11 Design Patterns for Reuse and Maintainability
面向可复用性和可维护性的设计模式

Wang Zhongjie
rainy@hit.edu.cn

June 21, 2021

Outline

- **Creational patterns**
  - Factory method pattern creates objects without specifying the exact class.
- **Structural patterns**
  - Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
  - Decorator dynamically adds/overrides behavior in a method of an object.
- **Behavioral patterns**
  - Strategy allows one of a family of algorithms to be selected at runtime.
  - Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
  - Iterator accesses the elements of an object sequentially without exposing its underlying representation.
  - Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Reading

- **CMU 17-214：Sep 12、Sep 17**
- 设计模式：第**1**、**2**章；第**4.1**、**4.4**、**4.5**、**5.4**、**5.9**、**5.10**节

- **CMU 17-214：Nov 26**
- 设计模式：第**3.1**、**3.2**、**3.3**、**4.2**、**4.3**、**4.7**、**5.5**、**5.7**、**5.11**、**(5.1)**、**(5.2)**节

Why reusable design patterns?

A design…

…enables flexibility to change (reusability)

…minimizes the introduction of new problems when fixing old ones (maintainability)

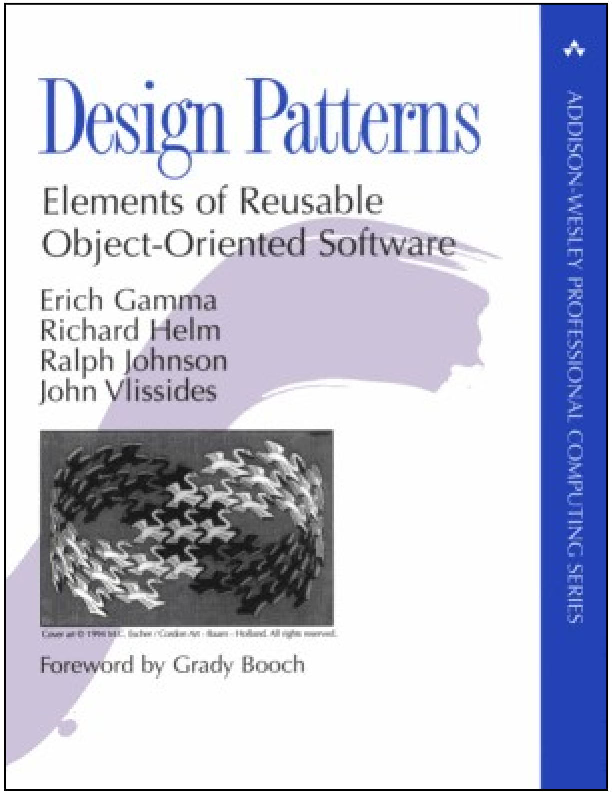
…allows the delivery of more functionality after an initial delivery (extensibility).

Design Patterns: a general, reusable solution to a commonly occurring problem within a given context in software design.

OO design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. 除了类本身，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

Gang of Four

- **Design Patterns: Elements of Reusable Object-Oriented Software**
- **By GoF (Gang of Four)**
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design patterns taxonomy

- **Creational patterns** 创建型模式
  - Concern the process of object creation
- **Structural patterns** 结构型模式
  - Deal with the composition of classes or objects
- **Behavioral patterns** 行为类模式
  - Characterize the ways in which classes or objects interact and distribute responsibility.

1 Creational patterns

Factory Method pattern

工厂方法模式

Factory Method

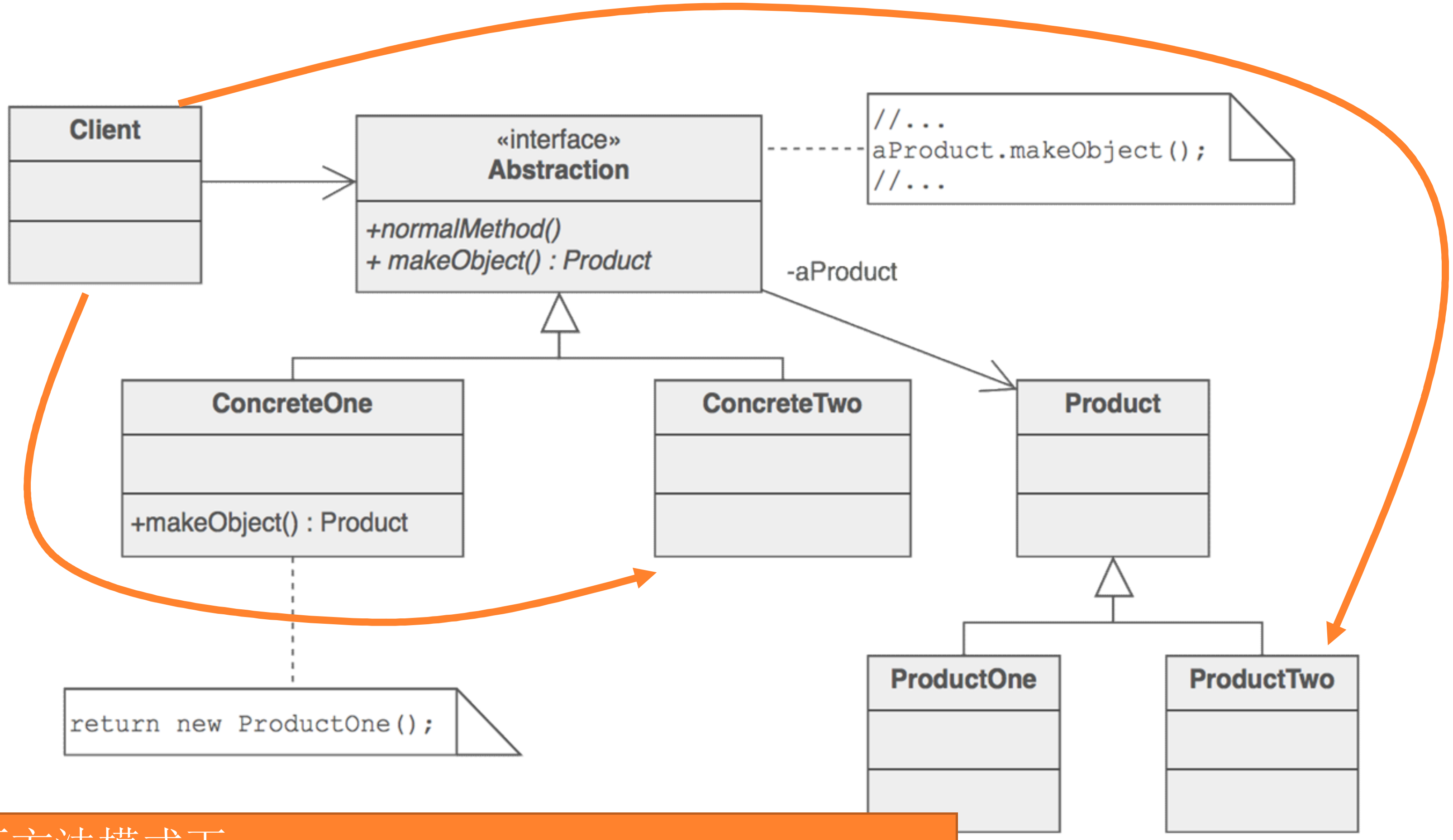
- **Also known as "Virtual Constructor"** 虚拟构造器
- **Intent:**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. *为创建对象定义一个接口，让子类决定初始化哪个类*
  - Factory Method lets a class defer instantiation to subclasses. *工厂化模式将类初始化 延伸至子类。*
- **When should we use Factory Method? ---- When a class:**
  - Can't predict the class of the objects it needs to create
  - Wants its subclasses to specify the objects that it creates
  - Delegates responsibility to one of multiple helper subclasses, and you need to localize the knowledge of which helper is the delegate.

当client不知道要创建哪个具体类的实例，或者不想在client代码中指明要具体创建的实例时，用工厂方法。

定义一个用于创建对象的接口，让其子类来决定实例化哪一个类，从而使一个类的实例化延迟到其子类。

Factory Method

常规情况下，client直接创建具体对象
Product p = new ProductTwo();



在工厂方法模式下：
Product p = new ConcreteTwo().makeObject();

Example

Abstract product

```
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

Concrete product 1

```
public class FileTrace implements Trace {
      private PrintWriter pw;
      private boolean debug;
      public FileTrace() throws IOException {
         pw = new PrintWriter( new FileWriter( "t.log" ) );
      }
      public void setDebug( boolean debug ) {
         this.debug = debug;
      }
      public void debug( String message ) {
        if( debug ) {
              pw.println( "DEBUG: " + message );
              pw.flush();
        }
      }
      public void error( String message ) {
        pw.println( "ERROR: " + message );
        pw.flush();
      }
}
```

Example

Abstract product

```
public interface Trace {
      // turn on and off debugging
      public void setDebug( boolean debug );
      // write out a debug message
      public void debug( String message );
      // write out an error message
      public void error( String message );
}
```

Concrete product 2

```
public class SystemTrace implements Trace {
      private boolean debug;
      public void setDebug( boolean debug ) {
        this.debug = debug;
      }
      public void debug( String message ) {
        if( debug )
           System.out.println( "DEBUG: " + message );
      }
      public void error( String message ) {
        System.out.println( "ERROR: " + message );
      }
}
```

How to use?

```
//... some code ...
Trace log = new SystemTrace();
log.debug( "entering log" );

Trace log2 = new FileTrace();
log.debug(“...”);
```

The client code is tightly coupled with concrete products.

Client需要在程序中直接实例化具体实现类，不是面向接口编程。

Example

```
interface TraceFactory {
  public Trace getTrace();
  public Trace getTrace(String type);
  void otherOperation(){};
}
```

```
lic class Factory1 implements TraceFactory {
      public Trace getTrace() {
            return new SystemTrace();
      }
}

public class Factory2 implements TraceFactory {
      public getTrace(String type) {
          if(type.equals(“file”)
              return new FileTrace();
          else if (type.equals(“system”)
              return new SystemTrace();
      }
```

```
Trace log1 = new Factory1().getTrace();
log1.setDebug(true);
log1.debug( "entering log" );
Trace log2 = new Factory2().getTrace("system");
log2.setDebug(false);
log2.debug("...");
```

不仅包含factory method，还可以实现其他功能

有新的具体产品类加入时，可以在工厂类里修改或增加新的工厂函数(OCP)，不会影响客户端代码

根据类型决定创建哪个具体产品

Client使用“工厂方法”来创建实例，得到实例的类型是抽象接口而非具体类

Example

```
public class TraceFactory1 {
      public static Trace getTrace() {
            return new SystemTrace();
      }
}

public class TraceFactory2 {
      public static Trace getTrace(String type) {
         if(type.equals(“file”)
             return new FileTrace();
         else if (type.equals(“system”)
             return new SystemTrace();
      }
}
```

静态工厂方法

既可以在ADT内部实现，也可以构造单独的工厂类

```
//... some code ...
Trace log1 = TraceFactory1.getTrace();
log1.setDebug(true);
log1.debug( "entering log" );

Trace log2 = TraceFactory2.getTrace(“system”);
log1.setDebug(true);
log2.debug(“...”);
```

Factory Method

- **Advantage:**
  - Eliminates 排除 the need to bind application-specific classes to your code.
  - Code deals only with the `Product` interface (`Trace`), so it can work with any user-defined `ConcreteProduct` (`FileTrace`, `SystemTrace`)
- **Potential Disadvantages 潜在劣势**
  - Clients may have to make a subclass of the `Creator`, just so they can create a certain `ConcreteProduct`. *客户可能需要创建Creator的子类*
  - This would be acceptable if the client has to subclass the `Creator` anyway, but if not then the client has to deal with another point of evolution. *如果客户端必须对创建者进行子类化，那么这是可以接受的，但是如果不是这样，那么客户端必须处理另一个进化点。*

- **Open-Closed Principle (OCP)**
  ——对扩展的开放，对修改已有代码的封闭

2 Structural patterns

(1) Adapter

加个“适配器”以便于复用

Adapter模式的意图：

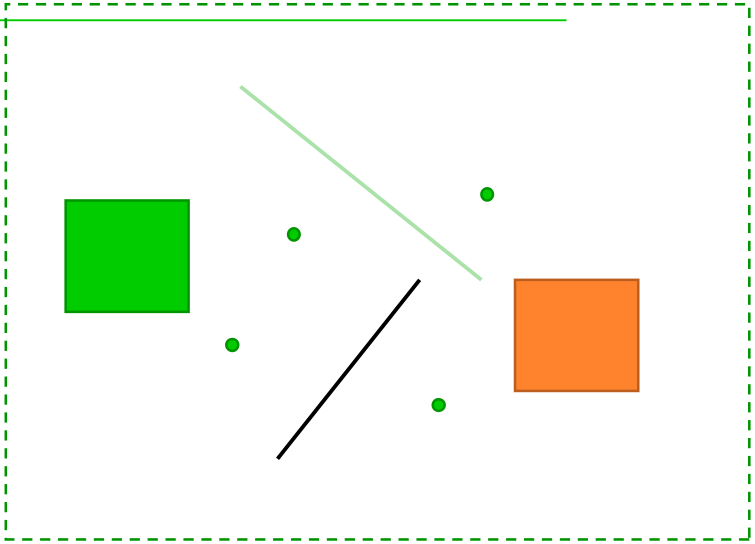
- 将一个类的接口转换成客户希望的另外一个接口。
- **Adapter**模式使原本由于接口不兼容而不能一起工作的类可以一起工作。
- 适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

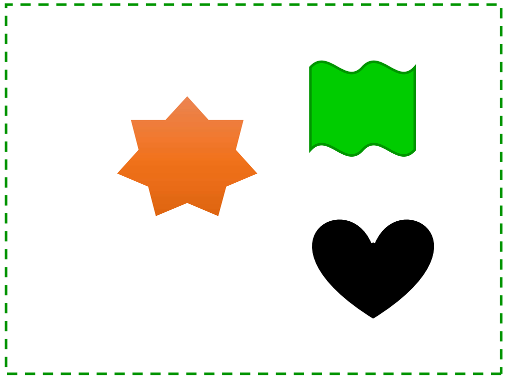
Adapter模式的优缺点

- 该模式的主要优点如下：
  - 客户端通过适配器可以透明地调用目标接口。
  - 复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类。
  - 将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。
- 其缺点是：
  - 对类适配器来说，更换适配器的实现过程比较复杂。

一个引导性的例子：

- 假设客户有如下需求：
  - 为有显示（display）行为的点、线、正方形分别创建类；
  - 客户对象不必知道到底自己拥有点、线还是正方形，他们只需要知道拥有这些形状的中的一种即可。
- 设计一个更高层次的概念，将这些形状都涵盖进去，称之为“可显示的形状”。
- 我们希望客户只关注于形状，而不关注于形状之间的区别。
- 好处：我们可以增/减任意形状，从而适应用户的变化

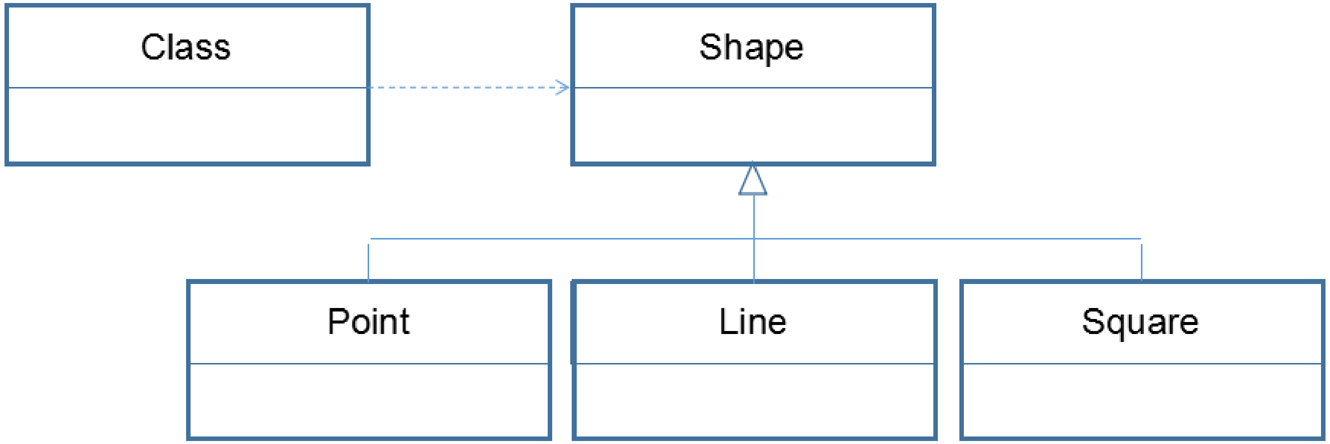




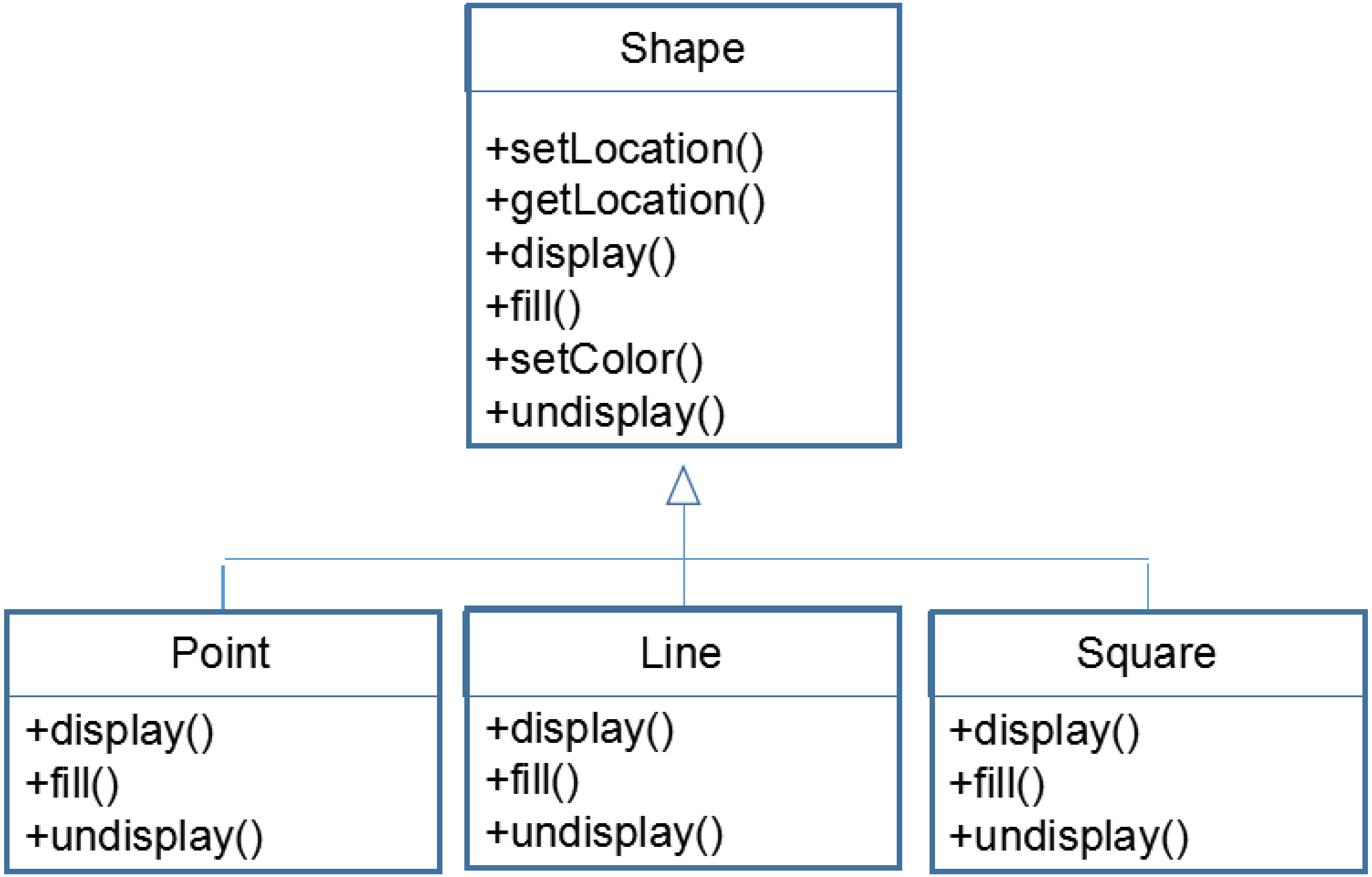
一个引导性的例子：

- 客户只是要实现：
  - 自我显示
  - 自我擦除
  - 填充
  - 设置颜色
  - ……

- 我们想：多态地使用派生类



一个引导性的例子：


Shape
+setLocation()
+getLocation()
+display()
+fill()
+setColor()
+undisplay()
Point
+display()
+fill()
+undisplay()
Line
+display()
+fill()
+undisplay()
Square
+display()
+fill()
+undisplay()

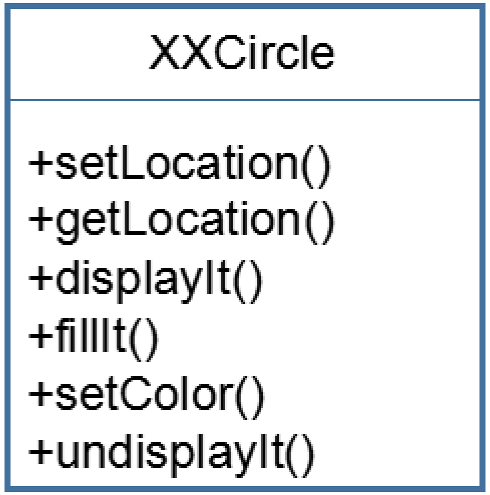
一个引导性的例子：

- 现在假设用户提出了新的要求：
  - 增加一个圆形显示
- 一个新的需求（需求在变）
- 但我们之前的设计显然可以适应这个变化。

- 我们有三个解决方案：
  - 按照我们之前的设计添加一个派生类Circle，重新实现其中的方法即可；
  - 看看其他的人是否有现成的类，根据我们已有的设计改造“它”，适应我们的设计；
  - 看看其他的人是否有现成的类，在不改变他人现成程序的基础上，想办法和我们已有的设计接口。

一个引导性的例子：

- 很幸运，我们找到了一个已经实现好了的类，但它是如下实现的：



XXCircle

+setLocation()
+getLocation()
+displayIt()
+fillIt()
+setColor()
+undisplayIt()

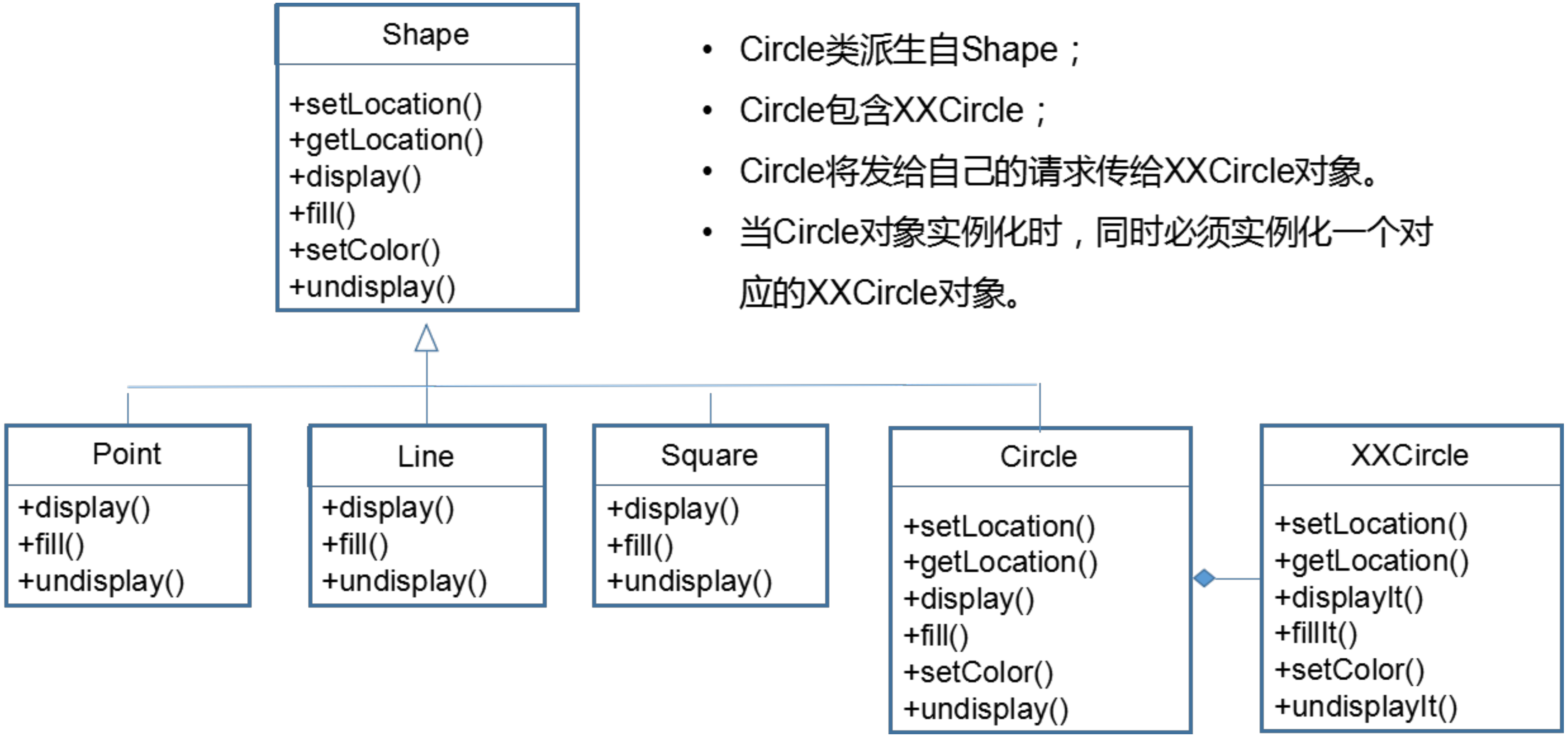
- 存在不同：
  - 名称和参数列表与Shape类不同；
  - 无法派生它。
- 解决方案：
  - 要求对方按照我的设计改（难为人）；
  - 我自己改（容易引入Bug）；
  - ？

Java代码片段：实现Adapter模式

