4.3 Design Patterns for Reuse

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂，Ex

目录

4.3.1 Design Patterns（设计模式）

一个设计模式是针对软件设计的特定需求，对于一些成功的软件设计的抽象。是对别人成功经验的有效总结。

对于一个设计模式而言，其不光看重类本身，而且看重多个类之间的交互关系。其复用粒度自然大于接口/类的复用粒度，但是其粒度是小与框架的。

分类

Creational patterns 创建型模式

关心于对象的创建

Structural patterns 结构型模式

关心于类的结构，类与类之间的关系

Behavioral patterns 行为类模式

关心于多个类的功能分配

4.3.2 some Structural patterns（一些结构型模式）

Adapter（适配器模式）

使用场合

适用于解决类之间接口不兼容的问题，可以将某个类或者接口转换为client期望的形式。

具体实现

问题：

A 调用 B，但 A 与 B 的调用参数不匹配。

解决办法：

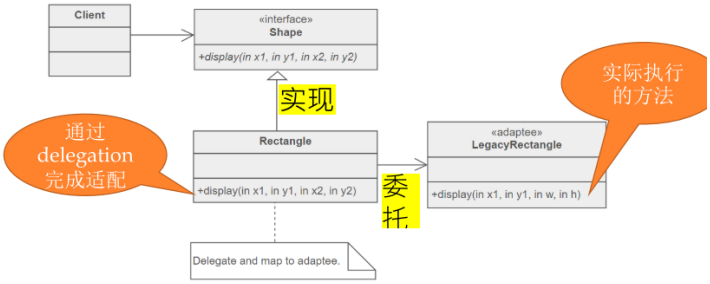
创建一个接口 C，C 满足 A 的调用参数，并且其规约中实现 B 的功能。其实定义一个 C 的实现类 D，由 D 完成调用参数的转换，并将其功能委托于 B。

之后客户端只需要定义一个接口 C 的引用，并将 D 的实例赋予 C 的引用，之后调用 C 的引用中需要的方法即可。

注：在这其中，设置接口 C 是为了增强程序的灵活性，可以实现不同的接口参数对于 A 的参数的统一，只需要创建对应的实现类，并将之赋予给 C 的引用。同时接口 C 也隔离了 client 与内部具体实现。

Ex：

- A `LegacyRectangle` component's `display()` method expects to receive "`x, y, w, h`" parameters. 左上角坐标+宽+高
- But the client wants to pass "upper left `x` and `y`" and "lower right `x` and `y`". 左上角坐标+右下角坐标
- This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object. ----Delegation



直接实现，出现调用参数不匹配。

```
class LegacyRectangle {
  void display(int x1, int y1, int w, int h) {... }
}

class Client {
  public display() {
    new LegacyRectangle().display(x1, y1, x2, y2);
  }
}
```

Delegation incompatible!

采用适配设计模式产生的结果。

```
interface Shape {
  void display(int x1, int y1, int x2, int y2);
}

class Rectangle implements Shape {
  void display(int x1, int y1, int x2, int y2) {
     new LegacyRectangle().display(x1, y1, x2-x1, y2-y1);
  }
}

class LegacyRectangle {
  void display(int x1, int y1, int w, int h) {...}
}

class Client {
  Shape shape = new Rectangle();
  public display() {
    shape.display(x1, y1, x2, y2);
  }
}
```

Adaptor类实现抽象接口

具体实现方法的适配

对抽象接口编程，与LegacyRectangle隔离

Decorator（装饰器模式）

使用场合

对于一个类相同的对象而言，其具有许多特性，但是每一个对象而言，其具有这些特性的子集。

如果强行实现，会产生大量的代码重复，违背了 DRY 原则。

对于继承的实现来讲存在许多的问题：这些特性之间很难梳理出较为合适的结构关系，因而难以形成合适的继承，其次，强行使用继承可能会导致继承许多无用的代码。

最为合适的方法是：类的组合，这种模式可以认为是类的组合的一种实现，其比 4.2 节中所讲的实现方式具有更强的复用程度，可以认为是上述方法在这种情景下的实例，专业度更强。

具体实现：

问题：

有一类对象集合 S = {A，B，……}，任意的 X 属于 S，X 具备共性方法集 M 的任意方法，且 X 具备特性方法集 N 的某一子集的任意方法。需要构造出相应的程序进行表达。

