5.2 Design Patterns for Maintainability

面向可维护性的设计模式

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂，Ex

目录

5.2.1 Creational patterns(创建模式) 1
- Factory Method pattern（工厂方法模式或者成为传统工厂模式） 2
- 抽象工厂模式 5

5.2.2 Structural patterns（结构模式） 7
- Proxy（代理模式） 7

5.2.3 Behavioral patterns（行为模式） 9
- Observer（观察者模式） 9
- Visitor（访问者模式） 11

5.2.4 关于 5.2 以及 4.3 中设计模式的总结 14
- 共性样式 1: 14
- 此类的模式: 15
  - Adaptor(装配器模式)（有点牵强） 15
  - Proxy(代理模式) 15
  - Template（模板模式） 16
- 共性样式 2: 16
- 此类的模式: 17
  - 桥接模式 17
  - 拓展：桥接模式。 17
  - 迭代器模式 18
  - 工厂模式 18
  - 抽象工厂模式 19
  - 观察者模式: 20
  - Visitor 模式 20

5.2.1 Creational patterns(创建模式)

关于如何“创建类的新实例”的模式

Factory Method pattern（工厂方法模式或者成为传统工厂模式）

适用场合：

为了满足表示独立性，不想使得client中得到某个接口的具体实现方法，避免表示暴露。例如，List 接口在 client 中初始化为 linklist 或者 arraylist 就造成了表示暴露。

形式化定义：

Client 需要初始化一个接口 product 类型的变量，该接口有两个实现类 productone，producttwo……，但是不希望 client 直接调用这两个实现类，避免表示暴露。

解决办法：

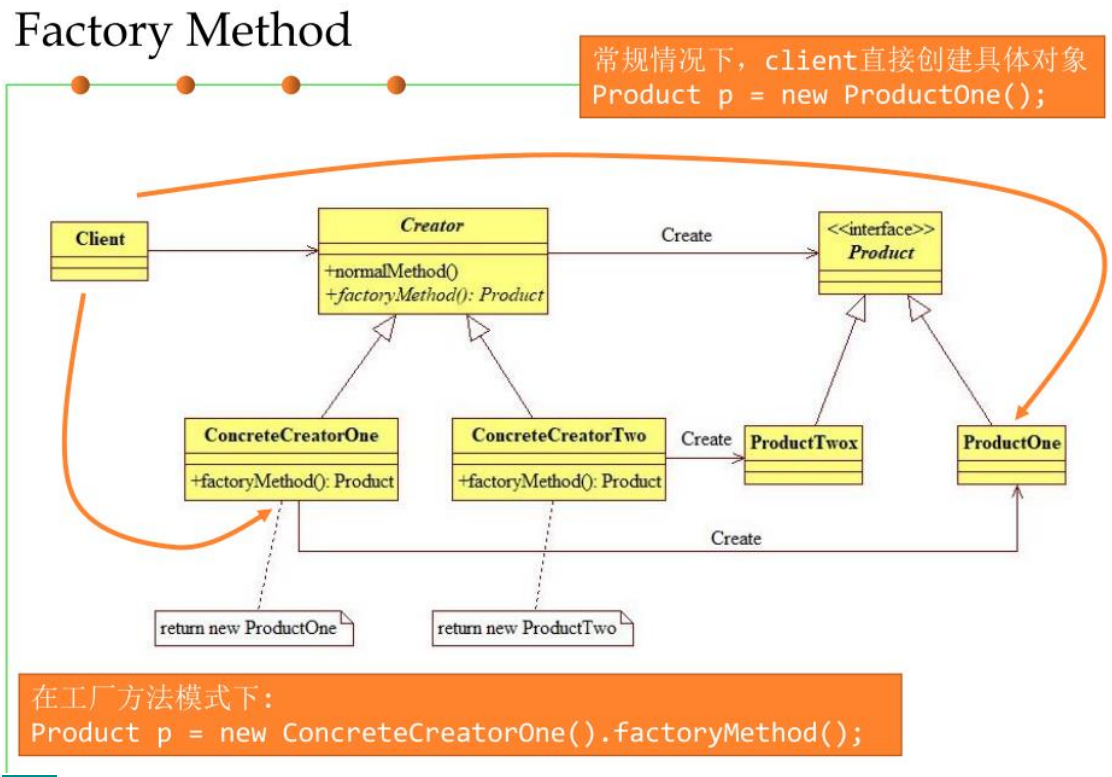
构造一个接口 Creator，用于创建 product 的具体对象，构造 Creator 的具体实现类 ConcreteCreatorone，ConcreteCreatortwo……分别对应于 Product 的具体实现类 profuctone，producttwo……，在每个 Creator 的具体实现类中，存在方法可以返回类型为接口 Product 的具体实现类的对象。