```
class Circle extends Shape{
    ....
    private XXCircle myCircle;
    public Circle(){
        myXXCircle=new XXCircle();
    }
   void public display(){
       myXXCircle.displayIt();
   }
   ...
}
```

- 复用了他人代码；
- 没有改变自我的设计，保持了多态性；
- 对用户而言形式统一；

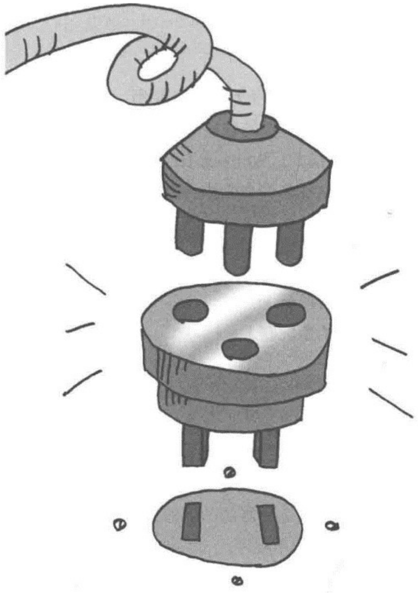
引用Adapter模式



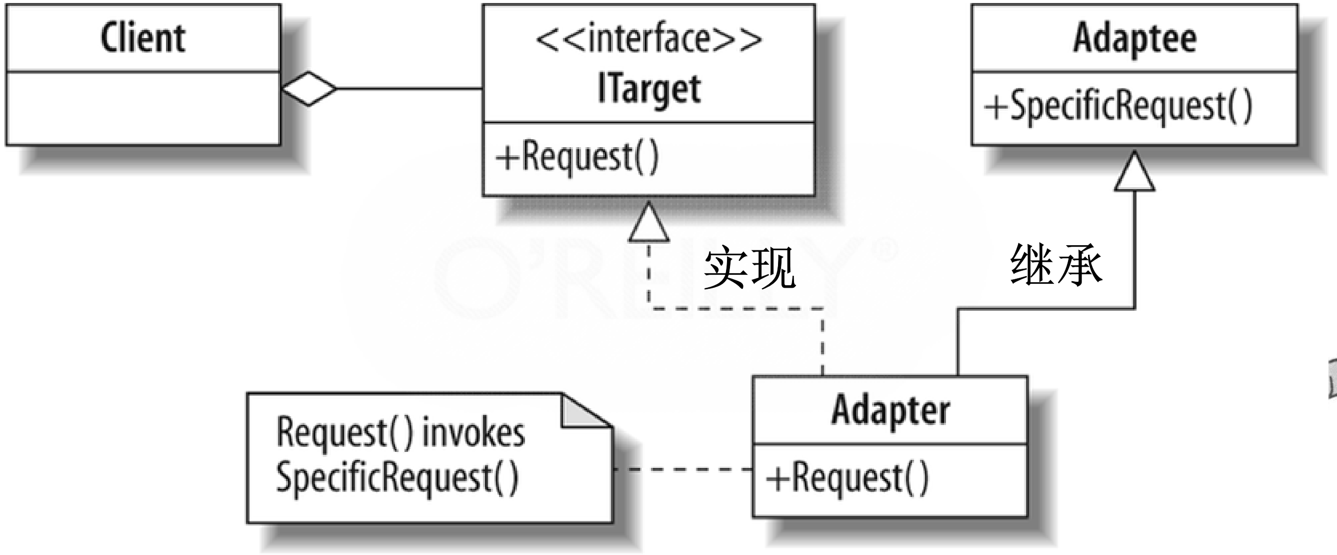
- Circle类派生自Shape；
- Circle包含XXCircle；
- Circle将发给自己的请求传给XXCircle对象。
- 当Circle对象实例化时，同时必须实例化一个对应的XXCircle对象。

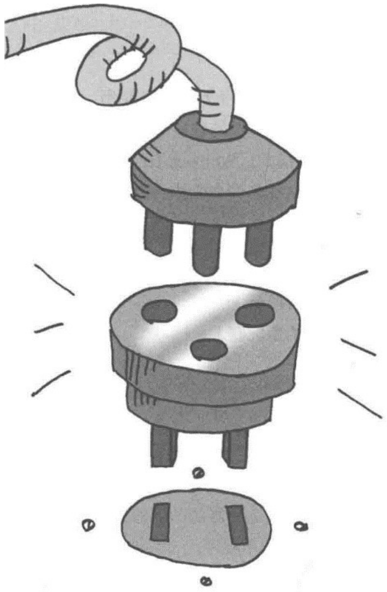
Adapter Pattern 适配器模式

- **Intent: Convert the interface of a class into another interface clients expect.** 意图：将类的接口转换为客户端期望的另一个接口
  - Adapter lets classes work together that couldn‘t otherwise because of incompatible interfaces. 解决类之间接口不兼容的问题
  - Wrap an existing class with a new interface. 为已有的类提供新的接口
- **Objective: to reuse an old component to a new system (also called “wrapper”)** 目标：对旧的不兼容组件进行包装，在新系统中使用旧的组件
- 加个“适配器”以便于复用



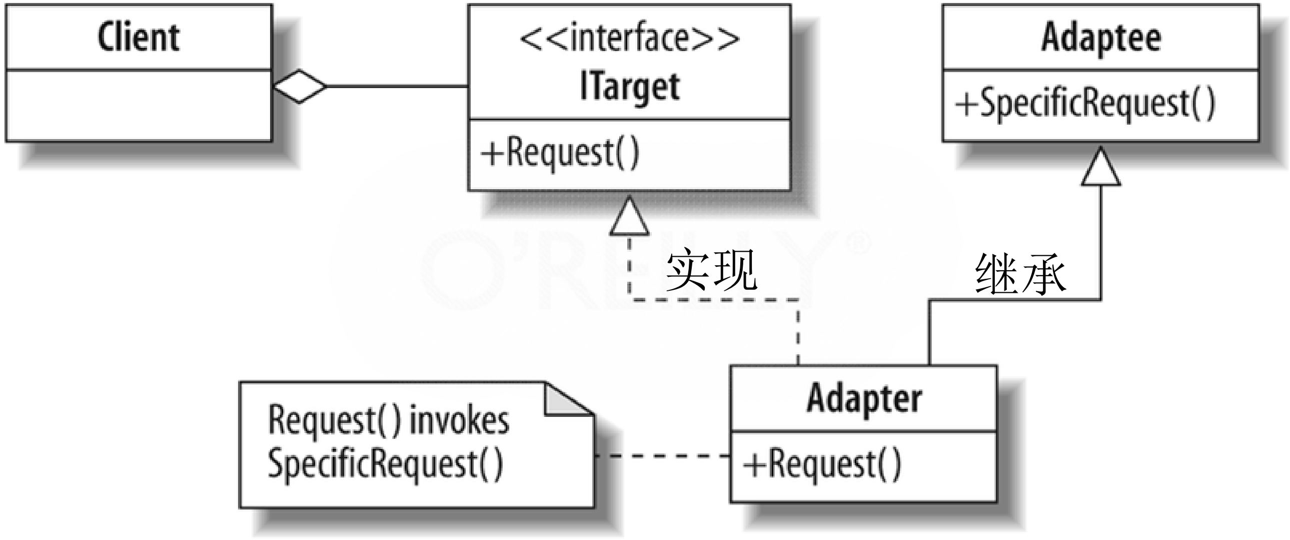
Adapter Pattern 适配器模式





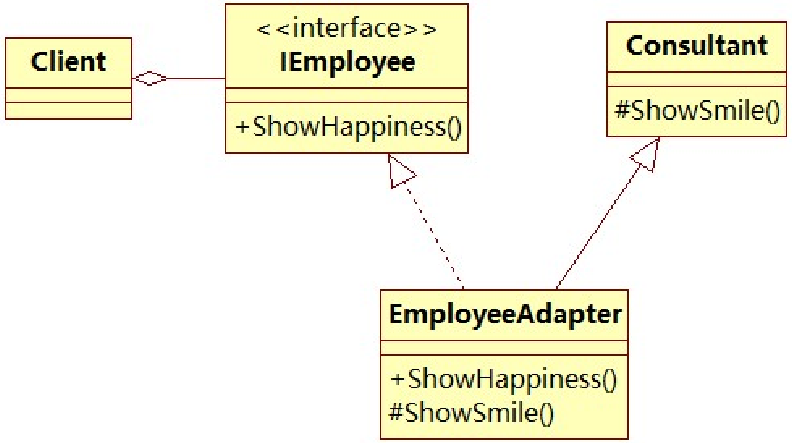
Example

- The ***Adaptee*** is the existing class.
- The ***ITarget*** is the interface defined in the existing library.
- The ***Adapter*** is the class that you create, it is inherited from the adaptee class and it implements the ITarget interface. Notice that it can call the SpecificRequest method(inherited from the adaptee) inside its request method(implemented by the ITarget).



Example

- An organization tree that is constructed where all the employees implements the *IEmployee* interface. The IEmployee interface has a method named ShowHappiness().
- We need to plug an existing *Consultant* class into the organization tree. The *Consultant* class is the adaptee which has a method named ShowSmile().
- **This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.** 通过增加额外的间接层来解决不协调/不兼容的问题



Example

```
public class Consultant { //已存在的类
    private String name;

    public Consultant(String name) {
        this.name = name;
    }

    protected void ShowSmile() {
        System.out.println("Consultant " + this.name + " showed
                                  smile");
    }
}

public interface IEmployee {//目标接口
        void ShowHappiness();
}
```

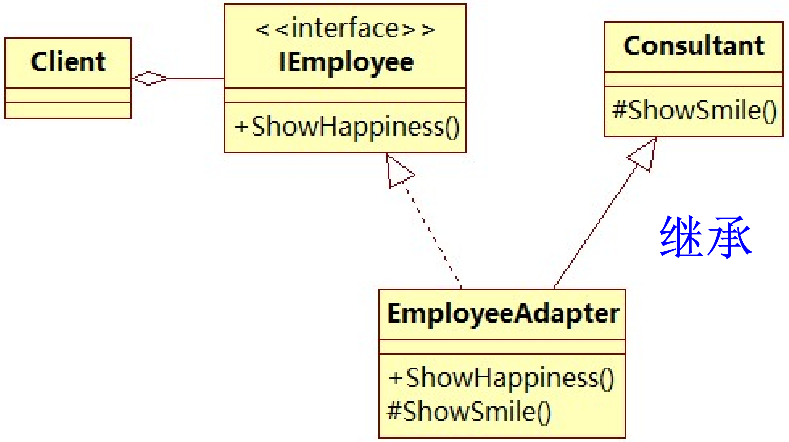
Example

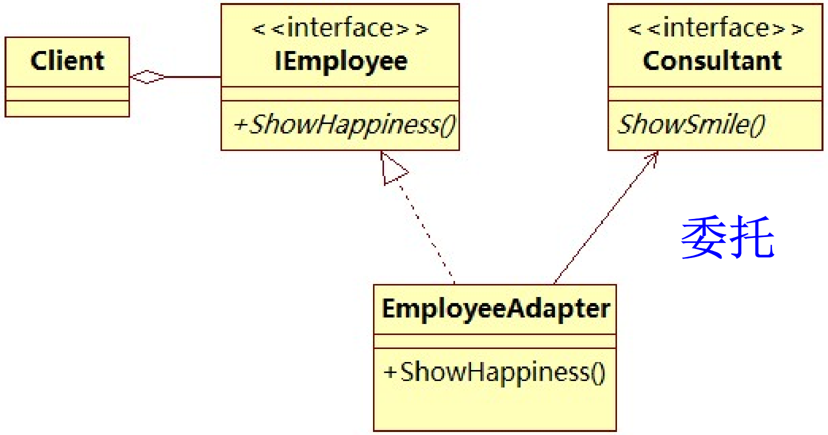
```
public class EmployeeAdapter extends Consultant implements
IEmployee {  //Adapter
    public EmployeeAdapter(String name){
       super(name);
    }

    @Override
    public void ShowHappiness() {
        ShowSmile(); // call the parent Consultant class
    }
}

public class Client {
    public static void main(String[] args){
    IEmployee em = new EmployeeAdapter("Bruno");
        em.ShowHappiness();
    }
}
```

Result: Consultant Bruno showed smile

Adapter Pattern

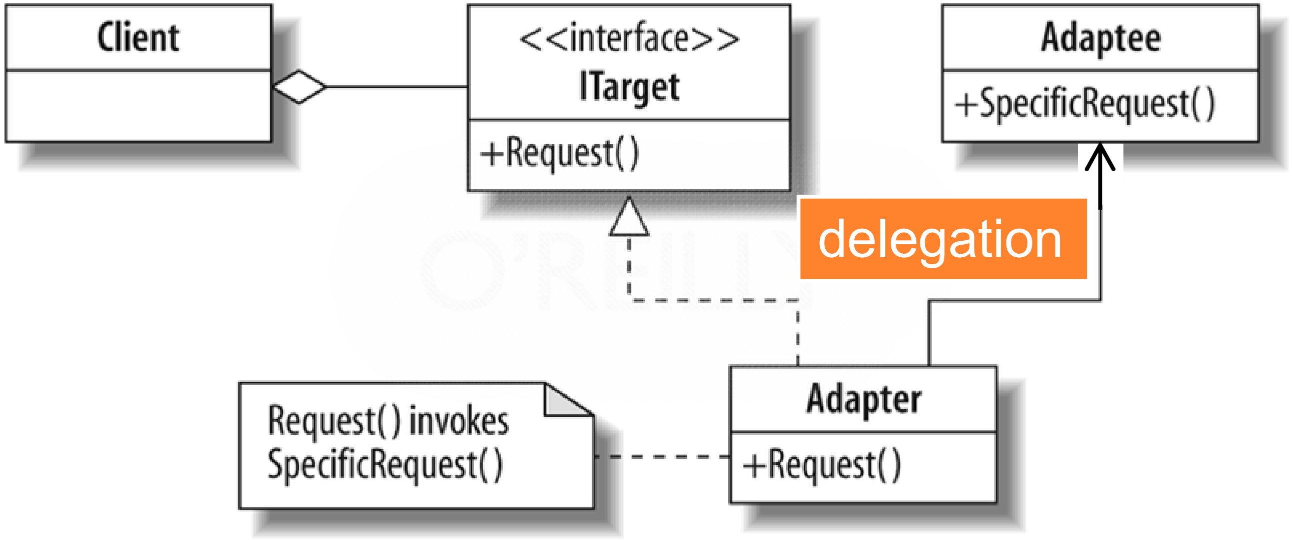
- **Two types of Adapter Design Pattern**
  - Inheritance
  - Delegation





Adapter Pattern

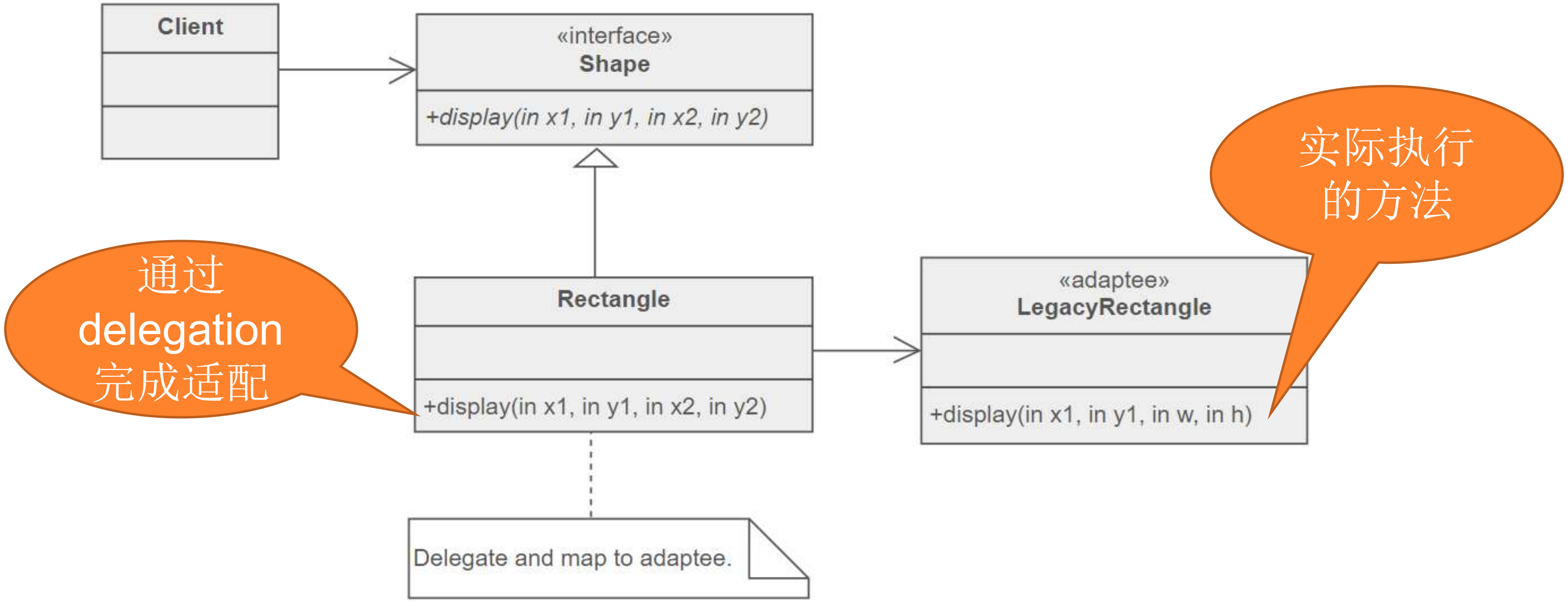
- **Intent: Convert the interface of a class into another interface that clients expect to get.** **将某个类/接口转换为client期望的其他形式**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Adapter是将两种不同的接口整合在一起。
  - Wrap an existing class with a new interface. 通过增加一个接口，将已存在的子类封装起来，client面向接口编程，从而隐藏了具体子类。
- **Object: to reuse an old component to a new system (also called** “wrapper”)



Example

Client里的调用代码怎么写？

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


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.
通过 delegation 完成适配
实际执行的方法

Example: without Adaptor pattern

```
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!

Example: with Adaptor pattern

```
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);
  }
}
```

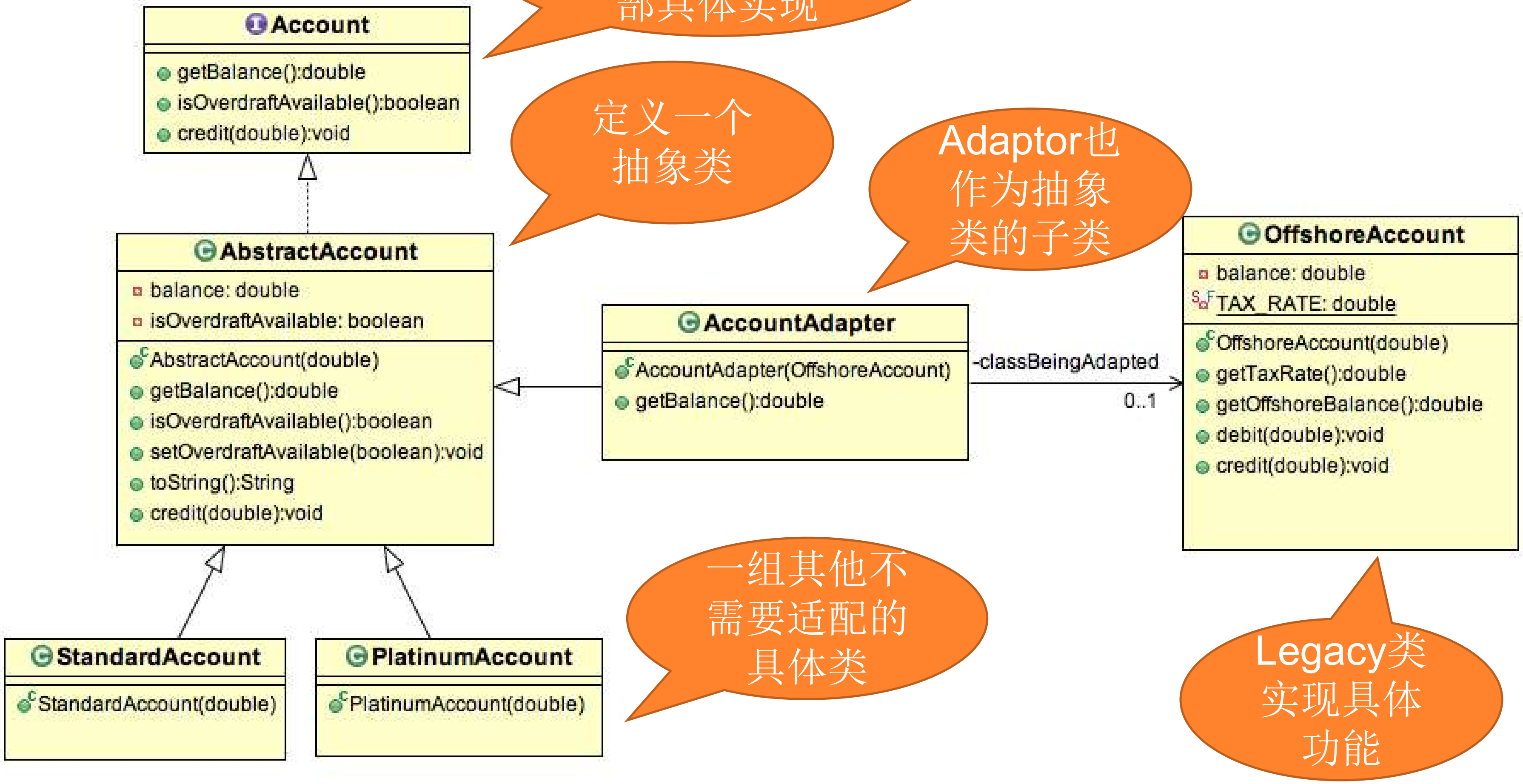
Adaptor类实现抽象接口

具体实现方法的适配

对抽象接口编程，与LegacyRectangle隔离

Example

Question: How does a client use this `OffshoreAccount`?