解决办法：

A. 定义一个具有公共方法集 M 的任意方法的接口 Component。

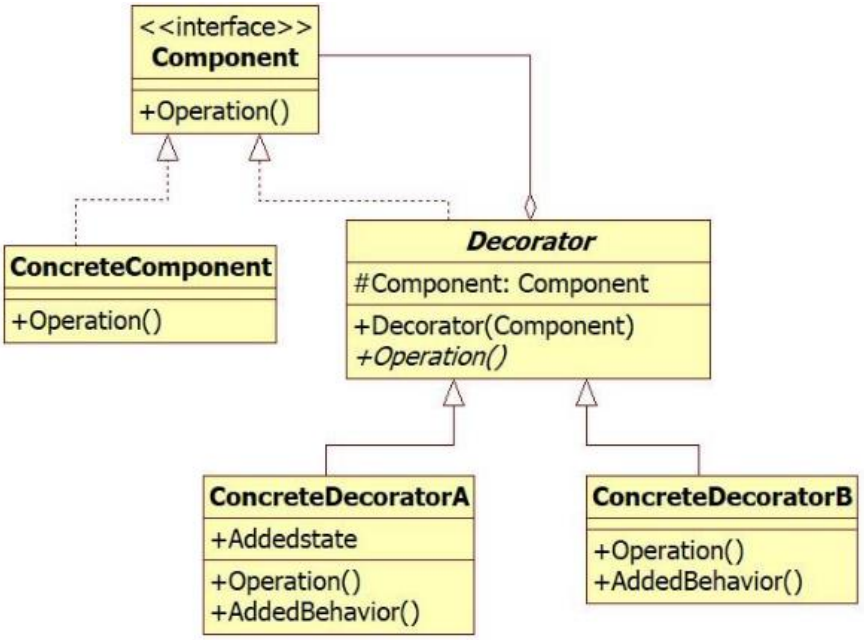
B. 定义一个实现了此接口的类 ConcreteComponent()，此类中仅仅实现了共性方法集。

C. 定义一个实现类 Component 的继承抽象类 Decorator：
   1. 成员变量：类型为 Component 的对象 Component
   2. 构造方法：其参数为 Component 的对象 Component，
      实现为 super. Component=Component.
   3. 需要实现的接口中的方法均委托于其属性变量 Component。
   4. 有必要的话，可以定义其他的抽象方法

D. 对于每个特性，定义其方法，这些方法均继承于 Decorator，命名为 ConcreteDecoratorA，ConcreteDecoratorB……

E. 客户端的实现可以使用嵌套定义，即 Component A = new ConcreteDecoratorD（ConcreteDecoratorE(ConcreteDecoratorF(ConcreteDecorator)))。此种完成装扮，其定义的对象可以使用其自由组合的特性。

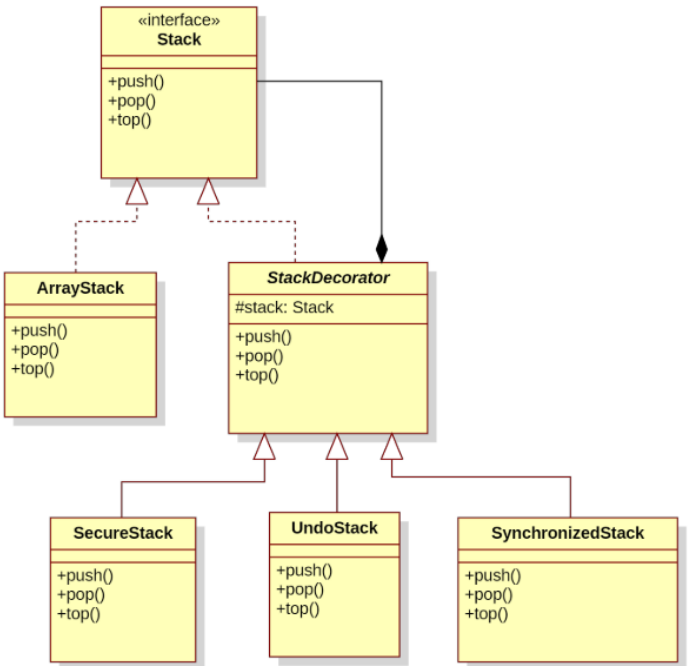
解决办法的形象图：



注：在客户端中需要对象的定义上由递归的存在，例如：Component A = new ConcreteDecoratorD（ConcreteDecoratorE(ConcreteDecoratorF(ConcreteDecorator)))，构造 ConcreteDecoratorD 先得构造 ConcreteDecoratorE，构造 ConcreteDecoratorE 先得构造 ConcreteDecoratorF，构造 ConcreteDecoratorF 先得构造 ConcreteDecorator，构造完成在逐步返回即可。

注：至于将此方法成为装扮方法，也是参照客户端中对象的定义来说明，这个设计模式最为核心的就是那个客户端中的对象的定义，即 Component A = new ConcreteDecoratorD（ConcreteDecoratorE(ConcreteDecoratorF(ConcreteDecorator)))。

Ex1：一个标准的实现例子，其中在 Decorator 未构造抽象方法，可以参照此例理解定义：




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

public class ArrayStack implements Stack {
   ... //rep

   public ArrayStack() {...}
   public void push(Item e) {
      ...
   }
   public Item pop() {
      ...
   }
   ...
}
```

实现最基础的Stack功能

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

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

给出一个用于decorator的基础类

Delegation (aggregation)

