能这一问题提出的。
- 优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。
- 缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。
- 模式：



- 实例一：定义外观类 ***ShapeMaker***，并对其进行实现 参考 外观模型 菜鸟教程
- 实例二：

假设我们有一个具有一组接口的应用程序来使用MySql / Oracle数据库，并生成不同类型的报告，如HTML报告，PDF报告

等。

因此，我们将有不同的接口集合来处理不同类型的数据库。现在客户端应用程序可以使用这些接口来获取所需的数据库连接并生成报告。但是，当复杂性增加或界面行为名称混淆时，客户端应用程序将难以管理它。

因此，我们可以在这里应用Facade模式，并在现有界面的顶部提供包装界面以帮助客户端应用程序。

Two Helper Classes for MySQL and Oracle： 分别封装了客户端所需的功能

```
public class MySqlHelper {

  public static Connection getMySqlDBConnection() {…}
  public void generateMySqlPDFReport
            (String tableName, Connection con){…}
  public void generateMySqlHTMLReport
            (String tableName, Connection con){…}
}

public class OracleHelper {

  public static Connection getOracleDBConnection() {…}
  public void generateOraclePDFReport
            (String tableName, Connection con){…}
  public void generateOracleHTMLReport
            (String tableName, Connection con){…}
}
```

A façade class:

```
public class HelperFacade {
  public static void generateReport
     (DBTypes dbType, ReportTypes reportType, String tableName){
        Connection con = null;
        switch (dbType){
        case MYSQL:
          con = MySqlHelper.getMySqlDBConnection();
          MySqlHelper mySqlHelper = new MySqlHelper();
          switch(reportType){
                case HTML:
                  mySqlHelper.generateMySqlHTMLReport(tableName, con);
                  break;
                case PDF:
                  mySqlHelper.generateMySqlPDFReport(tableName, con);
                  break;
          }
          break;
          case ORACLE: …
        }
        public static enum DBTypes      { MYSQL,ORACLE; }
        public static enum ReportTypes  { HTML,PDF;}
}
```

客户端代码：

Client code

Without facade

```
String tableName="Employee";

Connection con = MySqlHelper.getMySqlDBConnection();
MySqlHelper mySqlHelper = new MySqlHelper();
mySqlHelper.generateMySqlHTMLReport(tableName, con);

Connection con1 = OracleHelper.getOracleDBConnection();
OracleHelper oracleHelper = new OracleHelper();
oracleHelper.generateOraclePDFReport(tableName,
```

With facade

```
HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
HelperFacade.ReportTypes.HTML, tableName);
HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
HelperFacade.ReportTypes.PDF, tableName);
```

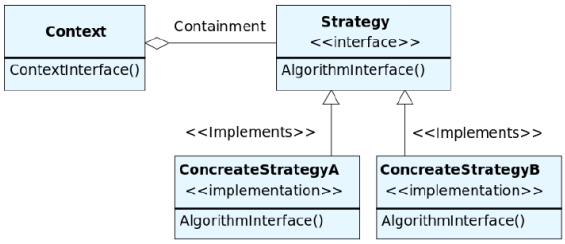
我们可以看到采用了Facade设计模式的客户端代码简洁了许多，更方便客户使用。

行为类模式：**Behavioral patterns**

【策略模式（**Strategy**）】

- 含义：在策略模式中，一个类的行为或算法可以在运行时更改。在策略模式中，我们创建表示各种模式的对象和一个行为随着策略对象改变而改变的 context对象，策略对象改变 context对象的执行算法。
- 用途：针对特定任务存在不同的算法，但客户端可以根据动态上下文在运行时切换算法。

- 实现方法：为算法创建一个接口，并为算法的每个变体创建一个实现类。
- 解决方法：将这些算法封装成一个一个的类，任意地替换
- 优点：算法可以自由切换、避免使用多重条件判断、扩展性良好。
- 缺点：策略类会增多、所有策略类都需要对外暴露。
- 模式：