定义一个接口，隔离client与内部具体实现

定义一个抽象类

Adaptor也作为抽象类的子类

一组其他不需要适配的具体类

Legacy类实现具体功能

(2) Decorator

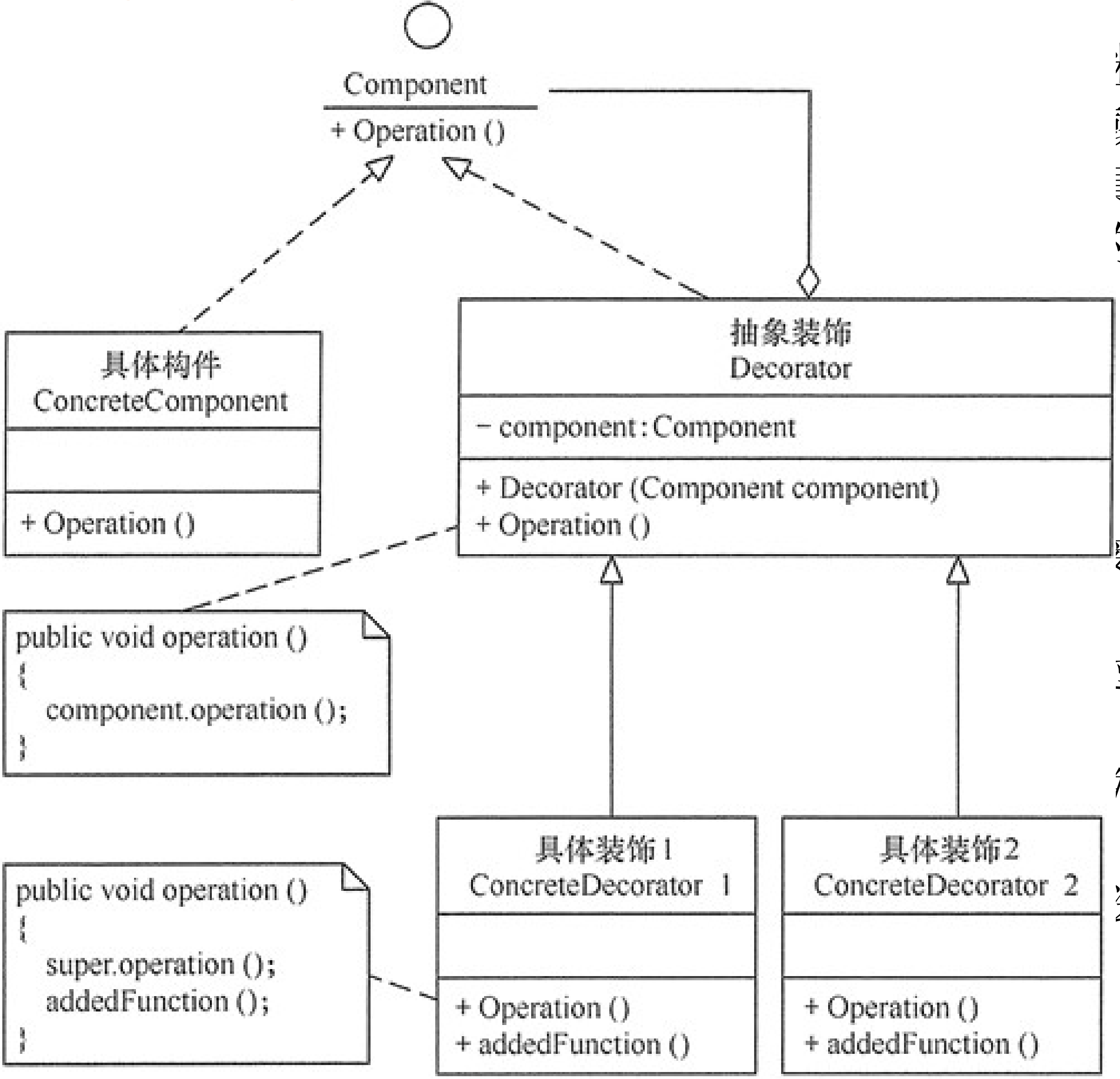
装饰边框与被装饰物的一致性

装饰模式的定义与特点

- 装饰（**Decorator**）模式的定义：
  - 指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。
- 装饰（**Decorator**）模式的主要优点有：
  - 采用装饰模式扩展对象的功能比采用继承方式更加灵活。
  - 可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。
- 其主要缺点是：
  - 装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

装饰模式的结构与实现

- 通常情况 承具有 静态特征 胀。如 果使用组 实对象 ，并在保 功能， 这就是装
- **1.** 模式的
  - 装饰模
    - 抽象构 *收附加责 任的对*
    - 具体构 *过装饰角色 为其添*
    - 抽象装 *的实例，可 以通过*
    - 具体装 *法，并给具 体构件*



```
public void operation ()
{
   component.operation ();
}
```

```
public void operation ()
{
   super.operation ();
   addedFunction ();
}
```

2. 模式的实现

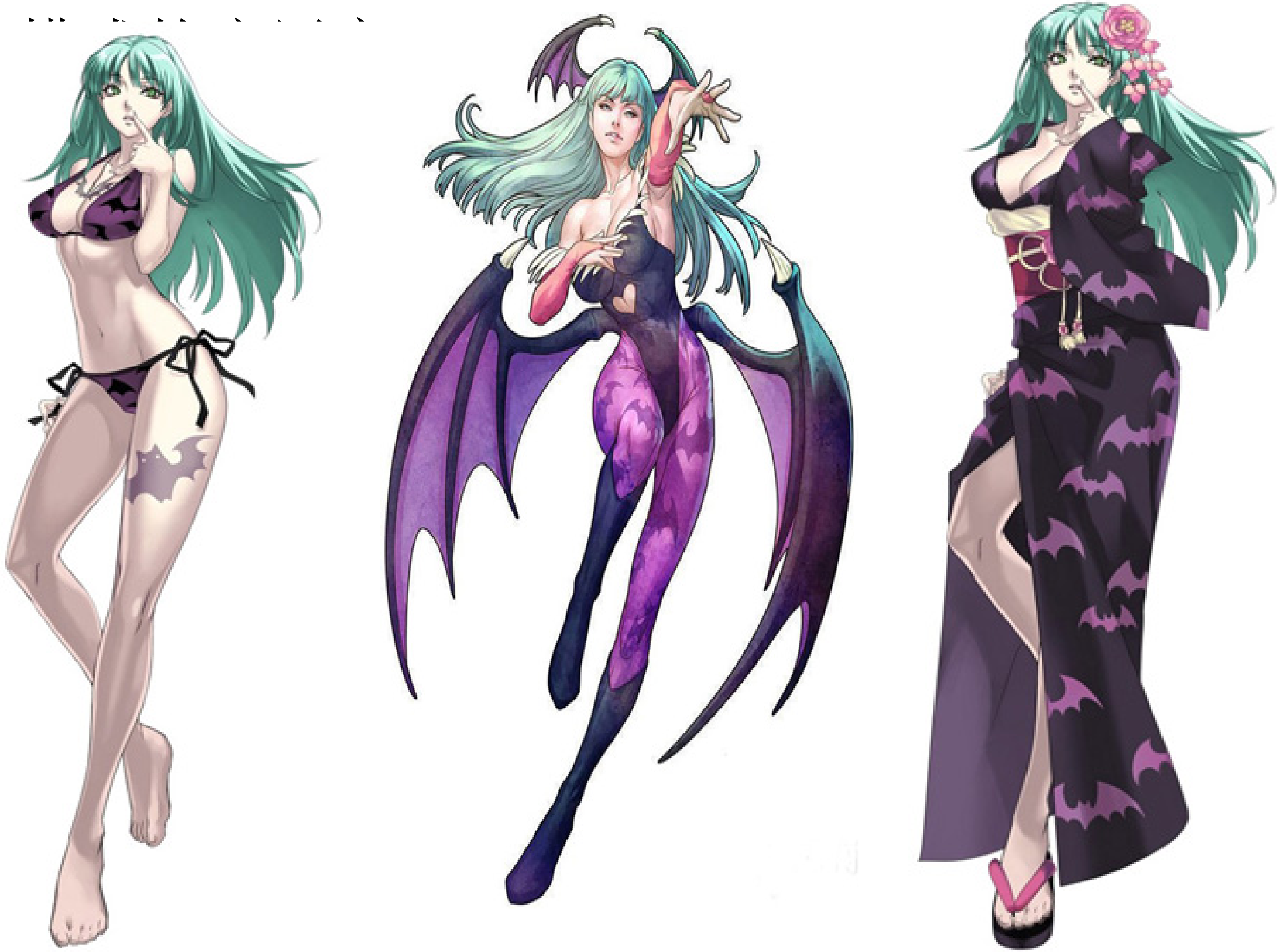
- 装饰模式的实现代码如下：

```
package decorator;
public class DecoratorPattern
{
   public static void main(String[] args)
   {
      Component p=new ConcreteComponent();
      p.operation();
      System.out.println("---------------------------------");
      Component d=new ConcreteDecorator(p);
      d.operation();
   }
}
//抽象构件角色
interface  Component
{
   public void operation();
}
//具体构件角色
class ConcreteComponent implements Component
{
   public ConcreteComponent()
   {
      System.out.println("创建具体构件角色");
   }
   public void operation()
   {
      System.out.println("调用具体构件角色的方法operation()");
   }
}
```

```
//抽象装饰角色
class Decorator implements Component
{
   private Component component;
   public Decorator(Component component)
   {
      this.component=component;
   }
   public void operation()
   {
      component.operation();
   }
}
//具体装饰角色
class ConcreteDecorator extends Decorator
{
   public ConcreteDecorator(Component component)
   {
      super(component);
   }
   public void operation()
   {
      super.operation();
      addedFunction();
   }
   public void addedFunction()
   {
      System.out.println("为具体构件角色增加额外的功能addedFunction()");
   }
}
```

装饰模式的应用场景

- 当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。
- 当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现。
- 当对象的功能要求可以动态地添加，也可以再动态地撤销时。
- 装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。



Motivating example of Decorator pattern

- **Suppose you want various extensions of a `Stack` data structure…**
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack` 同步堆栈: A stack that serializes concur
    accesses

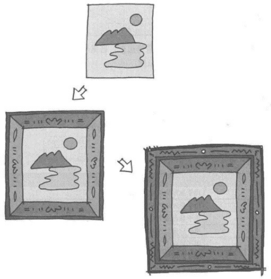
Inheritance

- **And arbitrarily composable extensions:**
  - `SecureUndoStack`: A stack that requires a password, and also lets you undo previous operations
  - `SynchronizedUndoStack`: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - `SecureSynchronizedStack`: …
  - `SecureSynchronizedUndoStack`: …

Inheritance hierarchies?
Multi-Inheritance?

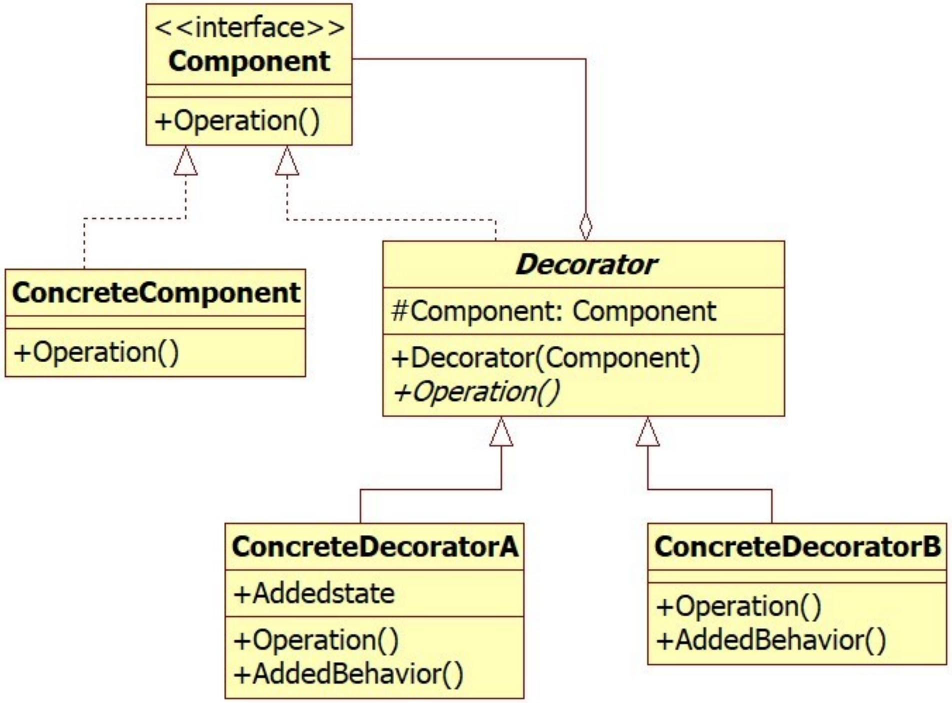
Decorator 装饰器模式

- **Problem: You need arbitrary or dynamically composable extensions to individual objects.** 问题: 需要对对象进行任意或者动态的扩展组合
- **Solution: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.** 方案: 实现一个通用接口作为要扩展的对象，将主要功能委托给基础对象**(stack)**，然后添加功能**(undo,secure,..)**。
- **It works in a recursive way.** 以递归的方式实现
- **Consequences:**
  - More flexible than static inheritance Customizable, cohesive extensions
- **Decorators use both subtyping and delegation**
- 装饰边框与被装饰物的一致性



Decorator

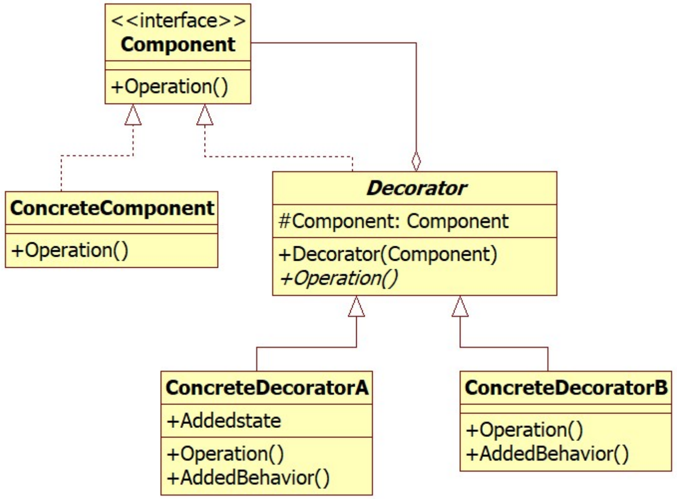
- The ***Component*** interface defines the operation, or the features that the decorators can perform. 接口：定义装饰物执行的公共操作
- The ***ConcreteComponent*** class is the **starting object** that you can dynamically add features to. You will create this object first and add features to it. 起始对象，在其基础上增加功能(装饰)，将通用的方法放到此对象中。



Decorator

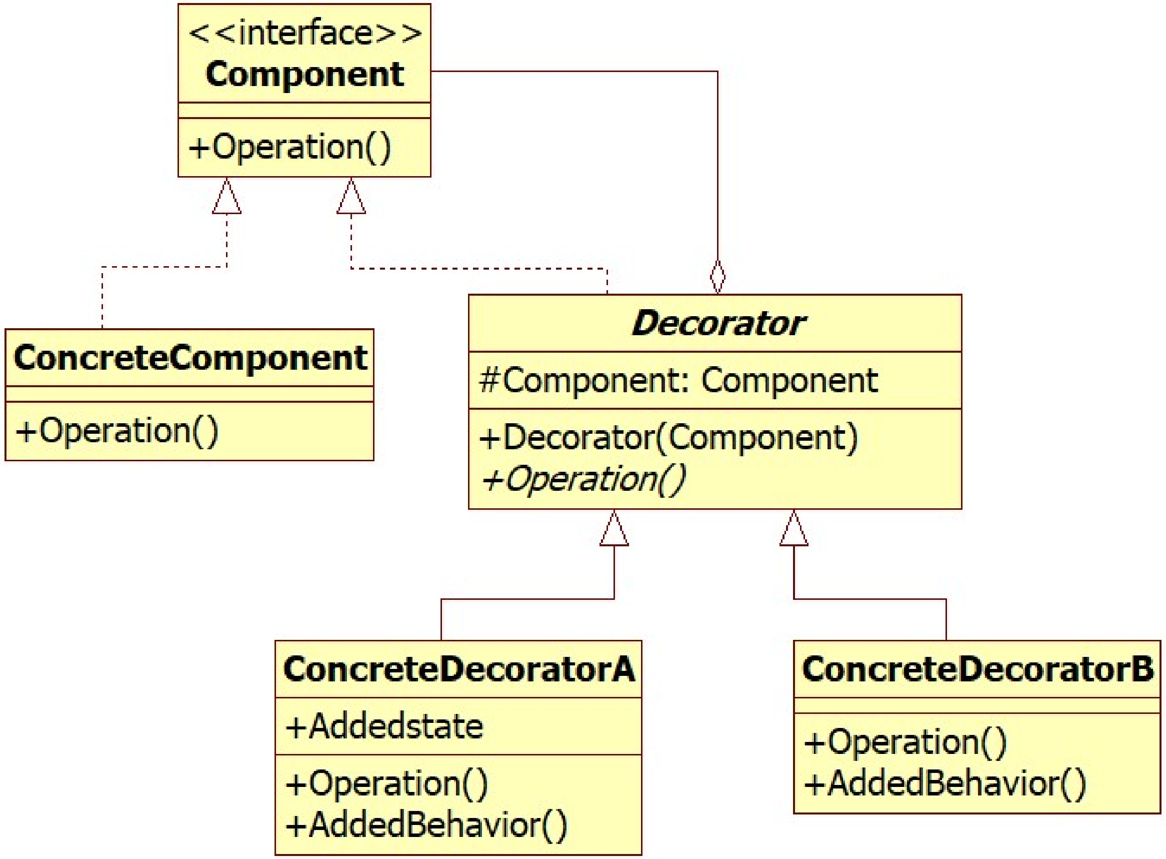
- The *Decorator* class is an abstract class and is the parent class of all the decorators. While it implements the Component interface to define the operations, it also contains a protected variable *component* that points to the object to be decorated. The *component* variable is simply assigned in the constructor. Decorator抽象类是所有装饰类的基类，里面包含的成员变量component 指向了被装饰的对象。
- The constructor for the Decorator class is simply:

```
public Decorator(Component input)
{
    this.component = input;
}
```



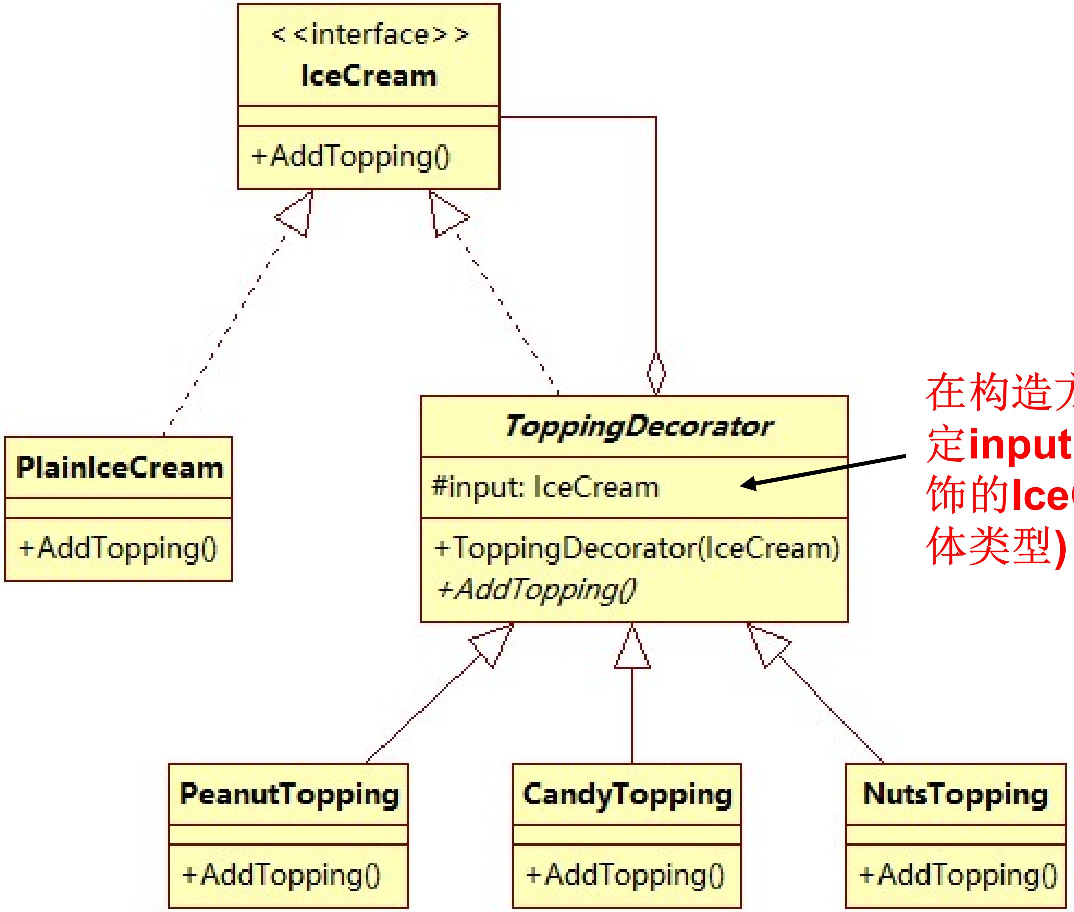
Decorator

- The *ConcreteDecorator* class are the actual decorator classes that can add features. You can have as many ConcreteDecorator class as you like, and each will represent a



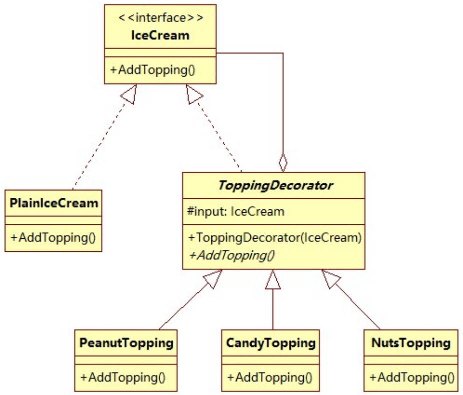
Example

- **In this example we have a plain ice cream where you can add different combination of toppings to it.**



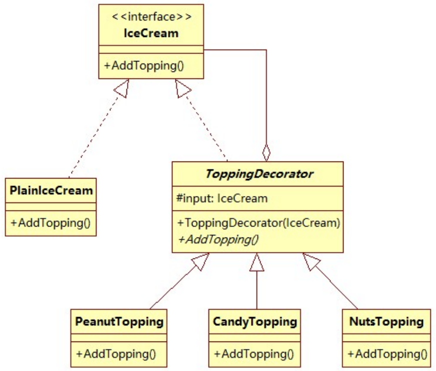
在构造方法中，指定**input**的值**(**被装饰的**IceCream**的具体类型**)**

Example



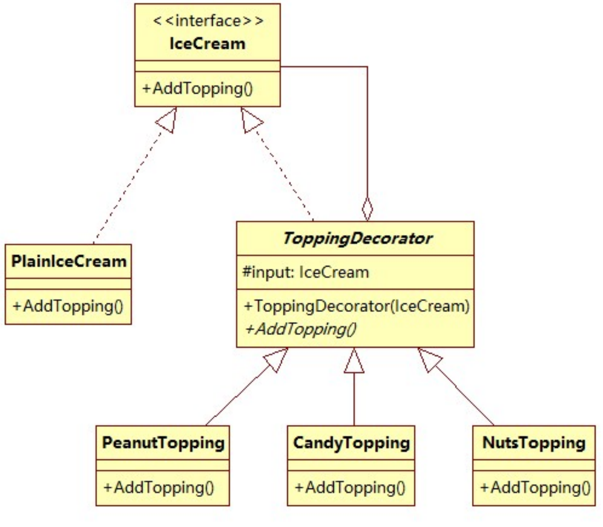
```
public interface IceCream { //顶层接口
    void AddTopping();
}
public class PlainIceCream implements IceCream{ //基础实现，无填加的冰激凌
    @Override
    public void AddTopping() {
        System.out.println("Plain IceCream ready for some
                                   toppings!");
    }
}

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

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

Example



```
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

}
```

Another Example



```
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!
```

Example2

- To construct a plain stack:
  - `Stack s = new ArrayStack();`
- To construct an undo stack:
  - `UndoStack s = new UndoStack(new ArrayStack());`
- To construct a secure synchronized undo stack:

  -
    ```
    SecureStack s = new SecureStack(
                             new SynchronizedStack(
                                new UndoStack(new ArrayStack()))
    ```
- **Flexibly Composible!**

Decorator vs. Inheritance

- **Decorator composes features at run time** 在运行时组合特性
  - Inheritance composes features at compile time 继承在编译时
- **Decorator consists of multiple collaborating objects** 由多个协作对象组成
  - Inheritance produces a single, clearly-typed object 继承常是单一的、清晰的对象
- **Can mix and match multiple decorations** 可以混合搭配多种装饰
  - Multiple inheritance is conceptually difficult

Decorators from `java.util.Collections`

- **Turn a mutable list into an 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);`

(2) Decorator

装饰器模式

Motivating example of Decorator pattern

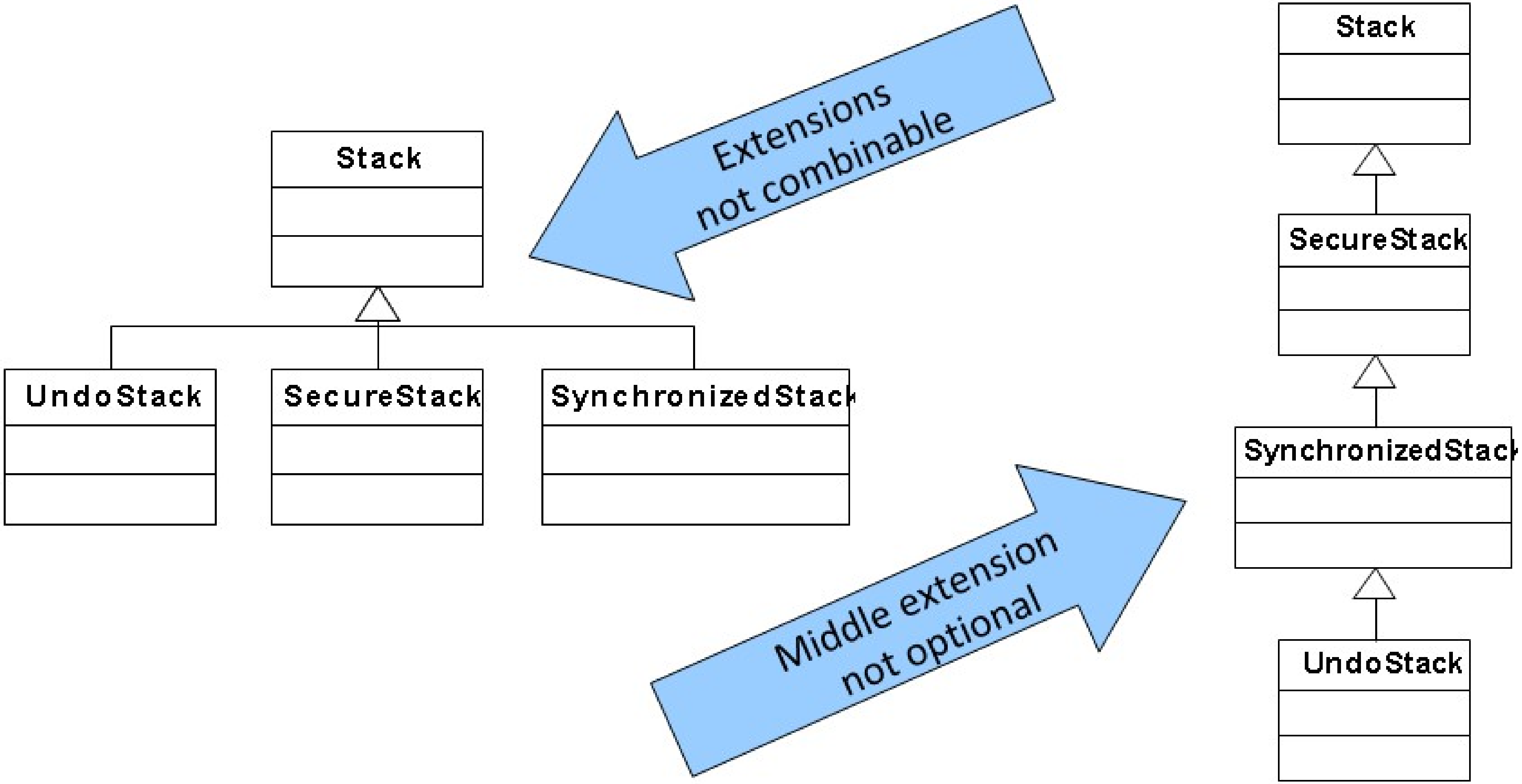
- **Suppose you want various extensions of a `Stack` data structure…**
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses
  - 用每个子类实现不同的特性

Inheritance

- **And arbitrarily composable extensions:** 如果需要特性的任意组合呢？
  - `SecureUndoStack`: A stack that requires a password, and also lets you undo previous operations
  - `SynchronizedUndoStack`: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - `SecureSynchronizedStack`: …
  - `SecureSynchronizedUndoStack`: …

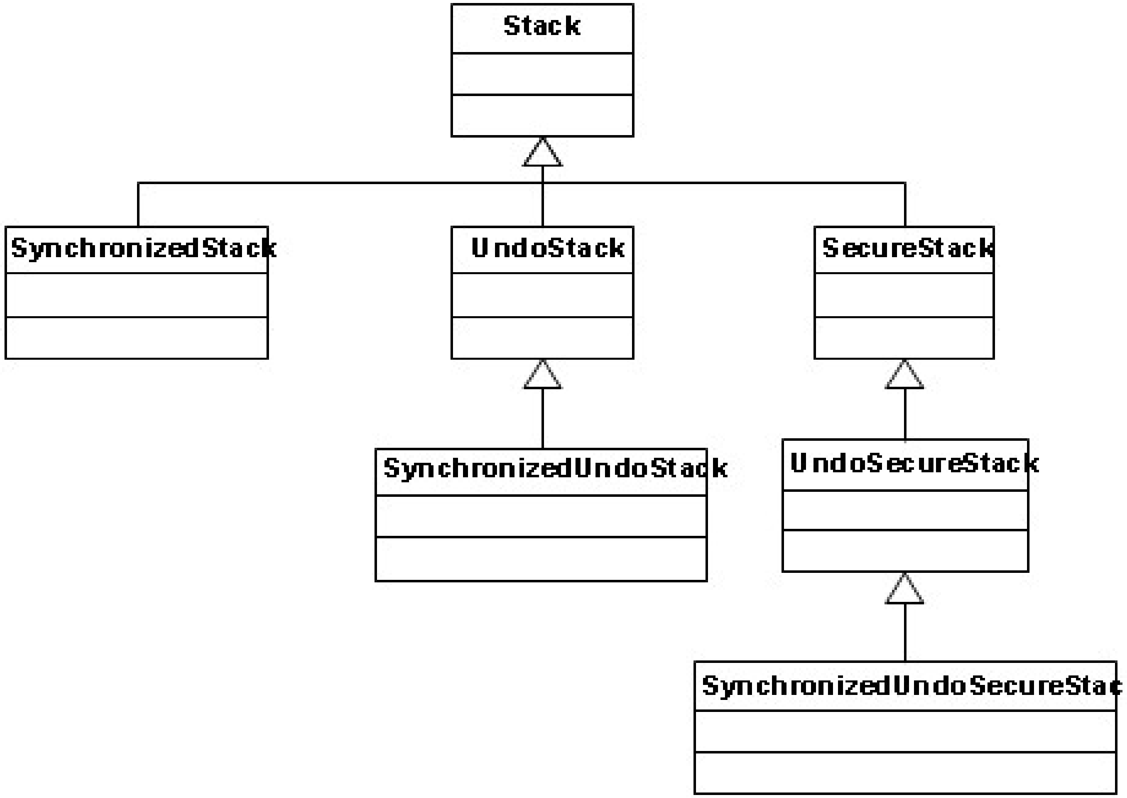
Inheritance hierarchies? Multi-Inheritance?