```
public class UndoStack extends StackDecorator  {

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

基础功能通过基类中的delegation实现

增加了新特性

增加了新特性

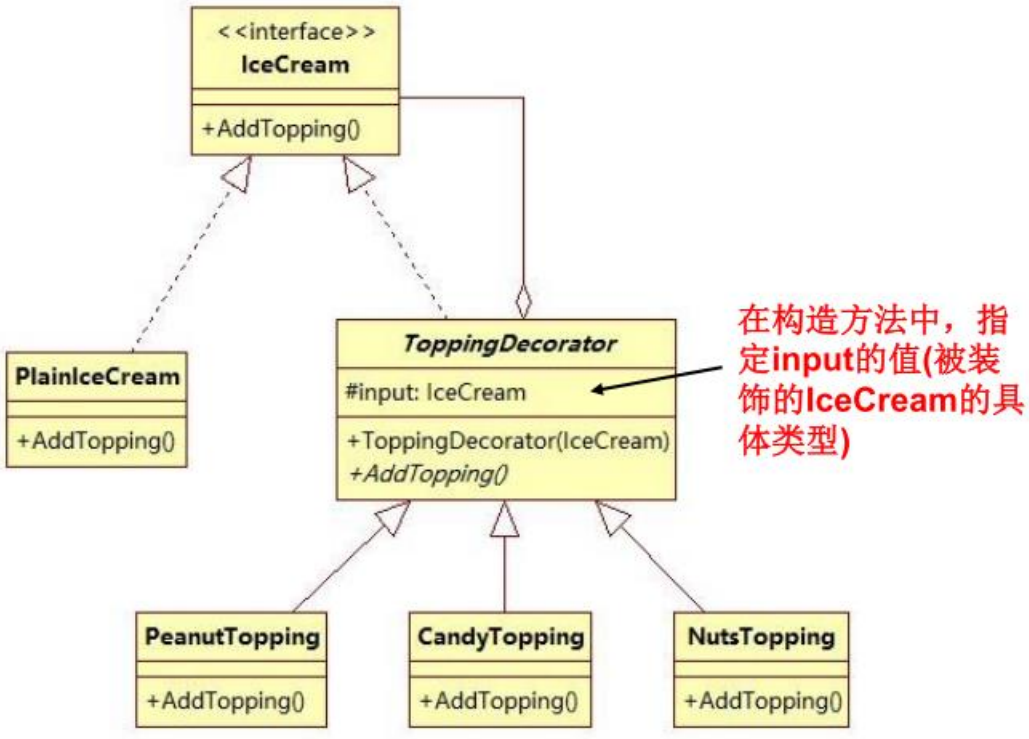
- To construct a plain stack:
  - `Stack s = new ArrayStack();`
- To construct an undo stack:
  - `Stack t = new UndoStack(new ArrayStack());`
- To construct a secure synchronized undo stack:
  -
```
Stack t = new SecureStack(
                    new SynchronizedStack(
                       new UndoStack(s))
t.push(e);
```

客户端需要一个具有多种特性的object，通过一层一层的装饰来实现

- **Flexibly Composable!**

就像一层一层的穿衣服…

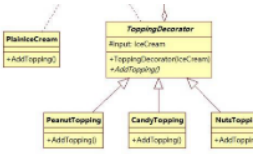
Ex2：另一个标准的实现例子，其中在 Decorator 构造抽象方法，可以参照此例理解定义：




```
public interface IceCream { //顶层接口
    void AddTopping();
}
public class PlainIceCream implements IceCream{ //基础实现，无填加的冰激凌
    @Override
    public void AddTopping() {
        System.out.println("Plain IceCream ready for some
                           toppings!");
    }
}

/*装饰器基类*/
public abstract class ToppingDecorator implements IceCream{
    protected final IceCream input;
    public ToppingDecorator(IceCream i){
        this.input = i;
    }

    public abstract void AddTopping();  //留给具体装饰器实现
}
```

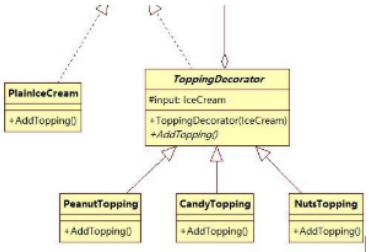
这里有一个特性就是将所有的add聚合到一个函数之中，所以在基类中定义了一个抽象方法

必须注重顺序




```
public class CandyTopping extends ToppingDecorator{
    public CandyTopping(IceCream i) {
        super(i);
    }

    public void AddTopping() {
        input.AddTopping();  //decorate others first
        System.out.println("Candy Topping added! ");
    }
}

public class NutsTopping extends ToppingDecorator{
    //similar to CandyTopping
}

public class PeanutTopping extends ToppingDecorator{
   //similar to CandyTopping

}
```

```
public class Client {
    public static void main(String[] args) {
        IceCream a = new PlainIceCream();
        IceCream b = new CandyTopping(a);
        IceCream c = new PeanutTopping(b);
        IceCream d = new NutsTopping(c);
        d.AddTopping();

       //or
       IceCream toppingIceCream =
           new NutsTopping(
               new PeanutTopping(
                   new CandyTopping(
                        new PlainIceCream()
                       )
                   )
               );
        toppingIceCream.AddTopping();
}
```




```
The result:
Plain IceCream ready for some toppings
Candy Topping added!
Peanut Topping added!
Nuts Topping added!
```