- 实例一：**Strategy**接口与**Contex**实现 参考：策略模式 菜鸟驿站
- 实例二：对于**Lab3**中的内容进行抽象
- 问题描述：对于一个图，我们可以计算图中顶点的不同类型的中心度，如degreeCentrality（点度中心度）、closenessCentrality（接近中心度）和betweenCentrality（中介中心度）。
  - 用Strategy原则：
  - **CentralityStrategy.java**

```
public interface CentralityStrategy {     //对应上图的 Strategy<<interface>>
    public abstract centrality();
}
```

  - 然后通过三个具体的实现类来实现这个接口：
  - **degreeCentralityStrategy.java**

```
 1 public class degreeCentralityStrategy<L extends Vertex, E extends Edge> implements
CentralityStrategy {
 2
 3     private final Graph<L, E> g;
 4     private final L v;
 5
 6     public degreeCentralityStrategy(Graph<L, E> g, L v) {
 7         this.g = g;
 8         this.v = v;
 9     }
10
11     @Override
12     public double centrality() {
13         return GraphMetrics.degreeCentrality(g, v);
14     }
15 }
```

  - **closenessCentralityStrategy.java**

```
 1 public class closenessCentralityStrategy<L extends Vertex, E extends Edge> implements
CentralityStrategy {
 2
 3     private final Graph<L, E> g;
 4     private final L v;
 5
 6     public closenessCentralityStrategy(Graph<L, E> g, L v) {
 7         this.g = g;
 8         this.v = v;
 9     }
10
11     @Override
12     public double centrality() {
13         return GraphMetrics.closenessCentrality(g, v);
14     }
15 }
```

- **betweennessCentralityStrategy.java**

```
 1 public class betweennessCentralityStrategy<L extends Vertex, E extends Edge> implements
CentralityStrategy{
 2
 3     private final Graph<L, E> g;
 4     private final L v;
 5
 6     public betweennessCentralityStrategy(Graph<L, E> g, L v) {
 7         this.g = g;
 8         this.v = v;
 9     }
10
11     @Override
12     public double centrality() {
13         return GraphMetrics.betweennessCentrality(g, v);
14     }
15 }
```

- 然后实现一个**Context**类 **centralityContext.java**

```
public class CentralityContext {

    public double centrality(CentralityStrategy centralityType) {
        return centralityType.centrality();
    }
}
```

- 可以使用**Context**来查看当它改变策略**Strategy**时的行为变化，如下**Main.java**

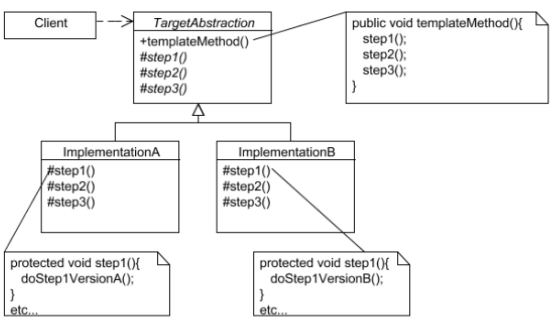
```
public class Main {

    public static void main(String[] args) {

        Graph<Vertex, Edge> graph = GraphFactory.createGraph("test/graph/GraphPoetTest.txt");
        String[] strings = {"F", "24"};
        Vertex vertex1 = VertexFactory.createVertex("to", "Word", strings);
        CentralityContext context = new CentralityContext();
        double degree = context.centrality(new DegreeCentralityStrategy<>(graph, vertex1));
        double closeness = context.centrality(new ClosenessCentralityStrategy<>(graph, vertex1));
        double betweenness = context.centrality(new BetweennessCentralityStrategy<>(graph, vertex1));
        System.out.println(degree);
        System.out.println(closeness);
        System.out.println(betweenness);
    }
}
```

【模板模式（**Template method**）】

- 含义：在模板模式中，一个抽象类公开定义了执行它的方法的方式/模式，它的子类可以按照需要重写实现方法，但调用将以抽象类中定义的方法进行。
- 意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
- 实现方法：共性的步骤在抽象类内公共实现，差异化的步骤在各个子类中实现。一般使用继承和重写实现模板模式。
- 优点：封装不变部分，扩展可变部分；提取公共代码，便于维护；行为由父类控制吗，子类实现
- 缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。
- 模式：