Limitations of inheritance



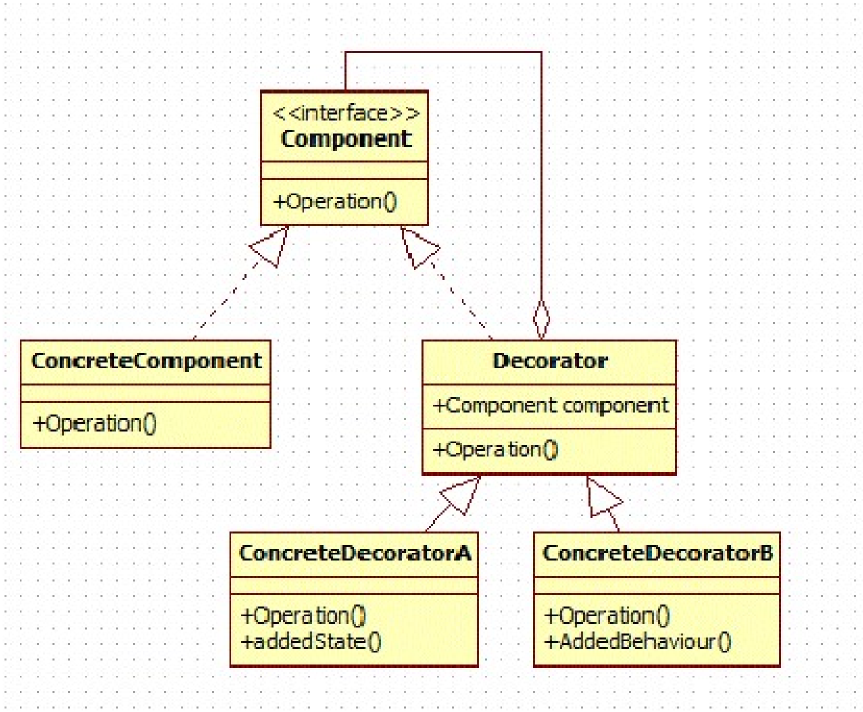
Limitations of inheritance

- **Combining inheritance hierarchies**
  - Combinatorial explosion 组合爆炸！
  - Massive code replication 大量的代码重复

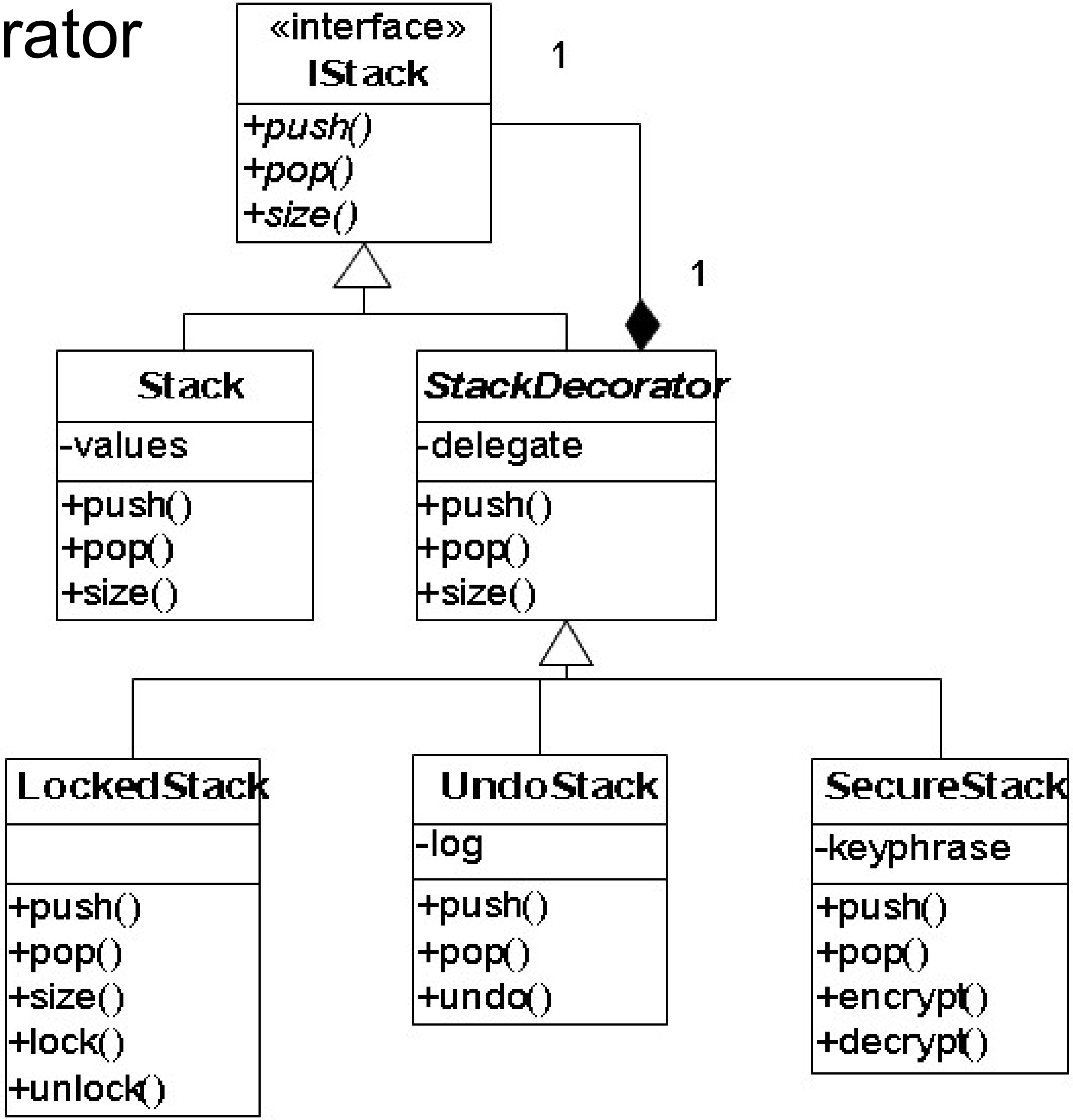


Decorator

- **Problem:** You need arbitrary or dynamically composable extensions to individual objects. 为对象增加不同侧面的特性
- **Solution**: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object. 对每一个特性构造子类，通过委派机制增加到对象上
- **Consequences:**
  - More flexible than static inheritance
  - Customizable, cohesive extensions
- **Decorators use both subtyping and delegation**



Example of Decorator



The ArrayStack Class

```
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

The AbstractStackDecorator Class

```
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)

The concrete decorator classes

```
public class UndoStack
       extends StackDecorator
       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

Using the decorator classes

- 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))
```

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…

Decorator vs. Inheritance

- **Decorator composes features at run time**
  - Inheritance composes features at compile time
- **Decorator consists of multiple collaborating objects**
  - Inheritance produces a single, clearly-typed object
- **Can mix and match multiple decorations**
  - Multiple inheritance is conceptually difficult

Decorators from `java.util.Collections`

- **Turn a mutable list into an 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);`

See section 3-1

- **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);`

See section 10-1

如何使用
factory method
模式实现

```
List<Trace> ts = new LinkedList<>();
List<Trace> ts2 =
       (List<Trace>) Collections.unmodifiableCollection(ts);

public static Stack UndoStackFactory(Stack stack) {
    return new UndoStack(stack);
}
```

3 Behavioral patterns

(1) Strategy

整体地替换算法

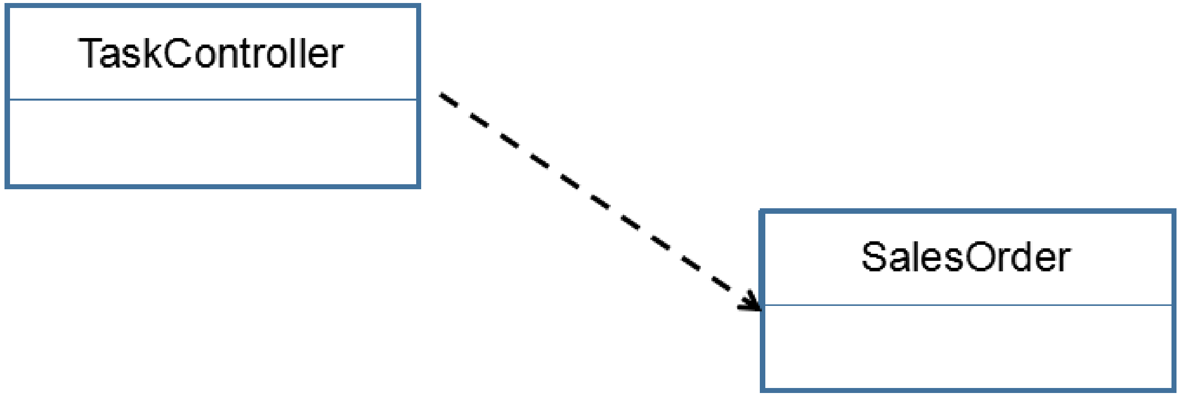
一个案例的研究

- 考虑一个国际电子商务公司的订单处理系统
- 此系统必须能够处理许多国家（地区）的订单。
- 为此，需要考虑需求可能出现各种变化。

- 上述情况在很多的项目中均会出现，因此设计时，需要考虑到系统的应变能力。

一个案例的研究

- 这个系统中有一个控制器对象，用于处理销售请求。
- 他能够确认何时有人在请求销售订单，并将请求转发给SalesOrder对象进行订单处理。



- 允许客户通过GUI填写订单
- 处理税额的计算
- 处理订单，打印销售收据

一个案例的研究

- 假设有新的需求，要求修改税额的处理办法：
  - 例如：要求处理本国之外的顾客的订单税额
  - 需要添加新的税额计算规则。

常规方法：

- *复制和粘贴*
- *使用switch或if语句，用变量指定*
- *使用函数指针或者委托*
- *继承*
- *将整个功能委托给新的对象*

```
//handle Tax switch(Nation){
      case US:
          US Tax rule here break;
      case Canada:
           Canada Tax rule here break;
}
```

```
//handle Currency switch(Nation){
      case US:
          US Currency rule here break;
      case Canada:
           Canada Currency rule here break;
}
```

```
//handle Language switch(Nation){
      case US:
          US Language break;
      case Canada:
           Canada Language break;
}
```

一个案例的研究

- 也许还有语言的问题，例如：德语、俄语、中文、意大利语、日语、……
- 突然情况变得很糟，要求添加加拿大魁北克省的法语。……

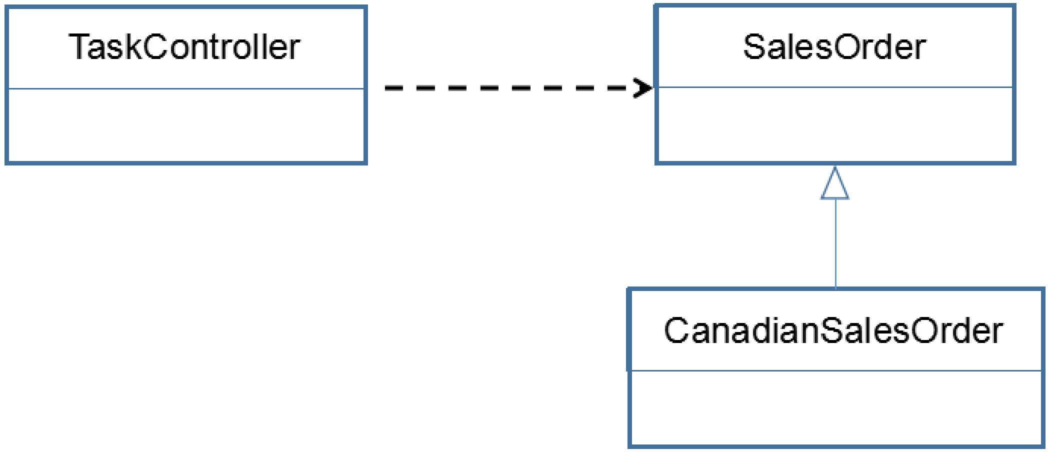
```
//handle Language switch(Nation){
      case US:
          US Language break;
      case Canada:
          if(inQuebec){
               //Use French break;
           }    else {
               //Canada Language break;
          }
     case Germany:
         //use German break;
}
```

- 分支流向开始模糊
- 阅读困难
- 理解困难
- 维护困难

“分支蔓延”

一个案例的研究

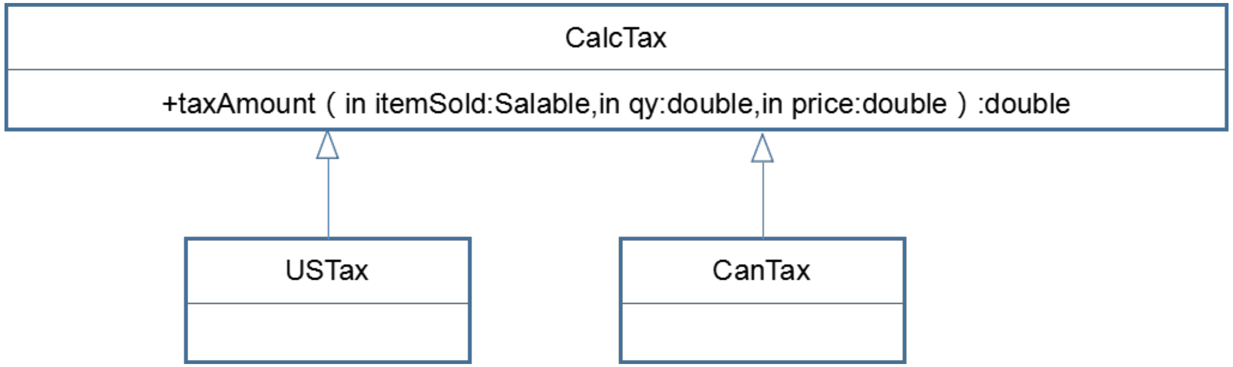
- 另一种思路，采用**OOP**中的继承技术：
  - 重用SalesOrder对象
  - 将新的缴税规则看成新种类的销售订单，只是缴税规则不一样。
  - 看上去很好，但是……
  - 如果又有新的变化，如：日期的格式、运费规则、……
  - 将导致很深的继承层，难于理解，冗余性高，内聚很弱，难以测试



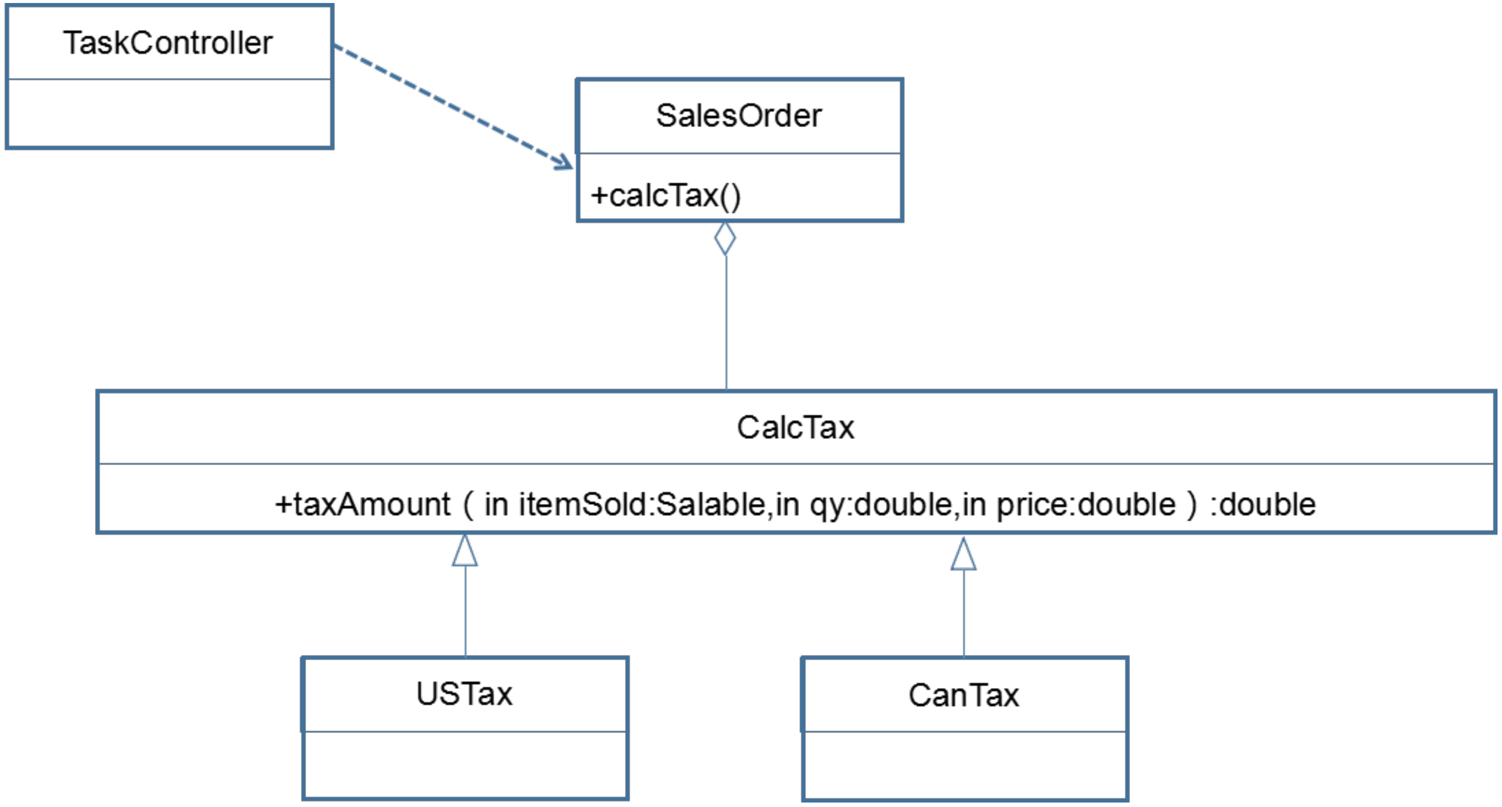
一个案例的研究

- 我们考虑：
  - 设计中什么应该是可变的；
  - 对变化的概念进行封装；
  - 优先使用对象聚集，而不是类继承；
- 改变一下思路：
  - 寻找变化，将其封装在一个单独的类中；
  - 将此类包含在另一个类中

本例中，预先要分析出缴税规则是确定变化的，将其封装，创建一个抽象类定义：如何在概念上完成税额计算，然后为每一变化派生具体的类。

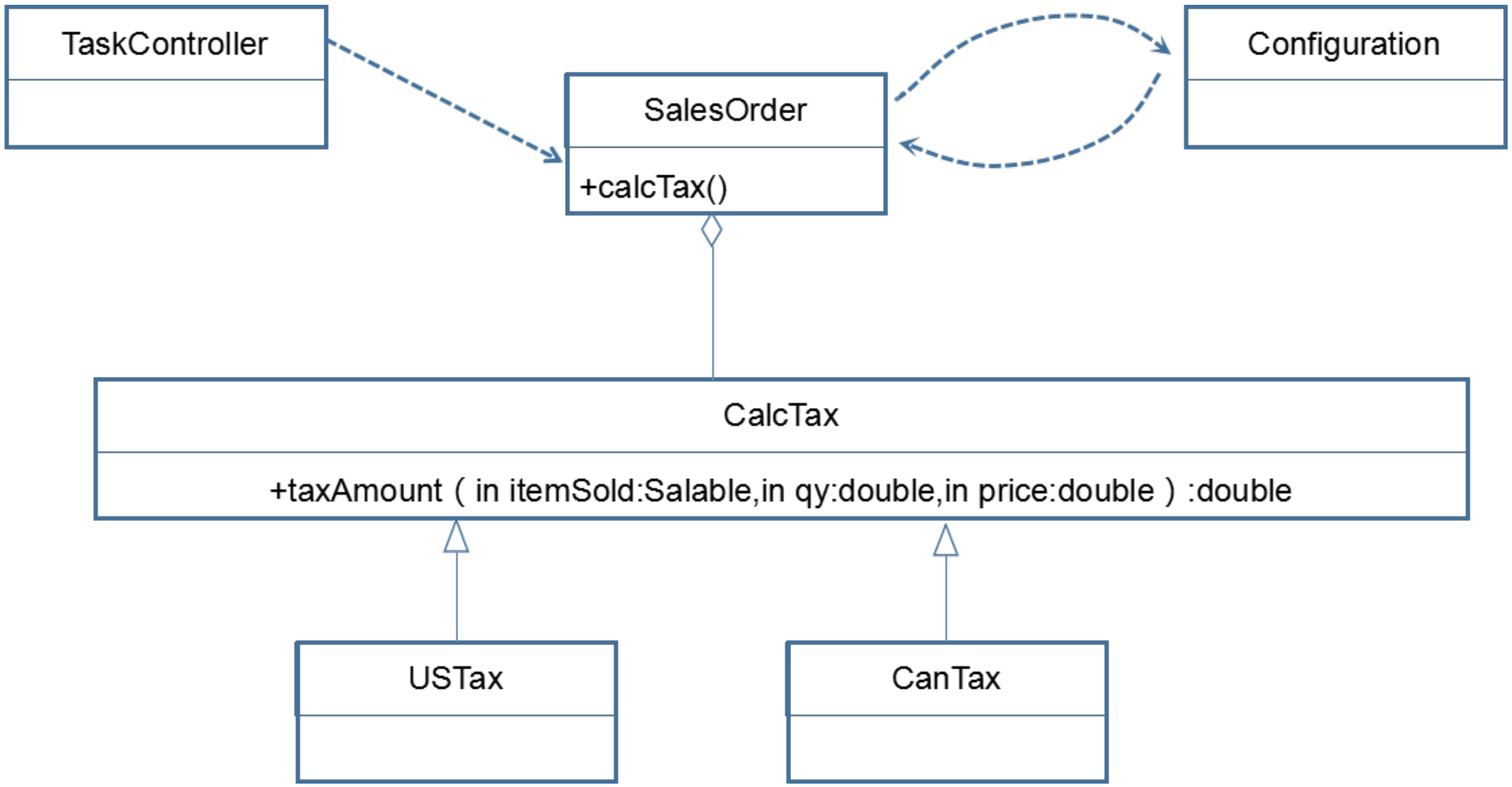

CalcTax
+taxAmount (in itemSold:Salable,in qy:double,in price:double) :double
USTax
CanTax