Façade（外观模式）

具体实现

场景：

客户端 client 需要与 A，B，C……等多个类进行交互，这些类具有高耦合度。

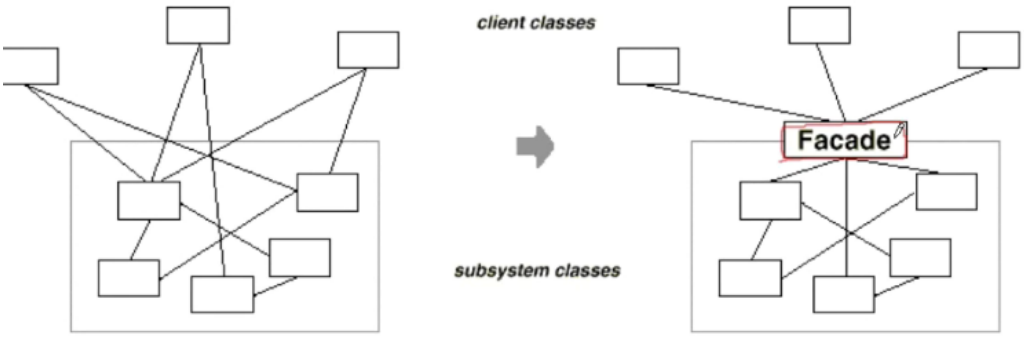
实现：

可以定义一个类 S，聚合 clinet 与 A，B，C……交互通道。最终 client 与 S 交互，就可以完成原先的功能。

优点：

A. 可以简化客户端的学习任务量
B. 可以解耦客户端与这些功能类，内部子系统的任何变化不会影响到 Facade 接口的变化。注意这里的耦合需要与内部类的耦合程度进行区分。

实现形象图：



Ex：

- Suppose we have an application with set of interfaces to use **MySql/Oracle** database and to generate different types of reports, such as **HTML report**, **PDF report** etc.
- So we will have different set of interfaces to work with different types of database.
- Now a client application can use these interfaces to get the required database connection and generate reports.
- But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it.
- So we can apply **Facade pattern** here and provide a **wrapper interface** on top of the existing interface to help client application.

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){…}
}
```

```
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;}
}
```

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);
```

4.3.2 Behavioral patterns

Strategy（策略模式）

问题：

有多种不同的算法 ConcreteStrategyA，ConcreteStrategyB，ConcreteStrategyC……来实

现同一个任务 context，但需要 client 根据需要动态切换算法，而不是写死在代码里。

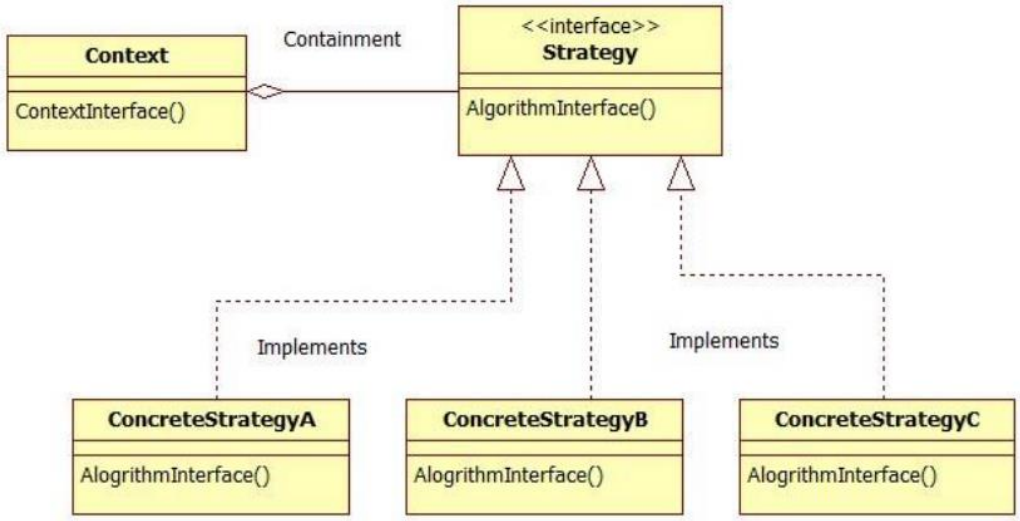
实现方案：

将不同的实现算法抽象，构造抽象接口 strategy，使得这些算法均实现了此接口。在 client 端中，实现该算法的方法需要传入一个以 strategy 定义的对象，并将其功能委托给此对象。