- 实例一：**Game**抽象类和重写 参考 模板模式 菜鸟教程

- 实例二：对于**Lab3**中的内容进行抽象
- 问题描述：边有很多类型：简单的分为：有向边、无向边
  - **Edge.java** //创建扩展了上述类的实现类

```
1 public abstract class Edge {
2 
3     private final String label;
4     private final double weight;
5 
6     //the constructor
7     public Edge(String label, double weight) {
8         this.label = label;
9         this.weight = weight;
10    }
11 
12 public abstract boolean addVertices(List<Vertex> vertices);
13 
14 public abstract boolean containVertex(Vertex v);
15 
16 public abstract Set<Vertex> vertices();
```

  - **DirectedEdge.java**

```
1 public class DirectedEdge extends Edge{
2 
3     private Vertex source;
4     private Vertex target;
5 
6     //the constructor
7     public DirectedEdge(String label, double weight) {
8         super(label, weight);
9     }
10 
11        @Override
12    public boolean addVertices(List<Vertex> vertices) {
13        source = vertices.get(0);
14        target = vertices.get(1);
15        return true;
16    }
17 
18    @Override
19    public boolean containVertex(Vertex v) {
20        return source.equals(v) || target.equals(v);
21    }
22 
```

```
23      @Override
24      public Set<Vertex> vertices() {
25          Set<Vertex> set = new HashSet<Vertex>();
26          set.add(source);
27          set.add(target);
28          return set;
29      }
```

- **UndirectedEdge.java**

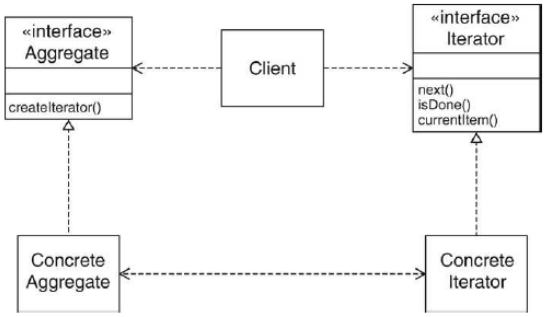
```
 1 public class UndirectedEdge extends Edge{
 2 
 3     private Vertex vertex1;
 4     private Vertex vertex2;
 5 
 6     public UndirectedEdge(String label, double weight) {
 7         super(label, weight);
 8     }
 9 
10     @Override
11     public boolean addVertices(List<Vertex> vertices) {
12         vertex1 = vertices.get(0);
13         vertex2 = vertices.get(1);
14         return true;
15     }
16 
17     @Override
18     public boolean containVertex(Vertex v) {
19         return vertex1.equals(v) || vertex2.equals(v);
20     }
21 
22     @Override
23     public Set<Vertex> vertices() {
24         Set<Vertex> set = new HashSet<Vertex>();
25         set.add(vertex1);
26         set.add(vertex2);
27         return set;
28     }
```

【迭代器模式（**Iterator**）】

- 含义：这种模式用于顺序访问集合对象的元素，而又无需暴露该对象的内部表示。
- 用途：解决 客户需要统一的策略来访问容器中的所有元素，与容器类型无关
- 实现方法：这种模式让自己的集合类实现`Iterable`接口，并实现自己的独特`Iterator`迭代器(`hasNext`, `next`,

remove），允许客户端利用这 个迭代器进行显式或隐式的迭代遍历。

- 优点： 1、它支持以不同的方式遍历一个聚合对象。 2、迭代器简化了聚合类。 3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。
- 缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。
- 模式：



- 实例：
  - Iterable接口：实现该接口的集合对象是可迭代遍历的

```
public interface Iterable<T> {
    ...
    Iterator<T> iterator();
}
```

  - Iterator接口：迭代器

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

  - 具体例子见下图：