一个案例的研究


TaskController
SalesOrder
+calcTax()
CalcTax
+taxAmount (in itemSold:Salable,in qy:double,in price:double) :double
USTax
CanTax

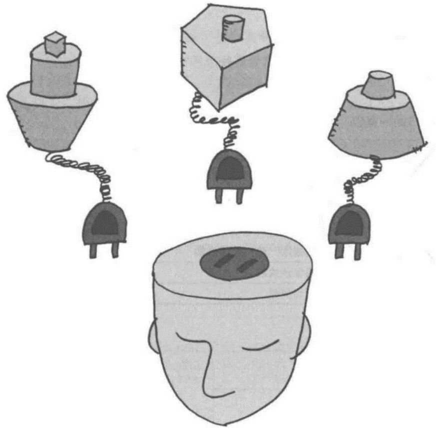
```
public class TaskController {
            public void process(){
        //this code is an emulation of a
        //processing task controller
        // ......
        //figure out which country you are in

        CalcTax myTax;
        myTax=getTaxRulesForCountey();
        SalesOrder mySO=new SalesOrder();
        mySO.process(myTax);
            }
      private CalcTax getTaxRulesForCountry(){
              //in real life,get the tax rules base on
              //country you are in.You may have the
              //configuration file
              //here just ruturn a USTax
              return USTax();
            }
}
```

```
public class SalesOrder{
    public void process(CalcTax taxToUse){
        long itemNumber=0;
        double price=0;
        //give the tax object to use
         //.....
         //Calculate tax
        double tax=taxToUse.taxAmount(long
itemSold,double price);
       }   }
public CanTax extends CalcTax{
      public double taxAmountlong
itemSold,double price){
         //according to the tax rule in Canada and
         //ruturn it
        return 0.0;
    }      }
public USTax extends CalcTax{
    public double taxAmountlong itemSold,double
price){
         //according to the tax rule in US and ruturn
it
        return 0.0;
     }     }
```


TaskController
SalesOrder
+calcTax()
Configuration
CalcTax
+taxAmount (in itemSold:Salable,in qy:double,in price:double) :double
USTax
CanTax

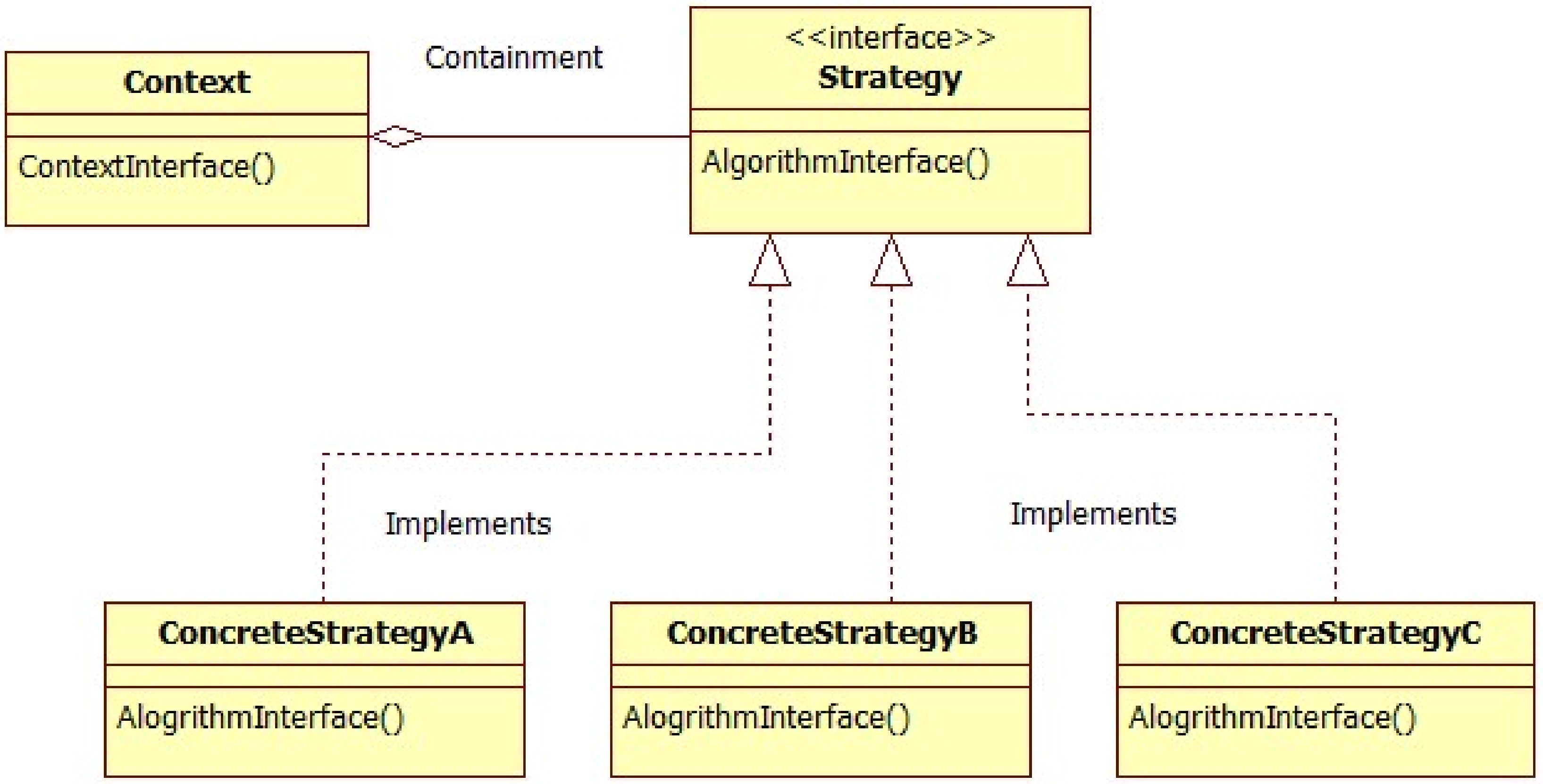
Strategy Pattern

- **Problem:** Different algorithms exists for a specific task, but client can switch between the algorithms at run time in terms of dynamic context. 针对特定任务存在多种算法，调用者需要根据上下文环境动态的选择和切换。
- **Example:** Sorting a list of customers (Bubble sort, mergesort, quicksort)
- **Solution:** Create an interface for the algorithm, with an implementing class for each variant of the algorithm. 定义一个算法的接口，每个算法用一个类来实现，客户端针对接口编写程序。
- **Advantage:**
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context

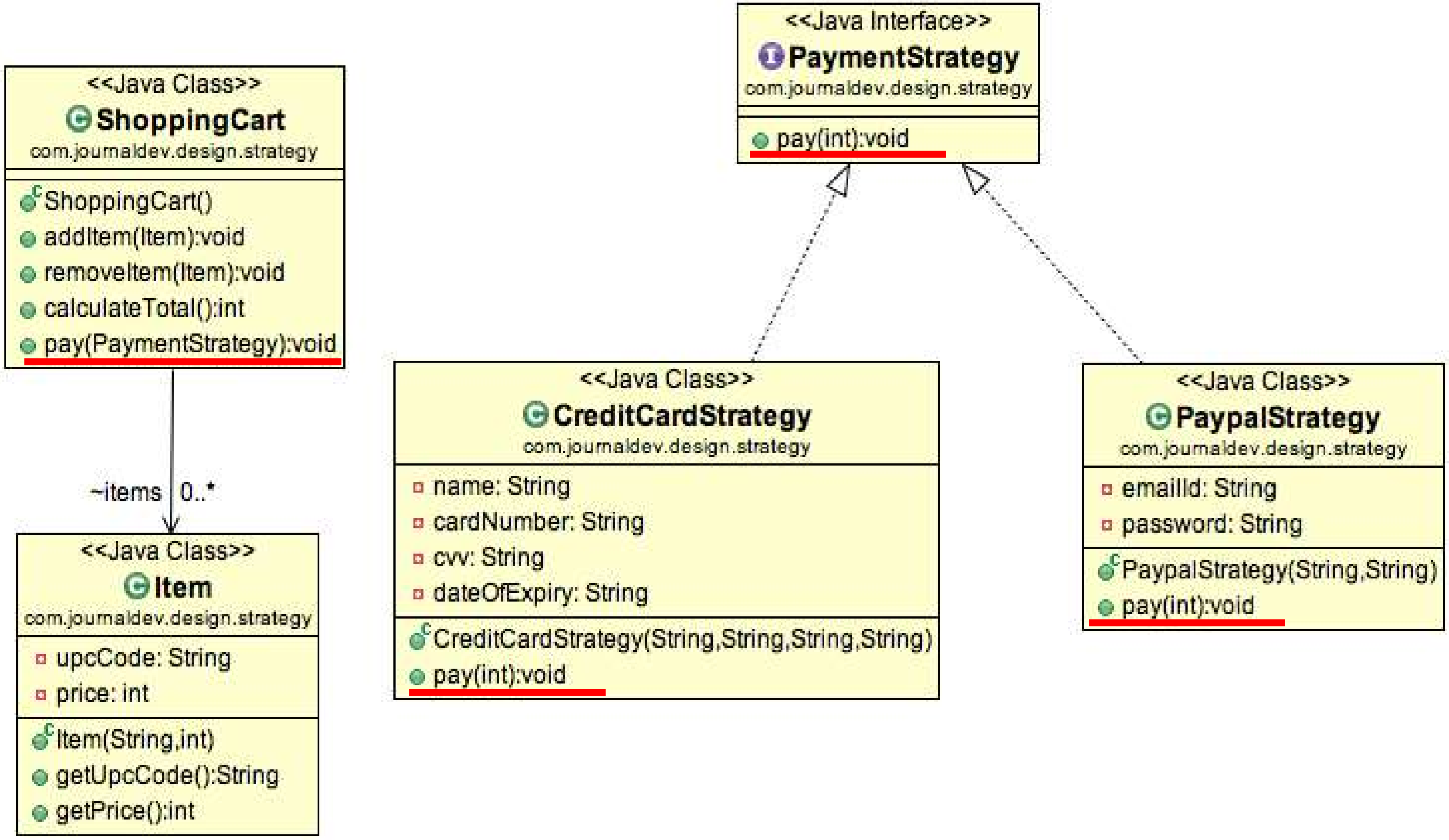


- 整体地替换算法

Strategy Pattern

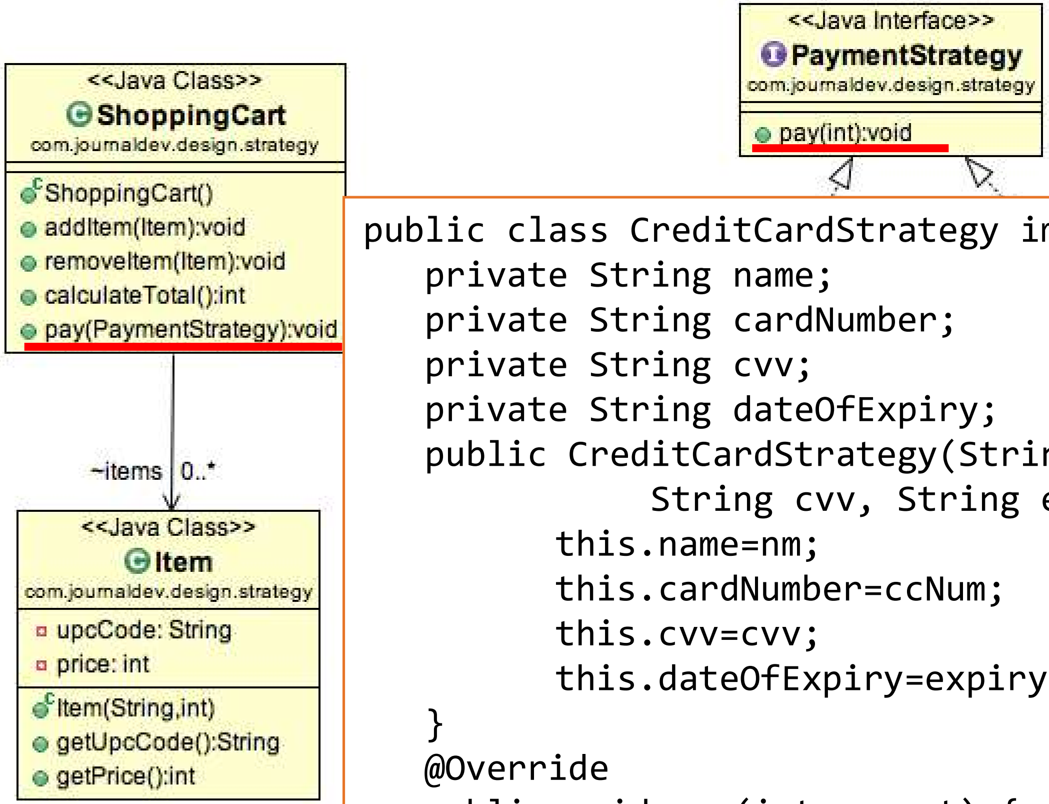


Code example



Code example

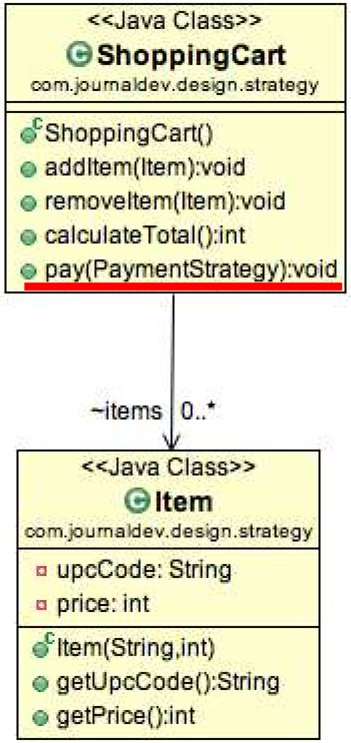
```
public interface PaymentStrategy {
     public void pay(int amount);
}
```



```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
            String cvv, String expiryDate){
        this.name=nm;
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
   }
   @Override
   public void pay(int amount) {
        System.out.println(amount +" paid with credit card");
   }
}
```

Code example

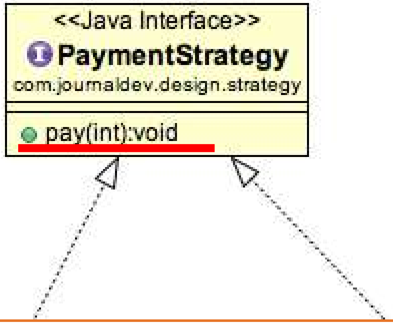
```
public interface PaymentStrategy {
     public void pay(int amount);
}
```



<<Java Class>>
ShoppingCart
com.journaldev.design.strategy
ShoppingCart()
addItem(Item):void
removeItem(Item):void
calculateTotal():int
pay(PaymentStrategy):void

~items 0..*

<<Java Class>>
Item
com.journaldev.design.strategy
upcCode: String
price: int
Item(String,int)
getUpcCode():String
getPrice():int



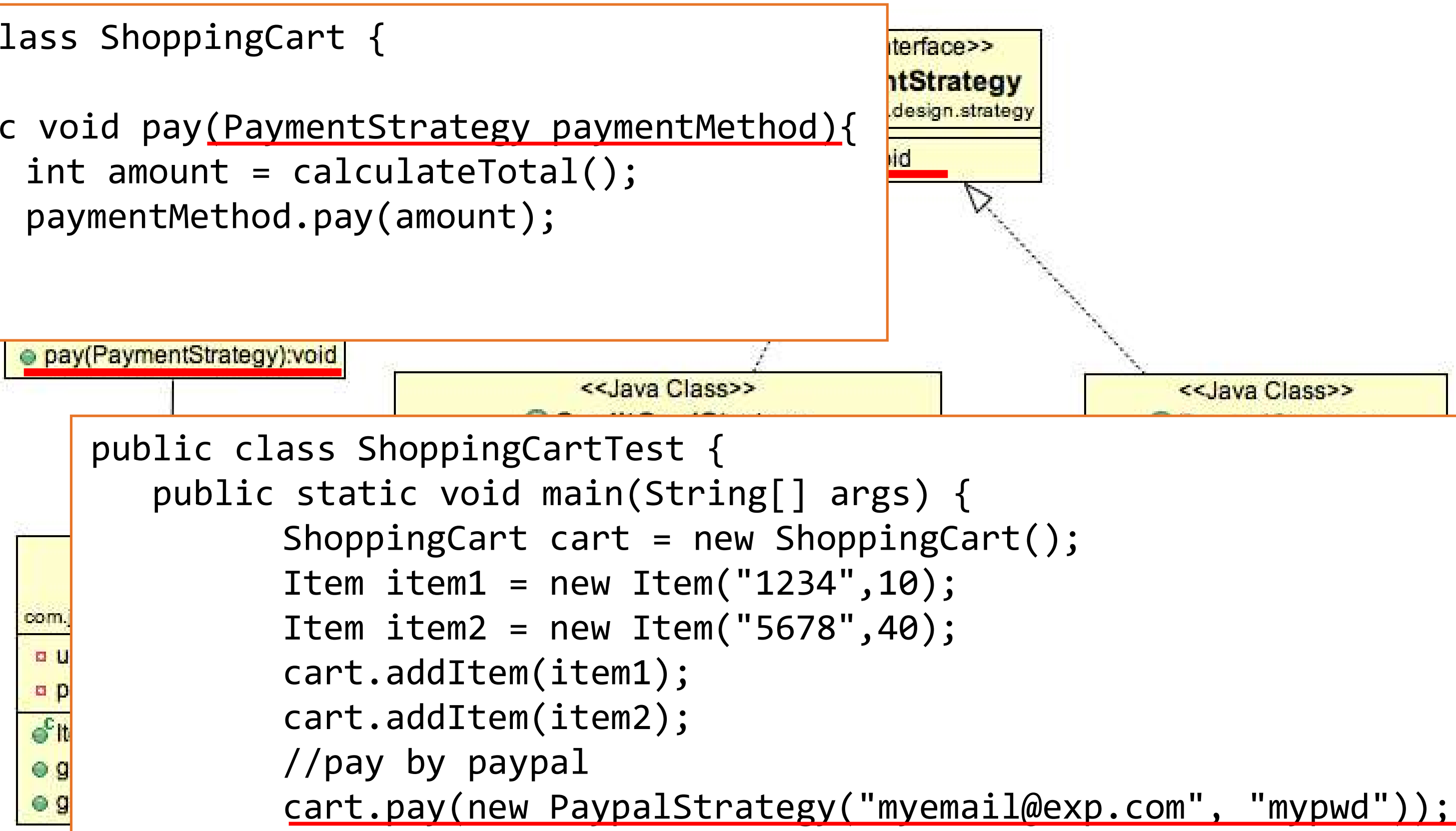
<<Java Interface>>
PaymentStrategy
com.journaldev.design.strategy
pay(int):void

```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
         this.emailId=email;
         this.password=pwd;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount + " paid using Paypal.");
   }
}
```

Code example

```
public interface PaymentStrategy {
     public void pay(int amount);
}
```

```
public class ShoppingCart {
   ...
   public void pay(PaymentStrategy paymentMethod){
         int amount = calculateTotal();
         paymentMethod.pay(amount);
   }
}
```



```
public class ShoppingCartTest {
   public static void main(String[] args) {
         ShoppingCart cart = new ShoppingCart();
         Item item1 = new Item("1234",10);
         Item item2 = new Item("5678",40);
         cart.addItem(item1);
         cart.addItem(item2);
         //pay by paypal
         cart.pay(new PaypalStrategy("myemail@exp.com", "mypwd"));
         //pay by credit card
         cart.pay(new CreditCardStrategy(“Alice", "1234", "786", "12/18"));
   }
}
```

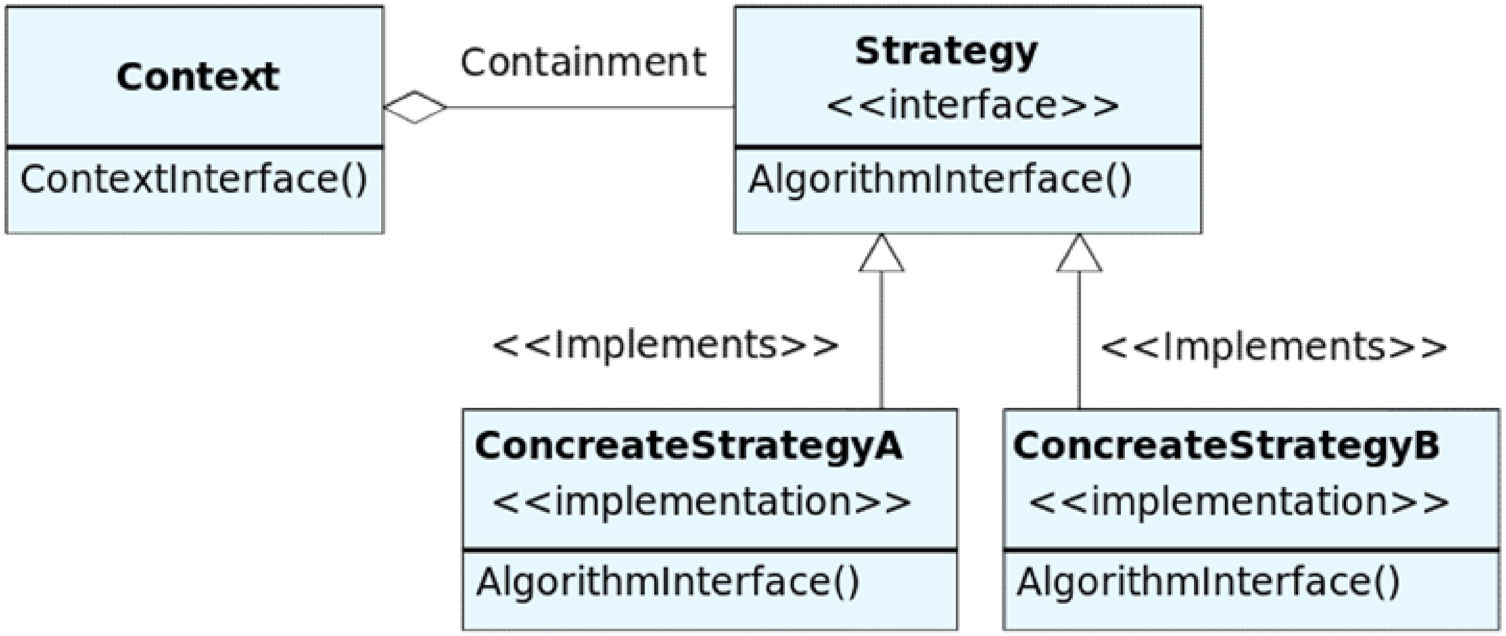
(1) Strategy

策略模式

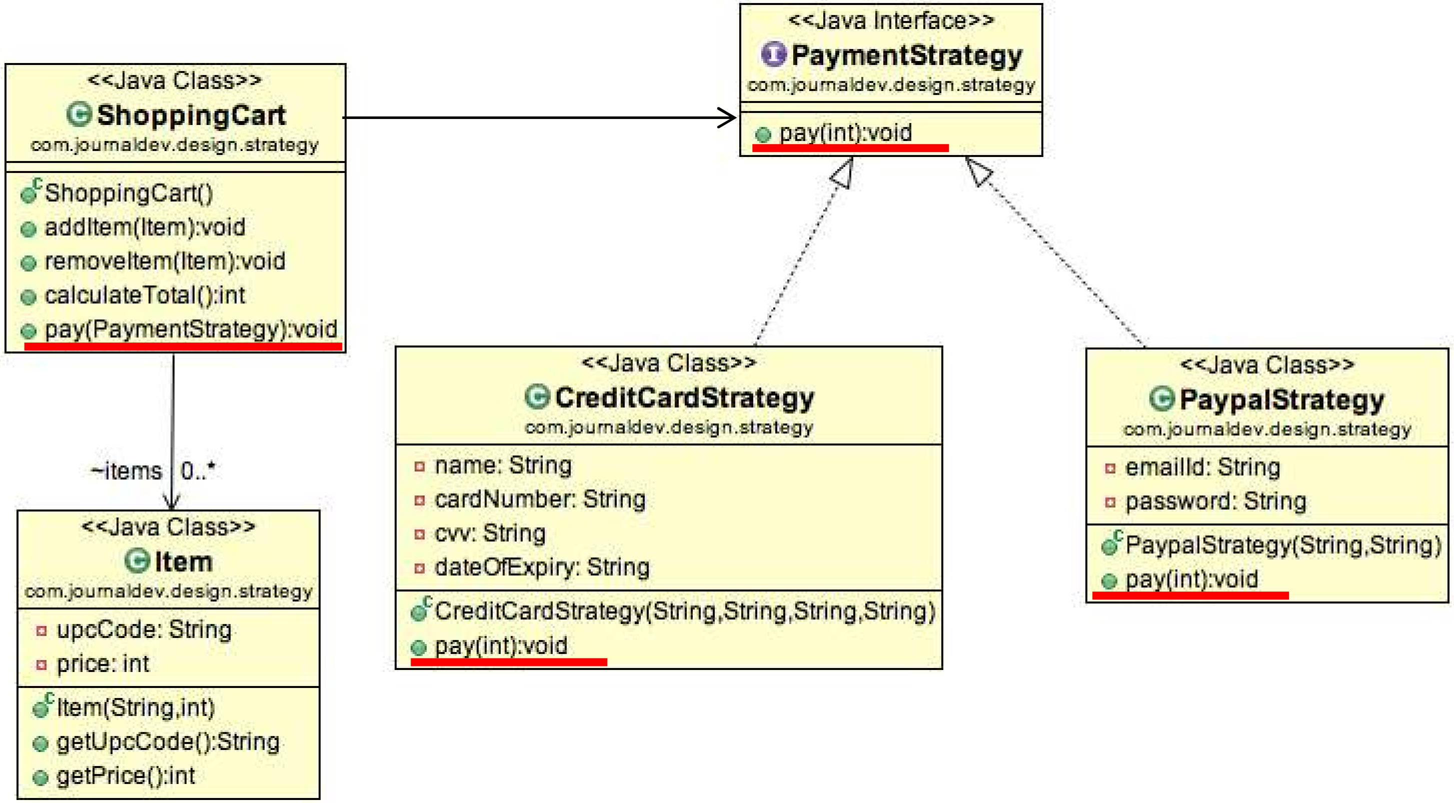
Strategy Pattern

- **Problem:** Different algorithms exists for a specific task, but client can switch between the algorithms at run time in terms of dynamic context. 有多种不同的算法来实现同一个任务，但需要client根据需要动态切换算法，而不是写死在代码里
- **Example:** Sorting a list of customers (bubble sort, mergesort, quicksort)
- **Solution:** Create an interface for the algorithm, with an implementing class for each variant of the algorithm. 为不同的实现算法构造抽象接口，利用delegation，运行时动态传入client倾向的算法类实例
- **Advantage:**
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context