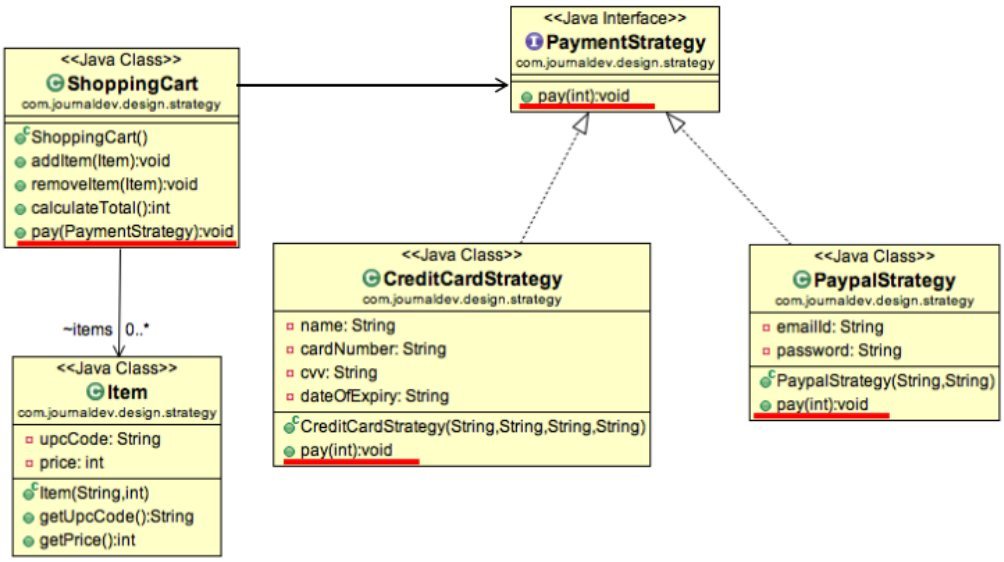
优点：

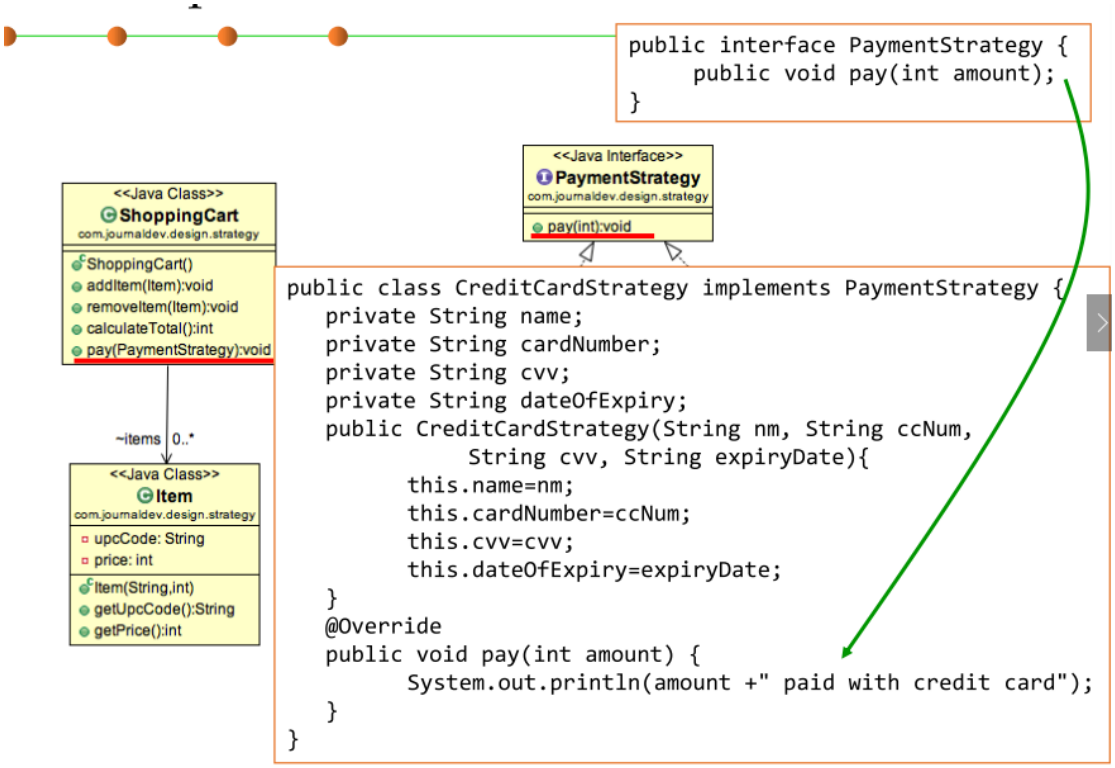
隔离了算法具体实现与客户端，易于扩展。

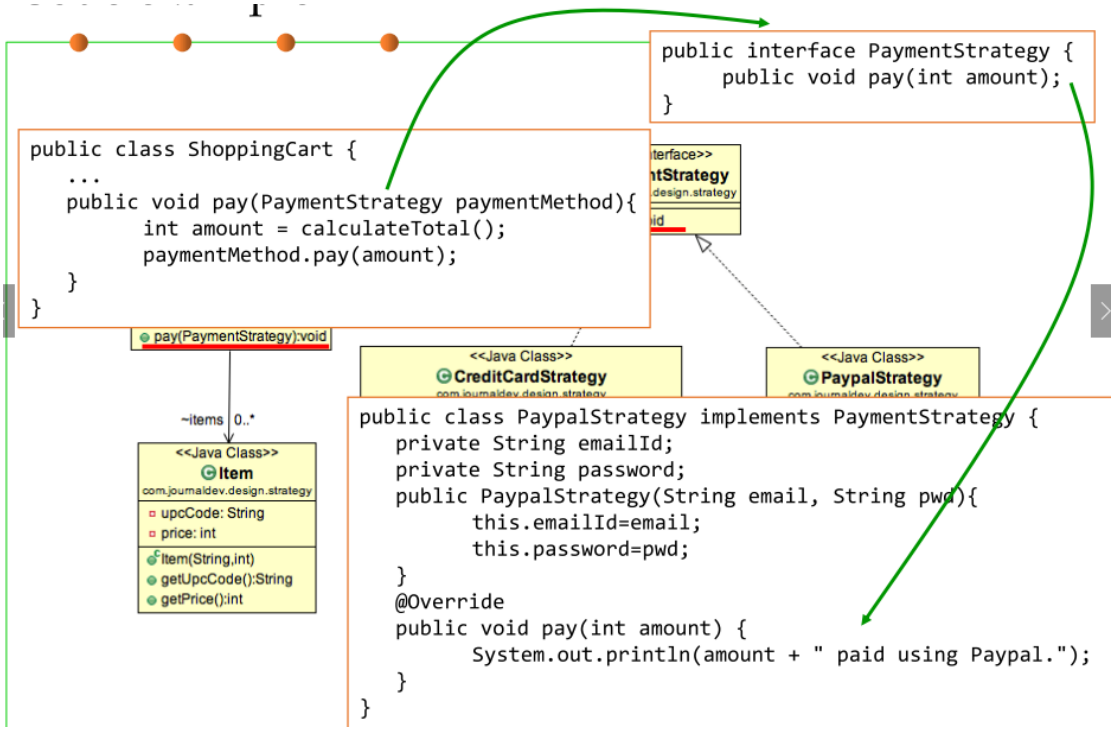
形象图：



Ex：







Template Method（模板模式）