Strategy Pattern



Code example



<<Java Class>>
ShoppingCart
com.journaldev.design.strategy

ShoppingCart()
addItem(Item):void
removeItem(Item):void
calculateTotal():int
pay(PaymentStrategy):void

<<Java Interface>>
PaymentStrategy
com.journaldev.design.strategy

pay(int):void

~items 0..*

<<Java Class>>
Item
com.journaldev.design.strategy

upcCode: String
price: int

Item(String,int)
getUpcCode():String
getPrice():int

<<Java Class>>
CreditCardStrategy
com.journaldev.design.strategy

name: String
cardNumber: String
cvv: String
dateOfExpiry: String

CreditCardStrategy(String,String,String,String)
pay(int):void

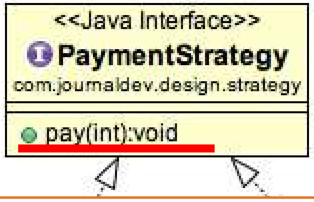
<<Java Class>>
PaypalStrategy
com.journaldev.design.strategy

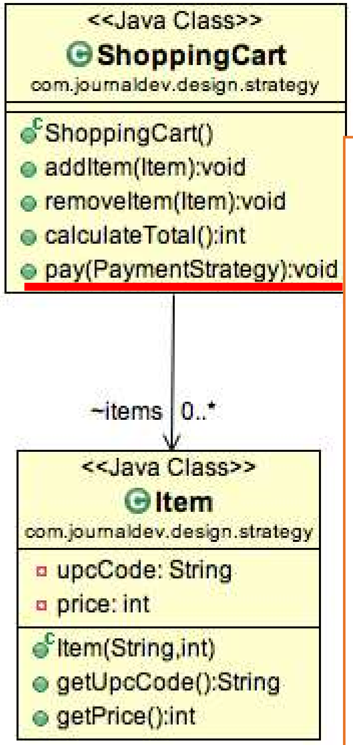
emailId: String
password: String

PaypalStrategy(String,String)
pay(int):void

Code example

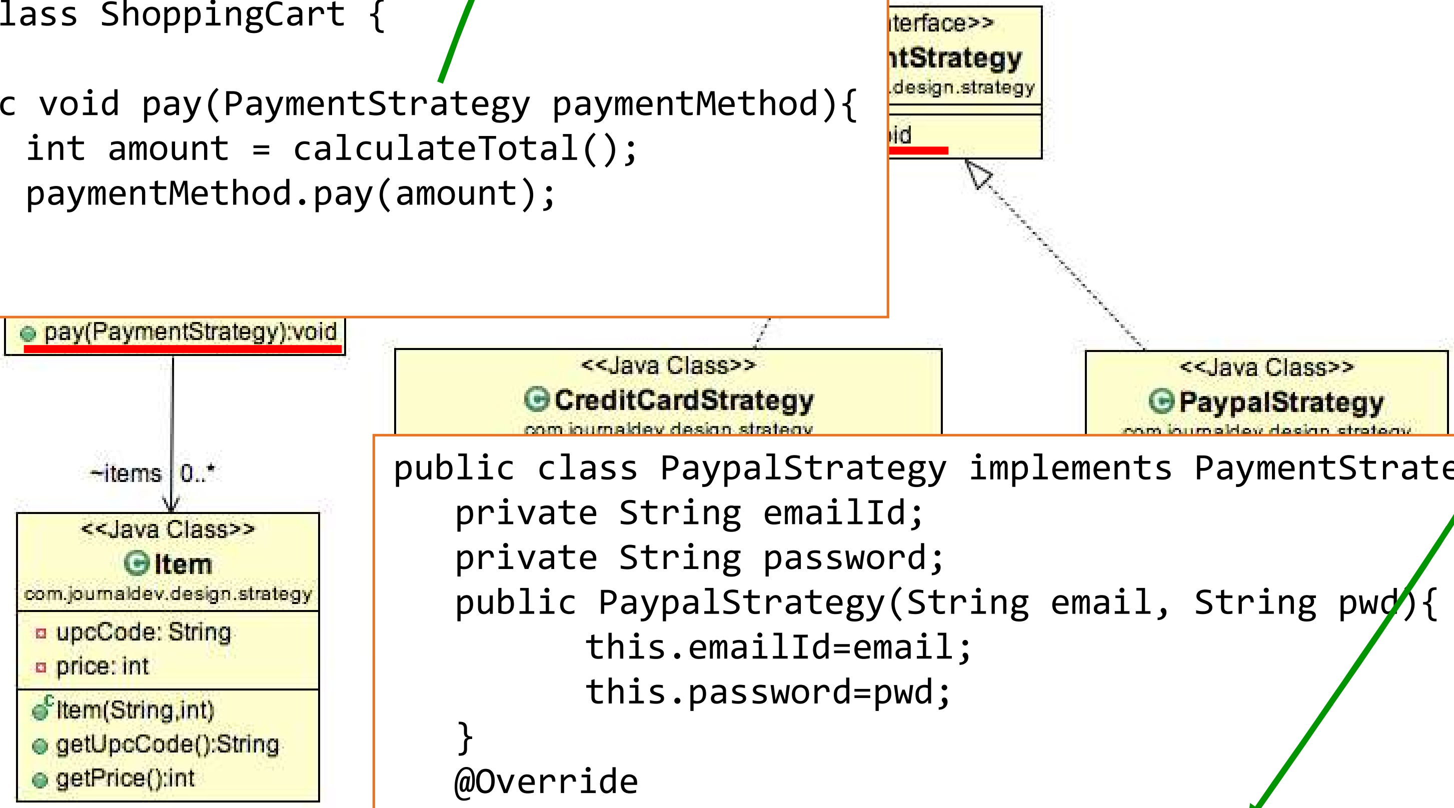
```
public interface PaymentStrategy {
     public void pay(int amount);
}
```





```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
               String cvv, String expiryDate){
         this.name=nm;
         this.cardNumber=ccNum;
         this.cvv=cvv;
         this.dateOfExpiry=expiryDate;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount +" paid with credit card");
   }
}
```

Code example



```
public interface PaymentStrategy {
     public void pay(int amount);
}
```

```
public class ShoppingCart {
   ...
   public void pay(PaymentStrategy paymentMethod){
         int amount = calculateTotal();
         paymentMethod.pay(amount);
   }
}
```

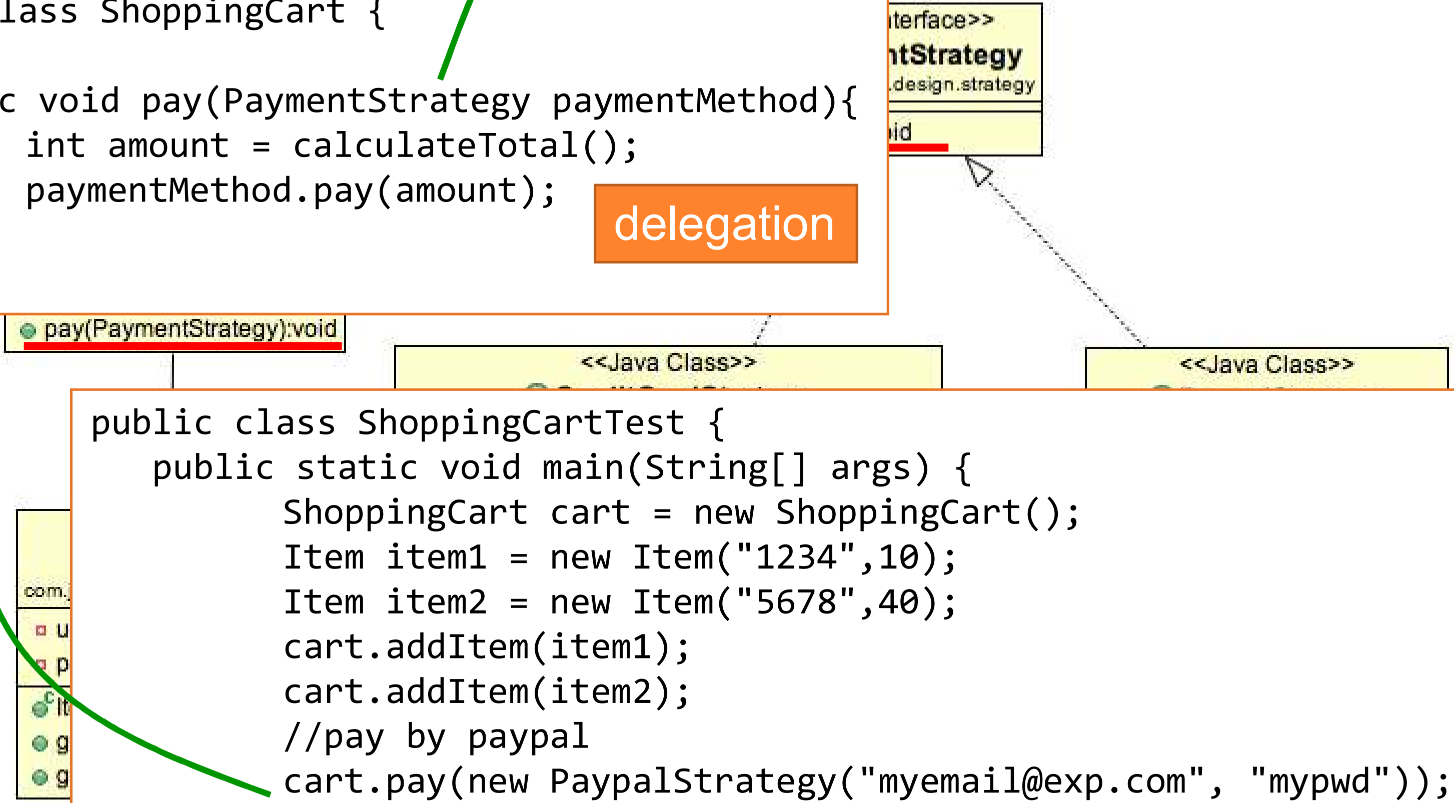
```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
         this.emailId=email;
         this.password=pwd;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount + " paid using Paypal.");
   }
}
```

Code example

```
public interface PaymentStrategy {
     public void pay(int amount);
}
```

```
public class ShoppingCart {
   ...
   public void pay(PaymentStrategy paymentMethod){
         int amount = calculateTotal();
         paymentMethod.pay(amount);
   }
}
```

delegation

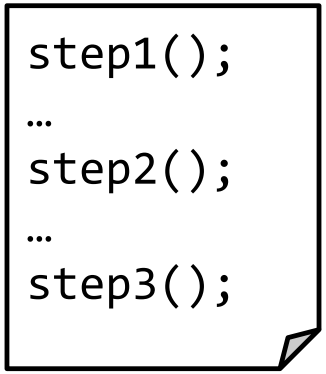


```
public class ShoppingCartTest {
   public static void main(String[] args) {
         ShoppingCart cart = new ShoppingCart();
         Item item1 = new Item("1234",10);
         Item item2 = new Item("5678",40);
         cart.addItem(item1);
         cart.addItem(item2);
         //pay by paypal
         cart.pay(new PaypalStrategy("myemail@exp.com", "mypwd"));
         //pay by credit card
         cart.pay(new CreditCardStrategy(“Alice", "1234", "786", "12/18"));
   }
}
```

(2) Template Method

Template Method Motivation

- **Problem:** Several clients share the same algorithm but differ on the specifics, i.e., an algorithm consists of customizable parts and invariant parts. **Common steps should not be duplicated in the subclasses but need to be reused.** 不同的客户端具有相同的算法步骤，但是每个步骤的具体实现不同。

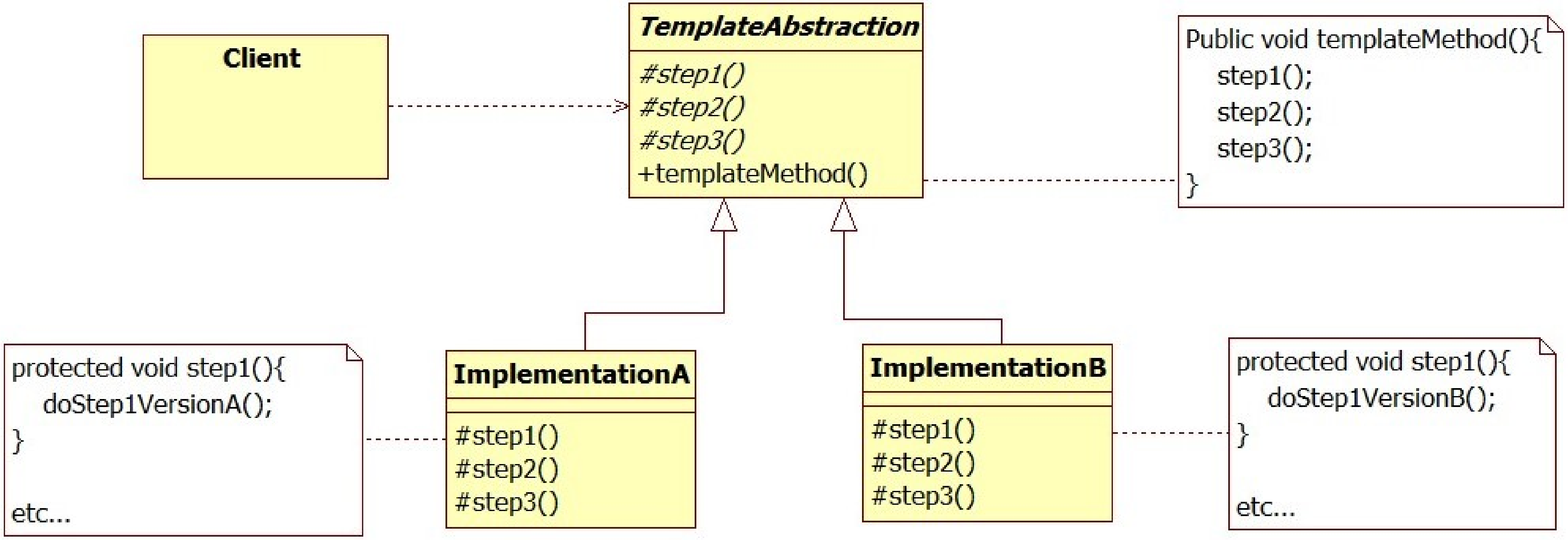


```
step1();
…
step2();
…
step3();
```

- **Examples:**
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types
- **Solution:**
  - The common steps of the algorithm are factored out into an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm.在父类中定义通用逻辑和各步骤的抽象方法声明
  - Subclasses provide different realizations for each of these steps.

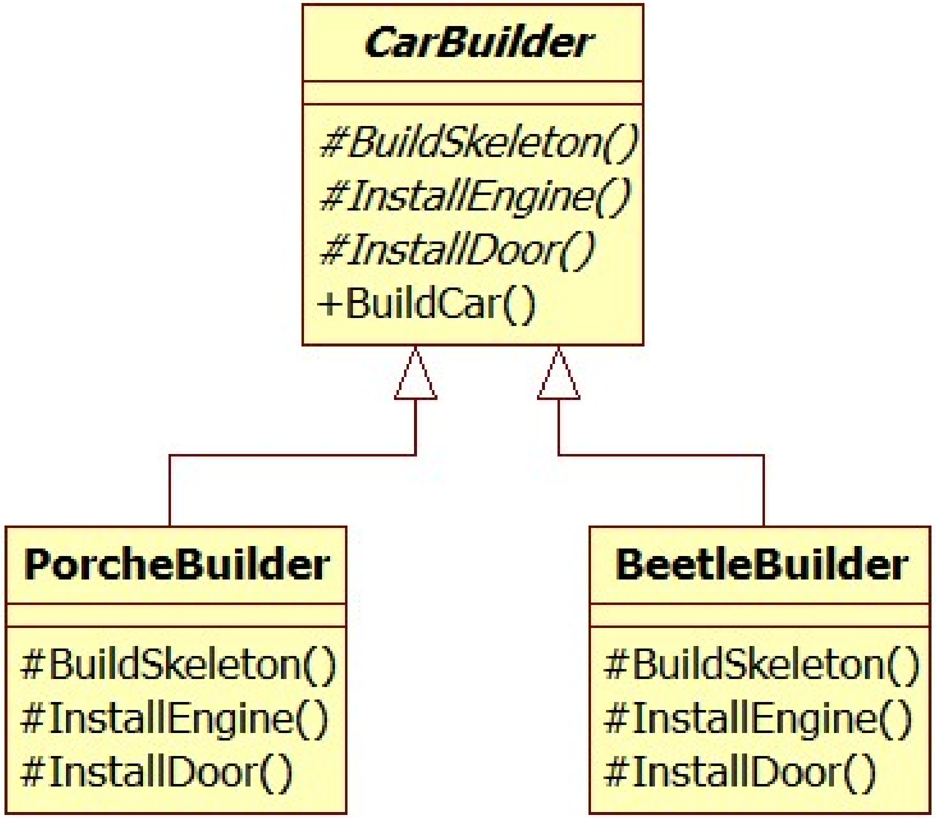
  子类中进行各步骤的具体实现

Template Method pattern



Example

- In the example, we will build 2 types of cars. One is a Porsche, the other is a VW Beetle.



Example

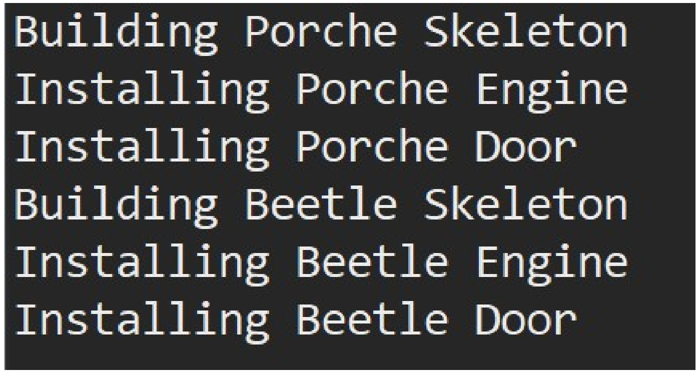
```
public abstract class CarBuilder {
        protected abstract void BuildSkeleton();
        protected abstract void InstallEngine();
        protected abstract void InstallDoor();

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

Example

```
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");
        }
}
```

Example

```
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
```

Template Method Pattern Applicability

- Template Method Design Pattern allows you to declare a general logic at the parent class so that all the child classes can use the general logic. 在父类声明一个通用逻辑
- **Template method** pattern uses **inheritance + overridable** methods **to vary part of an algorithm** 模板模式用继承**+**重写的方式实现算法的不同部分。
  - While strategy pattern uses delegation to vary the entire algorithm (interface and polymorphism). 策略模式用委托机制实现不同完整算法的调用**(**接口**+**多态**)**
- **Template Method is widely used in frameworks**
  - The framework implements the invariants of the algorithm 框架实现了算法的不变性
  - The client customizations provide specialized steps for the algorithm 客户端提供每步的具体实现
  - Principle: “Don’t call us, we’ll call you”.

(2) Template Method

模板模式

Template Method

- **Problem:** Several clients share the same algorithm but differ on the specifics, i.e., an algorithm consists of customizable parts and invariant parts. **Common steps should not be duplicated in the subclasses but need to be reused.**
  - 做事情的步骤一样，但具体方法不同

```
step1();
…
step2();
…
step3();
```

- **Examples:**
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types
- **Solution:**
  - The common steps of the algorithm are factored out into an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. 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
  - Subclasses provide different realizations for each of these steps.

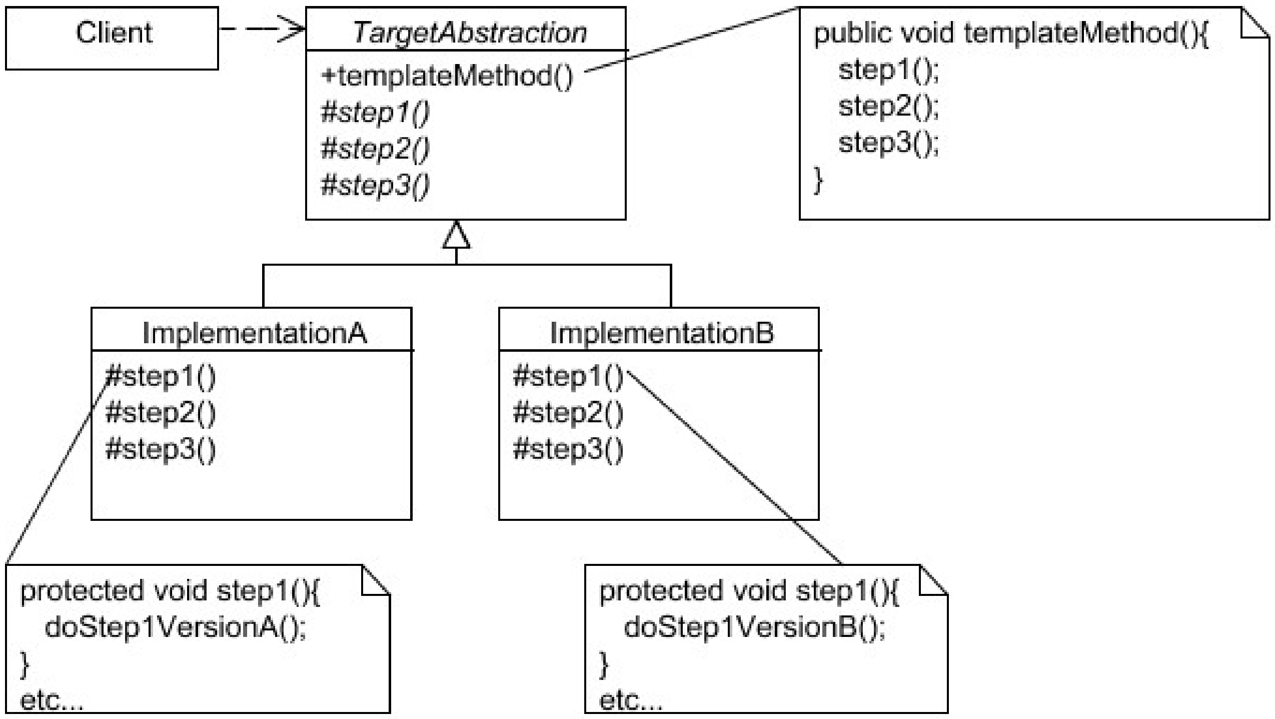
Template Method Pattern

- **Template method** pattern uses inheritance + overridable methods **to vary part of an algorithm** 使用继承和重写实现模板模式
  - While strategy pattern uses delegation to vary the entire algorithm (interface and ad-hoc polymorphism).

Whitebox or Blackbox framework?

- **Template Method is widely used in frameworks**
  - The framework implements the invariants of the algorithm
  - The client customizations provide specialized steps for the algorithm
  - Principle: “Don’t call us, we’ll call you”.

Template Method pattern

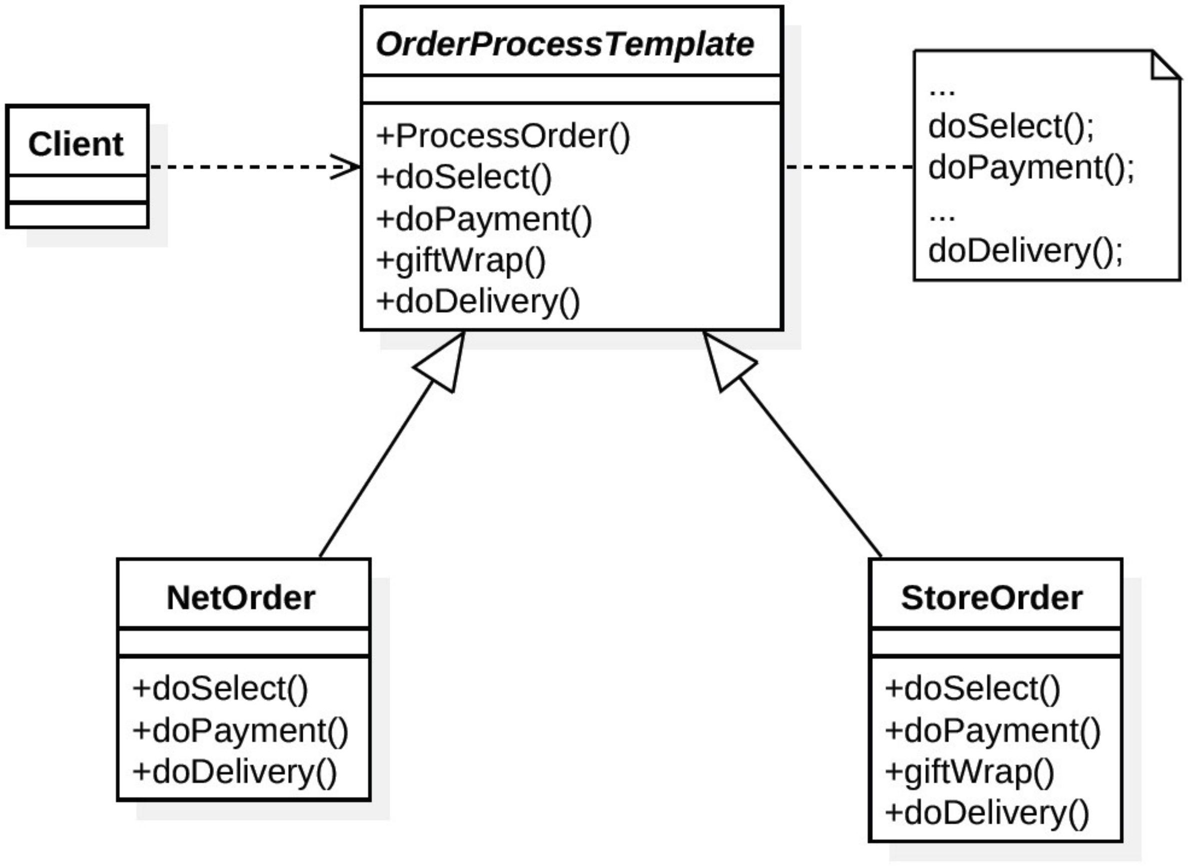


```
public void templateMethod(){
   step1();
   step2();
   step3();
}
```

```
protected void step1(){
   doStep1VersionA();
}
etc...
```

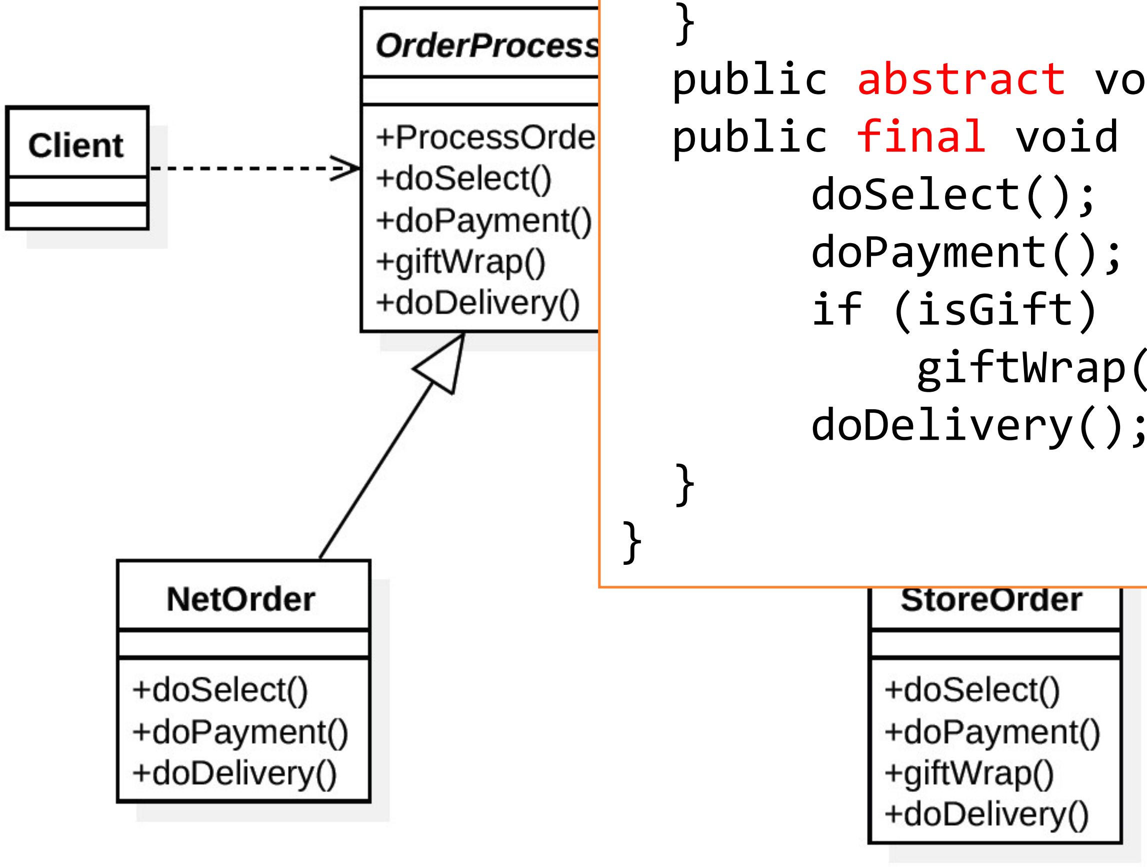
```
protected void step1(){
   doStep1VersionB();
}
etc...
```

Example


Client
OrderProcessTemplate
+ProcessOrder()
+doSelect()
+doPayment()
+giftWrap()
+doDelivery()
...
doSelect();
doPayment();
...
doDelivery();
NetOrder
+doSelect()
+doPayment()
+doDelivery()
StoreOrder
+doSelect()
+doPayment()
+giftWrap()
+doDelivery()

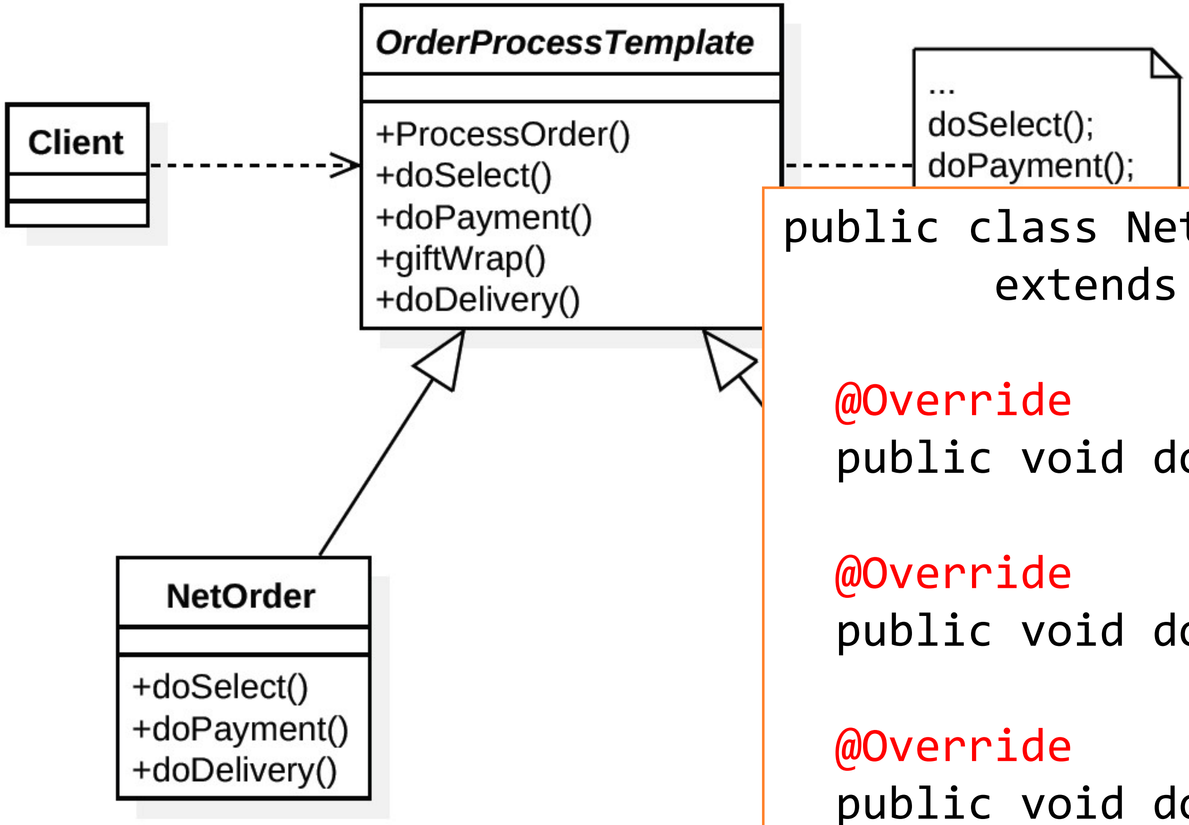
Example

```
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();
  }
}
```



Example

```
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() { … }
}
```

See the whitebox framework

Overriding

```
public class Calculator extends Application {
  protected String getApplicationTitle() { return "My Great Calculator"; }
  protected String getButtonText() { return "calculate"; }
  protected String getInititalText() { return "(10 - 3) * 6"; }
  protected void buttonClicked() {
    JOptionPane.showMessageDialog(this, "The result of " + getInput() +
        " is " + calculate(getInput()));
  }
  private String calculate(String text) { ... }
}
```

Extension via subclassing and overriding methods
Subclass has main method but gives control to framework

```
public class Ping extends Application {
  protected String getApplicationTitle() { return "Ping"; }
  protected String getButtonText() { return "ping"; }
  protected String getInititalText() { return "127.0.0.1"; }
  protected void buttonClicked() { ... }
}
```

Overriding

(3) Iterator

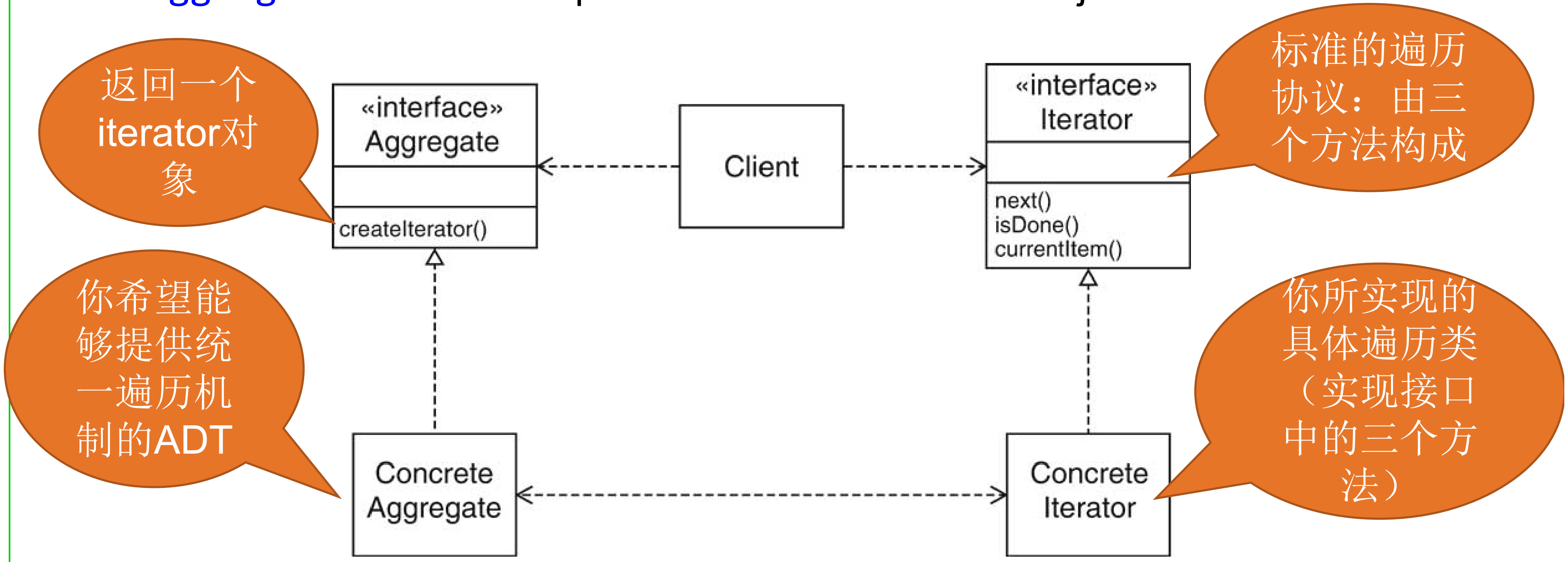
迭代器模式

Iterator Pattern

- **Problem:** Clients need uniform strategy to access all elements in a container, independent of the container type 客户端希望遍历被放入容器/集合类的一组ADT对象，无需关心容器的具体类型
  - 也就是说，不管对象被放进哪里，都应该提供同样的遍历方式
- **Solution:** A strategy pattern for iteration
- **Consequences:**
  - Hides internal implementation of underlying container
  - Support multiple traversal strategies with uniform interface
  - Easy to change container type
  - Facilitates communication between parts of the program

Iterator Pattern

- **Pattern structure**
  - Abstract Iterator class defines traversal protocol
  - Concrete Iterator subclasses for each aggregate class
  - Aggregate 聚集体 instance creates instances of Iterator objects
  - Aggregate instance keeps reference to Iterator object



Iterator pattern

- 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();
}
```

- **Iterator pattern**：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：

```
for (E e : collection) { … }

Iterator<E> iter = collection.iterator();
while(iter.hasNext()) { … }
```

Getting an Iterator

```
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.

An example of Iterator pattern

```
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) { … }
```

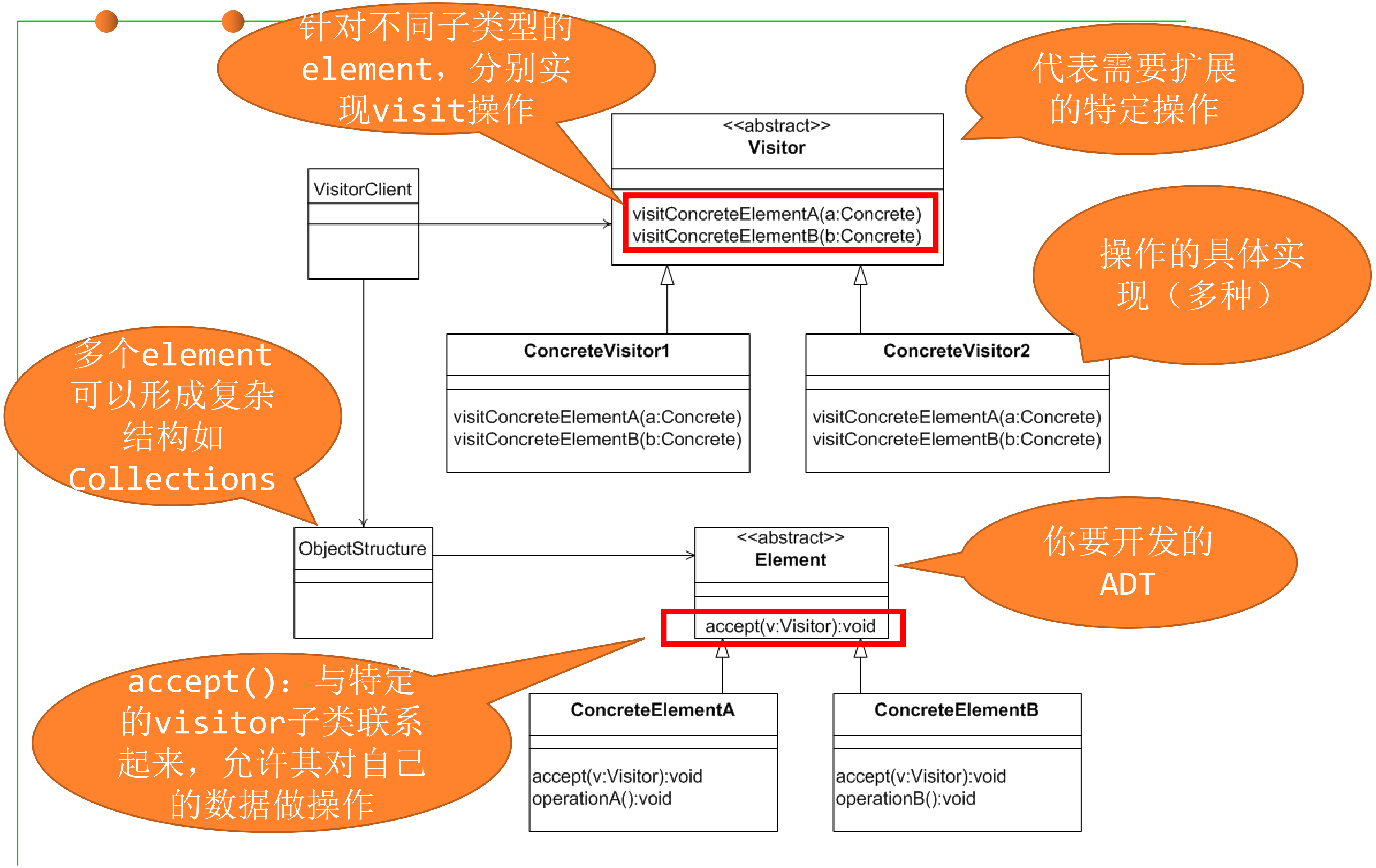
(4) Visitor

访问者模式

Visitor Pattern

- **Visitor pattern: Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.** 对特定类型的**object**的特定操作**(visit)**，在运行时将二者动态绑定到一起，该操作可以灵活更改，无需更改被**visit**的类
  - What the Visitor pattern actually does is to create an external class that uses data in the other classes.
  - If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- 本质上：将数据和作用于数据上的某种**/**些特定操作分离开来。
- 为**ADT**预留一个将来可扩展功能的“接入点”，外部实现的功能代码可以在不改变**ADT**本身的情况下通过**delegation**接入**ADT**

Visitor Pattern


针对不同子类型的element，分别实现visit操作
代表需要扩展的特定操作
<<abstract>>
Visitor
visitConcreteElementA(a:Concrete)
visitConcreteElementB(b:Concrete)
VisitorClient
操作的具体实现（多种）
ConcreteVisitor1
visitConcreteElementA(a:Concrete)
visitConcreteElementB(b:Concrete)
ConcreteVisitor2
visitConcreteElementA(a:Concrete)
visitConcreteElementB(b:Concrete)
多个element可以形成复杂结构如Collections
ObjectStructure
<<abstract>>
Element
accept(v:Visitor):void
你要开发的ADT
accept()：与特定的visitor子类联系起来，允许其对自己的数据做操作
ConcreteElementA
accept(v:Visitor):void
operationA():void
ConcreteElementB
accept(v:Visitor):void
operationB():void

Example

```
/* Abstract element interface (visitable) */
public interface ItemElement {
   public int accept(ShoppingCartVisitor visitor);
}


/* Concrete element */
public class Book implements ItemElement{
  private double price;
  ...
  int accept(ShoppingCartVisitor visitor) {
     visitor.visit(this);
  }
}
public class Fruit implements ItemElement{
  private double weight;
  ...
  int accept(ShoppingCartVisitor visitor) {
     visitor.visit(this);
  }
}
```

将处理数据的功能delegate到外部传入的visitor

Example

```
/* Abstract visitor interface */
public interface ShoppingCartVisitor {
   int visit(Book book);
   int visit(Fruit fruit);
}
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {
   public int visit(Book book) {
     int cost=0;
     if(book.getPrice() > 50){
        cost = book.getPrice()-5;
     }else
        cost = book.getPrice();
     System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
     return cost;
   }
   public int visit(Fruit fruit) {
     int cost = fruit.getPricePerKg()*fruit.getWeight();
     System.out.println(fruit.getName() + " cost = "+cost);
     return cost;
   }
}
```

这里只列出了一种visitor实现

这个visit操作的功能完全可以在Book类内实现为一个方法，但这就不可变了

Example

```
public class ShoppingCartClient {

   public static void main(String[] args) {

      ItemElement[] items = new ItemElement[]{
               new Book(20, "1234"),new Book(100, "5678"),
               new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
      }