问题：

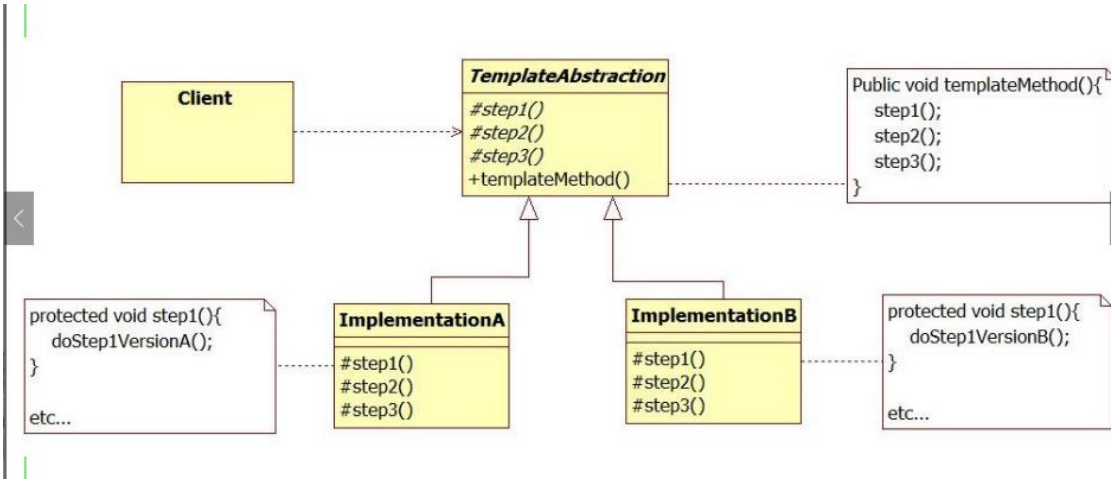
多个相关类 ImplementationA，ImplementationB，ImplementationC……中有着相同意义的方法 step1(),step2(),step3()……但具体实现不同，且客户端调用这些方法的顺序相同，如何采用一种有效的组织方式，进行程序设计？