      private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
           sum = sum + item.accept(visitor);
        return sum;
      }
}
```

只要更换visitor的具体实现，即可切换算法

Visitor vs Iterator

- Iterator: behavioral pattern, is used to access an aggregate sequentially without exposing its underlying representation. So you could hide a `List` or `array` or similar aggregates behind an `Iterator`. 迭代器：以遍历的方式访问集合数据而无需暴露其内部表示，将“遍历”这项功能delegate到外部的iterator对象。

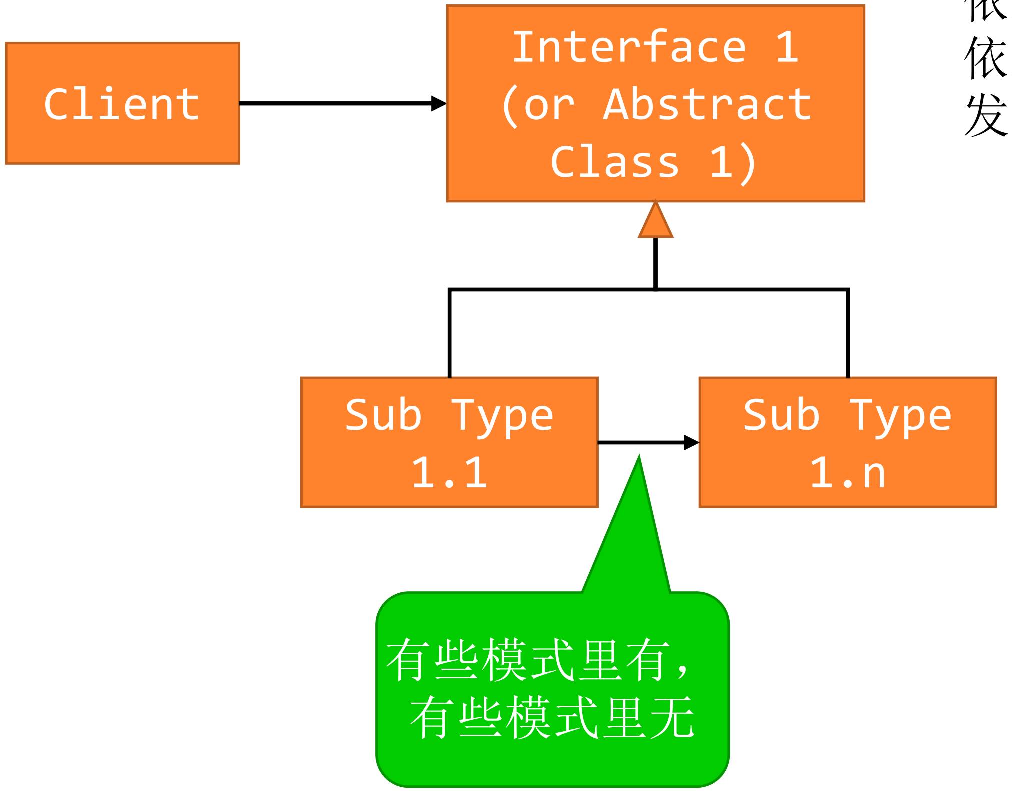
- Visitor: behavioral pattern, is used to perform an action on a structure of elements without changing the implementation of the elements themselves. 在特定ADT上执行某种特定操作，但该操作不在ADT内部实现，而是delegate到独立的visitor对象，客户端可灵活扩展/改变visitor的操作算法，而不影响ADT

Strategy vs visitor

- **Visitor: behavioral pattern**
- **Strategy: behavioral pattern**
- 二者都是通过**delegation**建立两个对象的动态联系
  - 但是Visitor强调是的外部定义某种对ADT的操作，该操作于ADT自身关系不大（只是访问ADT），故ADT内部只需要开放accept(visitor)即可，client通过它设定visitor操作并在外部调用。
  - 而Strategy则强调是对ADT内部某些要实现的功能的相应算法的灵活替换。这些算法是ADT功能的重要组成部分，只不过是delegate到外部strategy类而已。
- 区别：visitor是站在外部client的角度，灵活增加对ADT的各种不同操作（哪怕ADT没实现该操作），strategy则是站在内部ADT的角度，灵活变化对其内部功能的不同配置。

4 Commonality and Difference of Design Patterns

设计模式的对比：共性样式1



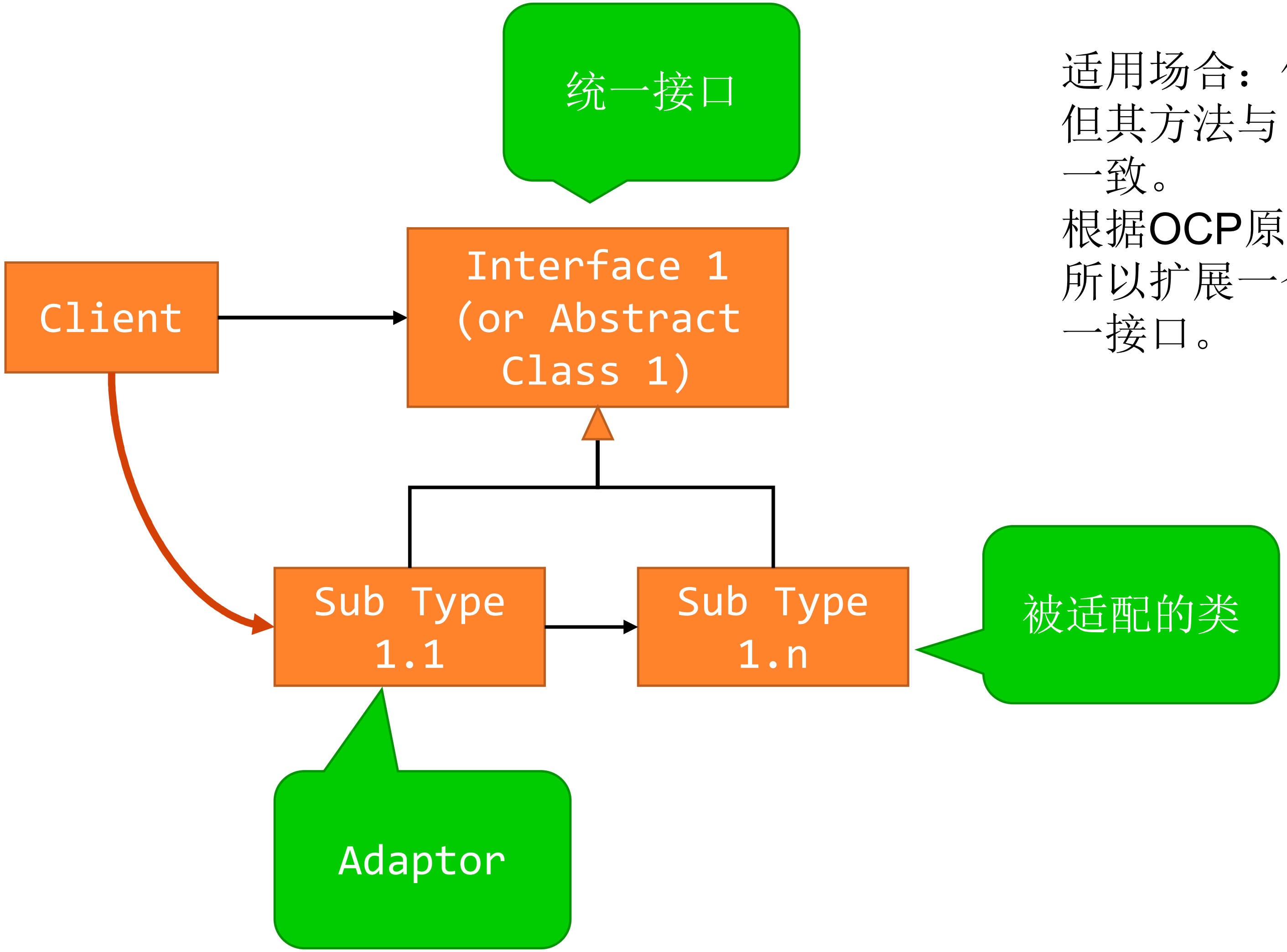
只使用“继承”，不使用“delegation”

核心思路：OCP/DIP

依赖反转，客户端只依赖“抽象”，不能依赖于“具体”

发生变化时最好是“扩展”而不是“修改”

Adaptor



适用场合：你已经有了一个类，但其方法与目前client的需求不一致。

根据OCP原则，不能改这个类，所以扩展一个adaptor和一个统一接口。

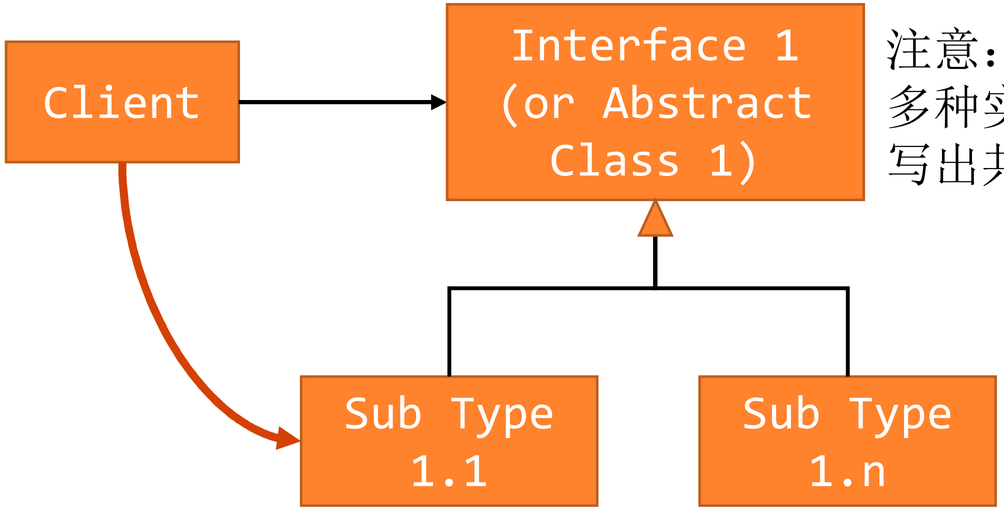
Template

(1)要提供一个统一的算法方法，final的，按次序调用一系列代表算法步骤的abstract方法

(2) 要提供一组abstract方法，分别代表算法的某个步骤

适用场合：有共性的算法流程，但算法各步骤有不同的实现

典型的“将共性提升至超类型，将个性保留在子类型”

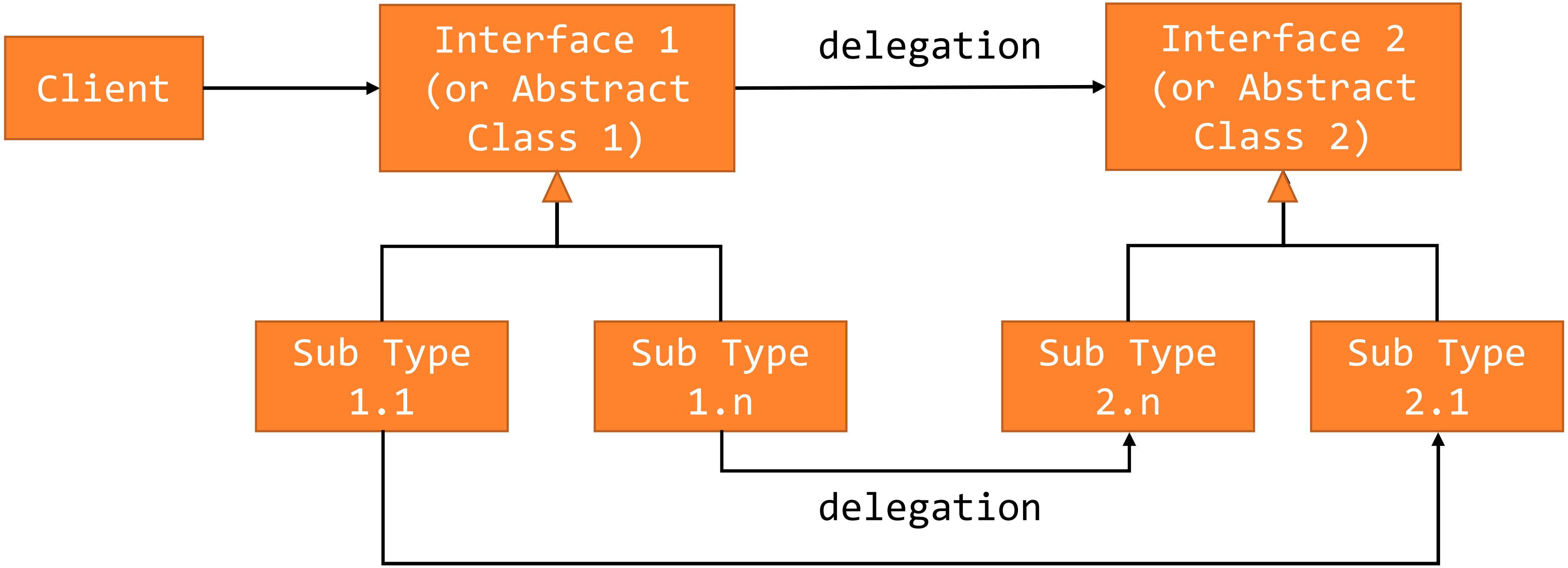


注意：如果某个步骤不需要有多种实现，直接在该抽象类里写出共性实现即可。

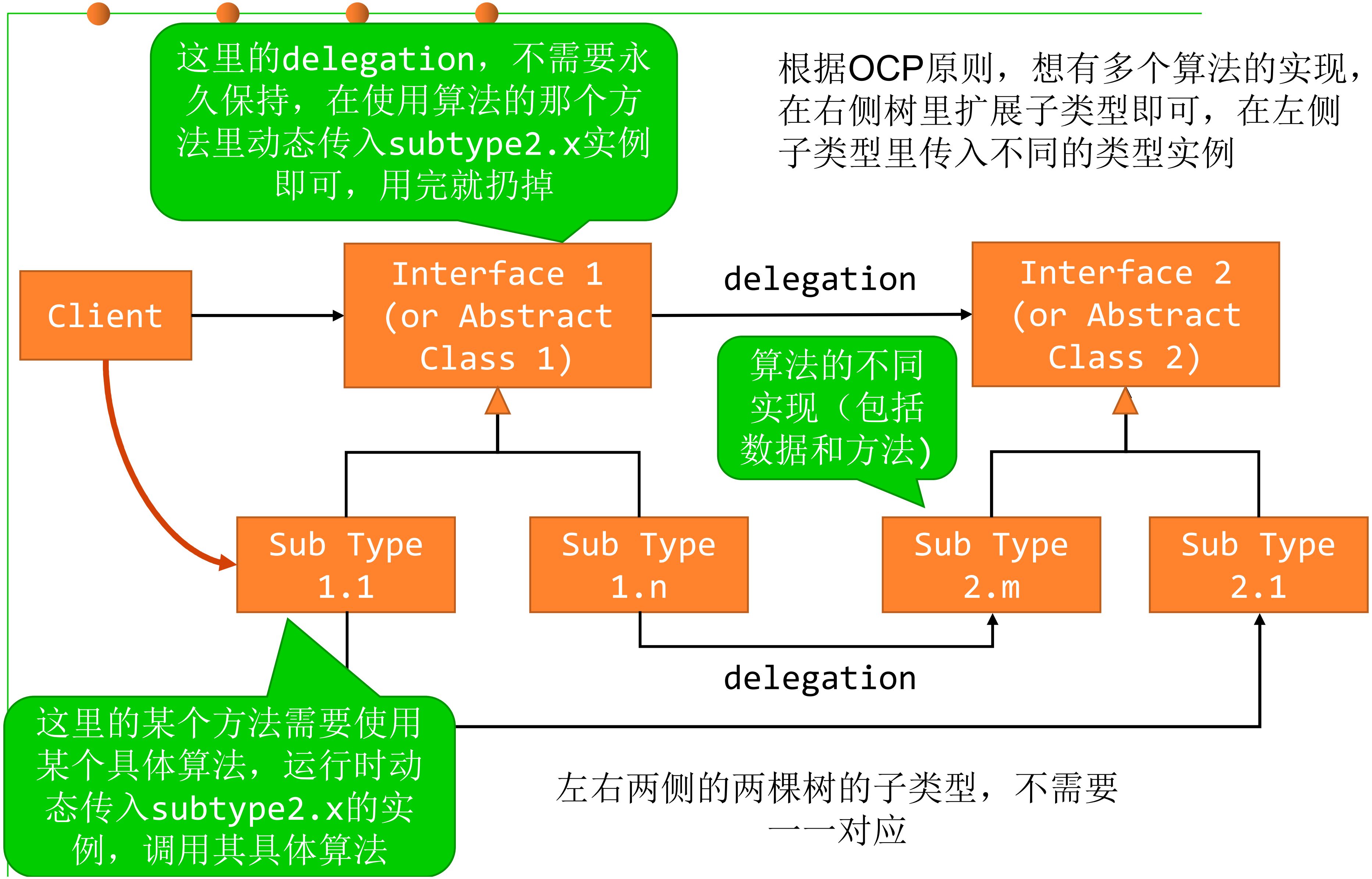
每个子类型，只需要实现上面的各个abstract方法即可。

设计模式的对比：共性样式2

两棵“继承树”，两个层次的“delegation”



Strategy



这里的delegation，不需要永久保持，在使用算法的那个方法里动态传入subtype2.x实例即可，用完就扔掉

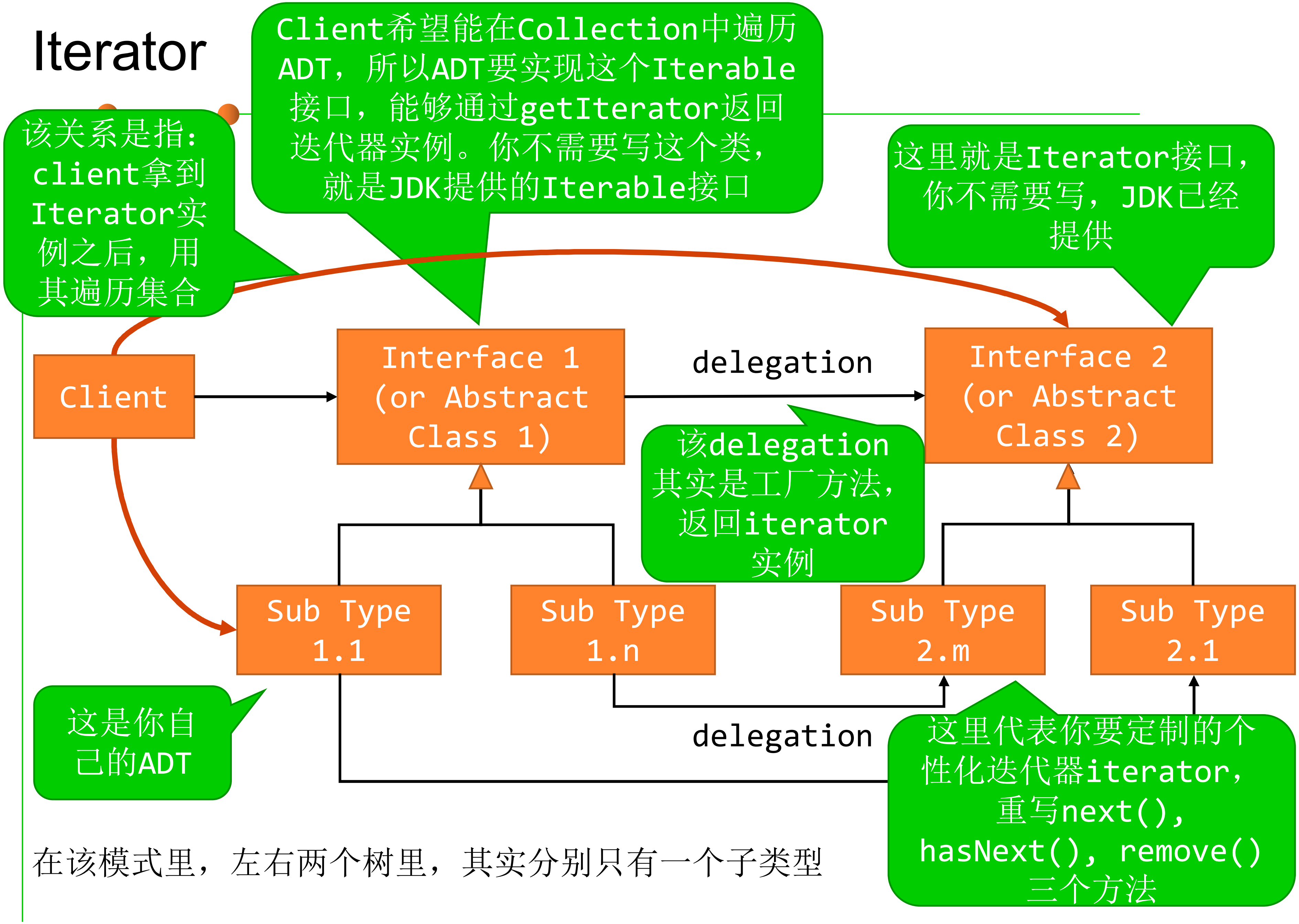
根据OCP原则，想有多个算法的实现，在右侧树里扩展子类型即可，在左侧子类型里传入不同的类型实例

算法的不同实现（包括数据和方法）

这里的某个方法需要使用某个具体算法，运行时动态传入subtype2.x的实例，调用其具体算法

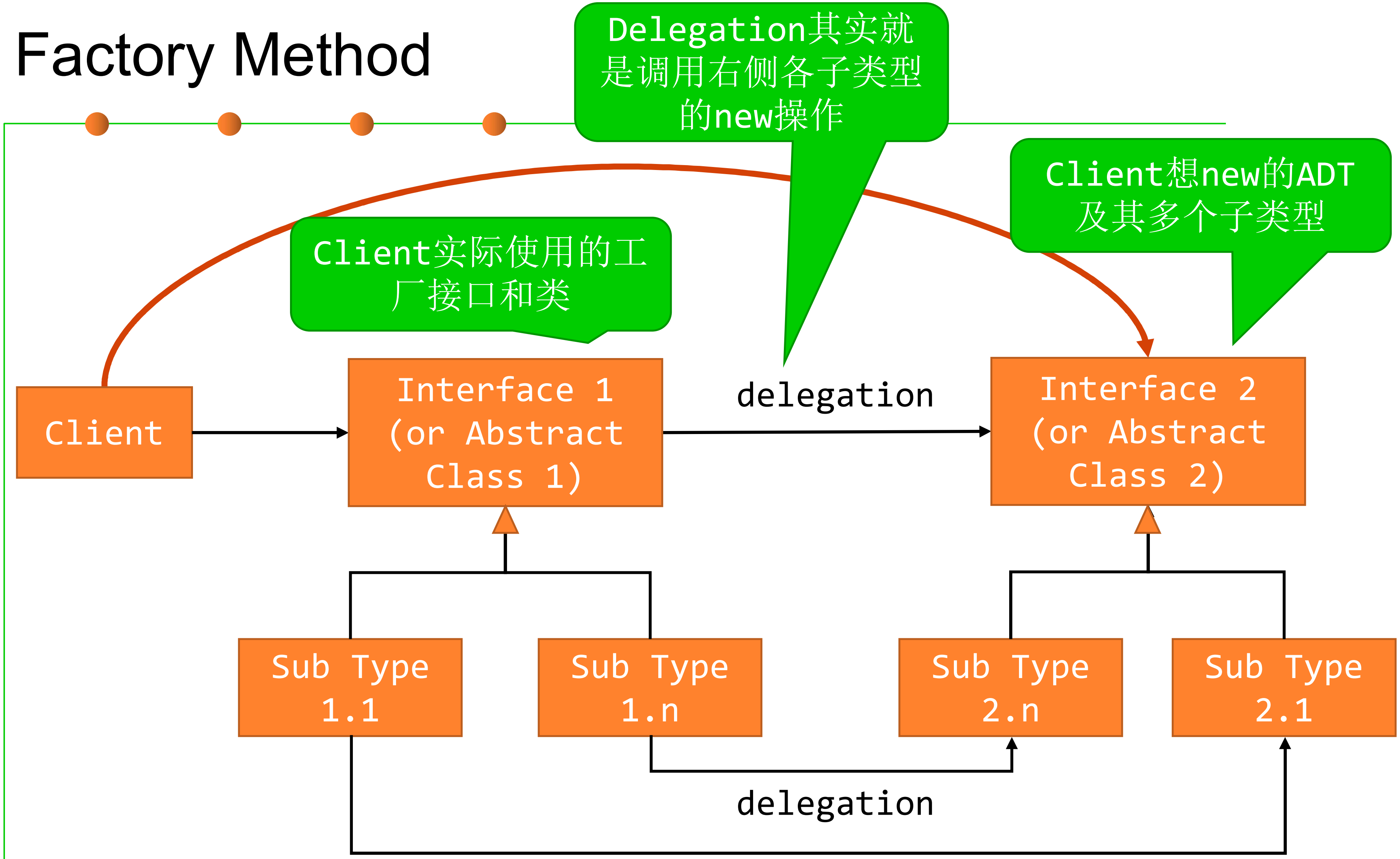
左右两侧的两棵树的子类型，不需要一一对应

Iterator


Client希望能在Collection中遍历ADT，所以ADT要实现这个Iterable接口，能够通过getIterator返回迭代器实例。你不需要写这个类，就是JDK提供的Iterable接口
该关系是指：client拿到Iterator实例之后，用其遍历集合
这里就是Iterator接口，你不需要写，JDK已经提供
Client
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
该delegation其实是工厂方法，返回iterator实例
Sub Type 1.1
Sub Type 1.n
Sub Type 2.m
Sub Type 2.1
这是你自己的ADT
delegation
这里代表你要定制的个性化迭代器iterator，重写next()，hasNext()，remove()三个方法

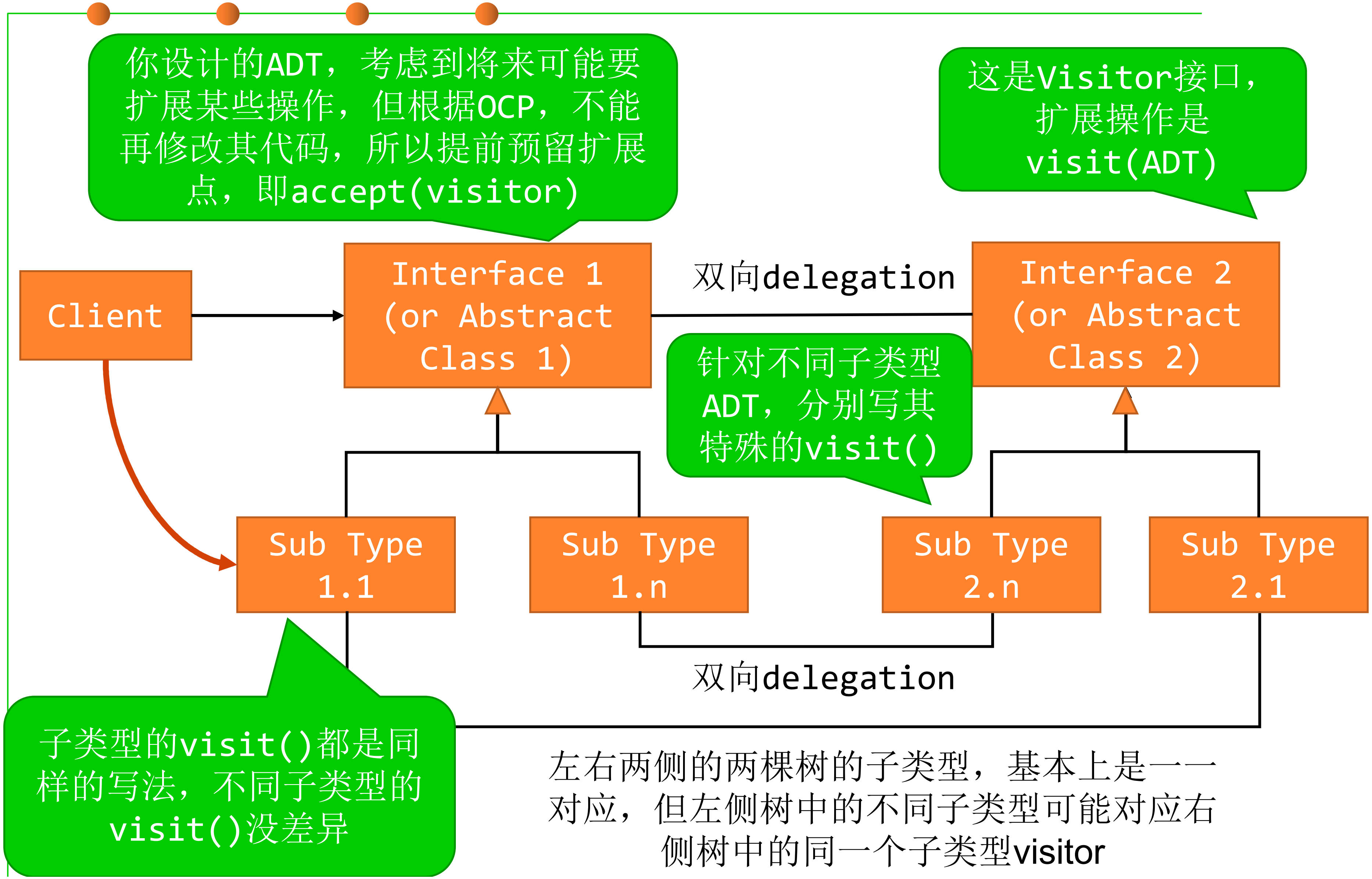
在该模式里，左右两个树里，其实分别只有一个子类型

Factory Method


Delegation其实就是调用右侧各子类型的new操作
Client想new的ADT及其多个子类型
Client实际使用的工厂接口和类
Client
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
Sub Type 1.1
Sub Type 1.n
Sub Type 2.n
Sub Type 2.1
delegation

左右两棵树的子类型一一对应。如果在工厂方法里使用type表征右侧的子类型，那么左侧的子类型只要1个即可。

Visitor


你设计的ADT，考虑到将来可能要扩展某些操作，但根据OCP，不能再修改其代码，所以提前预留扩展点，即accept(visitor)
这是Visitor接口，扩展操作是visit(ADT)
Client
Interface 1 (or Abstract Class 1)
双向delegation
Interface 2 (or Abstract Class 2)
针对不同子类型ADT，分别写其特殊的visit()
Sub Type 1.1
Sub Type 1.n
Sub Type 2.n
Sub Type 2.1
双向delegation
子类型的visit()都是同样的写法，不同子类型的visit()没差异

左右两侧的两棵树的子类型，基本上是一一对应，但左侧树中的不同子类型可能对应右侧树中的同一个子类型visitor

Summary

Summary

- **Creational patterns**
  - Factory method
- **Structural patterns**
  - Adapter
  - Decorator
- **Behavioral patterns**
  - Strategy
  - Template method
  - Iterator
  - Visitor

The end

June 21, 2021