实现办法：

定义这些相关类 ImplementationA，ImplementationB，ImplementationC……的一个抽象类 TemplateAbstraction，（TemplateAbstraction 与这些相关类存在子类型关系）使得此超类中包含方法 step1(),step2(),step3()……，并定义一个额外的方法 TemplateMethods，其中按照需要的顺序调用 step1(),step2(),step3()……，正如客户端使用的那样。

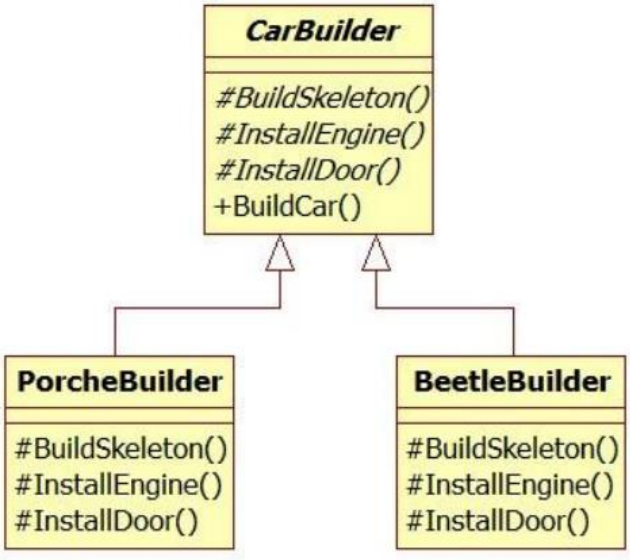
最终，在客户端中，定义 TemplateAbstraction 声明的变量，并且以其具体实现进行初始化，客户端只需要调用 TemplateMethods 方法即可。

形象图：



注：此策略正式白盒框架的实现方式。而黑盒框架虽然严格规定了步骤，但是其却并未能够符合这种结构，没有使用子类型关系，而是使用接口的方法来实现。

Ex：



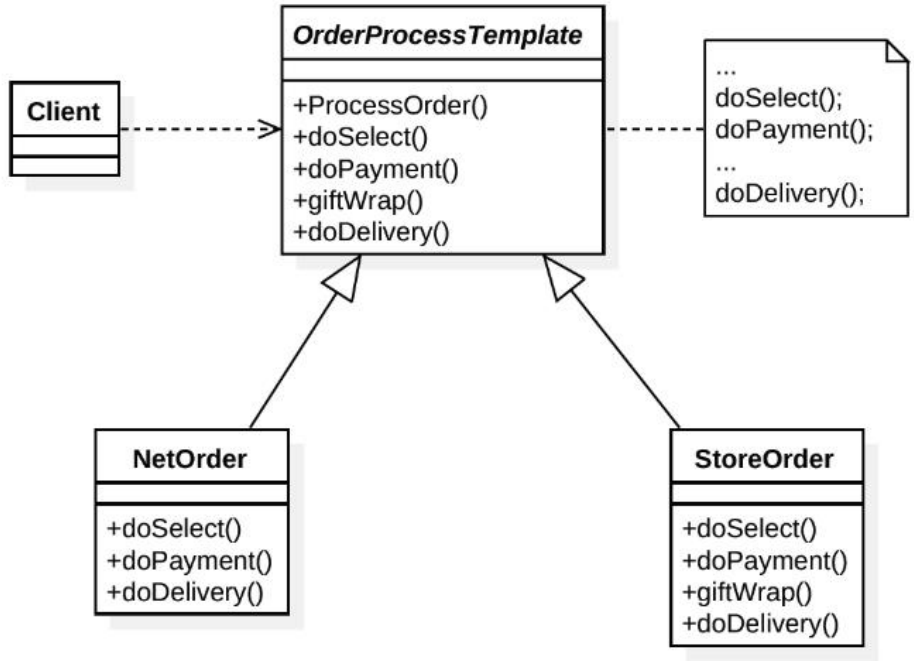
```
public abstract class CarBuilder {
        protected abstract void BuildSkeleton();
        protected abstract void InstallEngine();
        protected abstract void InstallDoor();

        // Template Method that specifies the general logic
        public void BuildCar() {  //通用逻辑
                BuildSkeleton();
                InstallEngine();
                InstallDoor();
        }
}
```

```
public class PorcheBuilder extends CarBuilder {
         protected void BuildSkeleton() {
                 System.out.println("Building Porche Skeleton");
         }
         protected void InstallEngine() {
                 System.out.println("Installing Porche Engine");
         }
         protected void InstallDoor() {
                 System.out.println("Installing Porche Door");
         }
}


public class BeetleBuilder extends CarBuilder {
         protected void BuildSkeleton() {
                 System.out.println("Building Beetle Skeleton");
         }
         protected void InstallEngine() {
                 System.out.println("Installing Beetle Engine");
         }
         protected void InstallDoor() {
                 System.out.println("Installing Beetle Door");
         }
}
```

```
public static void main(String[] args) {
        CarBuilder c = new PorcheBuilder();
        c.BuildCar();

        c = new BeetleBuilder();
        c.BuildCar();
        }
```

```
Building Porche Skeleton
Installing Porche Engine
Installing Porche Door
Building Beetle Skeleton
Installing Beetle Engine
Installing Beetle Door
```

Ex:



```
public abstract class OrderProcessTemplate {
  public boolean isGift;

  public abstract void doSelect();
  public abstract void doPayment();
  public final void giftWrap() {
        System.out.println("Gift wrap done.");
  }
  public abstract void doDelivery();
  public final void processOrder() {
        doSelect();
        doPayment();
        if (isGift)
            giftWrap();
        doDelivery();
  }
}
```

```
OrderProcessTemplate netOrder = new NetOrder();
netOrder.processOrder();

OrderProcessTemplate storeOrder = new StoreOrder();
storeOrder.processOrder();
```

```
public class NetOrder
        extends OrderProcessTemplate {

  @Override
  public void doSelect() { … }

  @Override
  public void doPayment() { … }

  @Override
  public void doDelivery() { … }
}
```

Iterator（迭代器）

问题：

在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望在不暴露其内部结构的同时，可以让外部客户代码访问其中包含的元素；同时这种“透明遍历”也为“同一种算法在多种集合对象上进行操作”提供了可能。

具体实现

定义一个接口 Aggregate/Iterable，此接口的作用是返回一个迭代器，所返回的迭代器的类型同样是一个接口 Iterator，其内规定了返回一个迭代器所需要的方法：

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

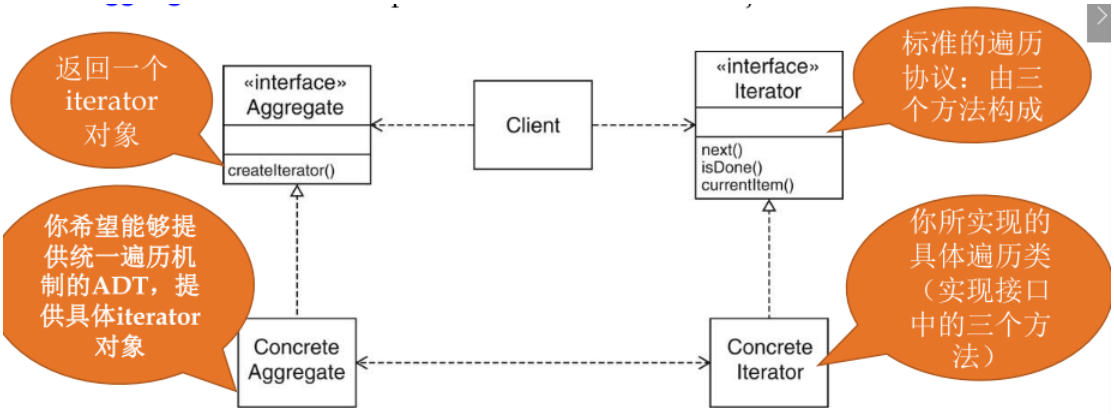
而 Iterator 接口的具体实现为：

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

当某个对象需要进行迭代时， 该对象首先实现接口 Aggregate/Iterable，实现该接口需要返回一个适用于该对象的迭代器，该迭代器实现接口 Iterator。

当客户端进行调用时，可以先由该对象的 Iterator<T> iterator()方法，返回一个迭代器，之后对其进行操作。

形象图：



优点：

该设计模式首先实现了问题中的信息隐藏

其迭代器采用接口的类型定义，避免了信息泄露。

支持多种多样的遍历策略，且善于改变

第一个接口就是单纯的使得该对象拥有那个返回迭代器的方法，该方法的返回类型是一个接口，这个接口隔离了对象以及迭代器的具体实现，故而可以在实现这个接口的前提下，对具体实现进行修改。

Ex:

```
public interface Collection<E> extends Iterable<E> {
   boolean add(E e);
   boolean addAll(Collection<? extends E> c);
   boolean remove(Object e);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
   boolean contains(Object e);
   boolean containsAll(Collection<?> c);
   void clear();
   int size();
   boolean isEmpty();
   Iterator<E> iterator();
   Object[] toArray()
   <T> T[] toArray(T[] a);
   …
}
```

Defines an interface for creating an Iterator, but allows `Collection` implementation to decide which `Iterator` to create.


```
public class Pair<E> implements Iterable<E> {
   private final E first, second;
   public Pair(E f, E s) { first = f; second = s; }
   public Iterator<E> iterator() {
      return new PairIterator();
   }

   private class PairIterator implements Iterator<E> {
      private boolean seenFirst = false, seenSecond = false;
      public boolean hasNext() { return !seenSecond; }
      public E next() {
         if (!seenFirst) { seenFirst = true; return first; }
         if (!seenSecond) { seenSecond = true; return second; }
             throw new NoSuchElementException();
      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
   }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```