缺点：

这种工厂方法模式是十分有利于保护表示独立性，但是增加了的代码的复杂度。

每增加一种产品就需要增加一个新的工厂子类。

形象图：



Ex：

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 );
}
```

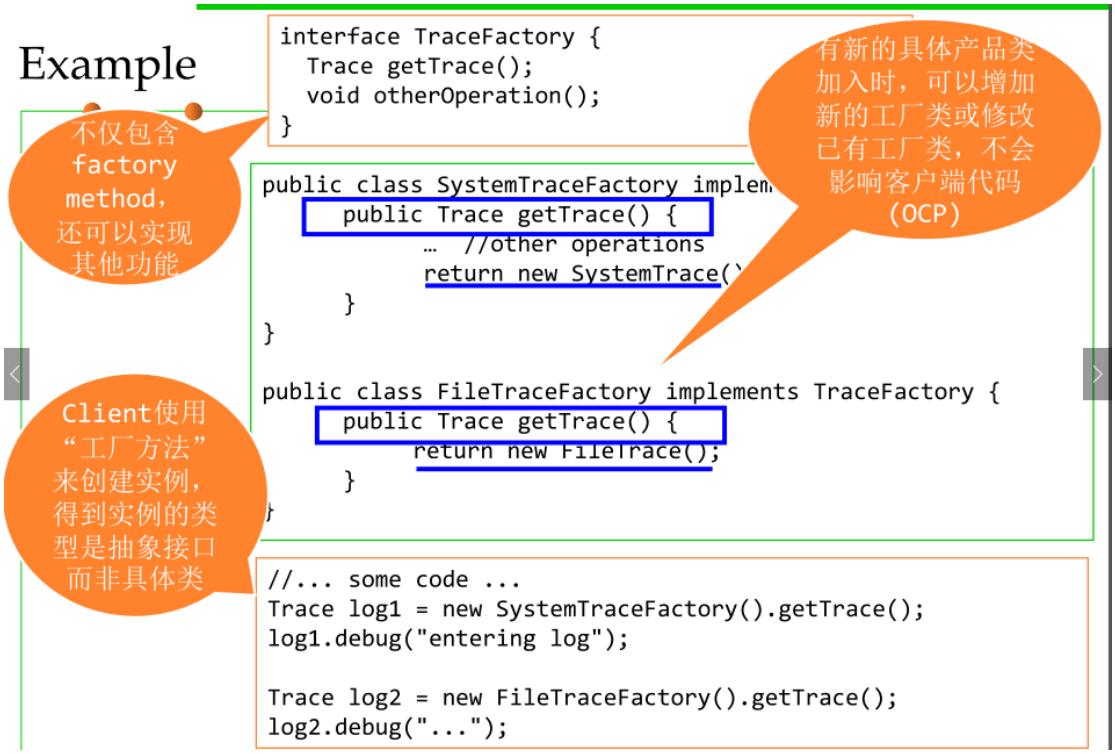
Concrete product 2

```
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 log1 = new SystemTrace();
log1.debug("entering log");

Trace log2 = new FileTrace();
log2.debug("... ");
```

The client code is tightly coupled with concrete products.




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

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

另外一种实现方式：也可以根据类型决定创建哪个具体产品

```
Trace log = new Factory().getTrace("system");
log.setDebug(false);
log.debug("...");
```

在某些情况下，工厂类不需要创建实例，故而采用静态方法也可以。

Example

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

public class TraceFactory {
      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 = SystemTraceFactory.getTrace();
log1.setDebug(true);
log1.debug( "entering log" );

Trace log2 = TraceFactory.getTrace("system");
log1.setDebug(true);
log2.debug("... ");
```

相比于通过构造器(new)构建对象：

1. **静态工厂方法可具有指定的名称**
2. **不必在每次调用的时候都创建新对象**
3. **可以返回原返回类型的任意子类型**

抽象工厂模式

适用场合：

与传统工厂模式类似，但是抽象工厂模式是，提供接口以创建一组相关或相互依赖的对象，而不是传统工厂方法中的单个对象

形式化定义：

Client 需要初始多个接口类型不同的对象，例有：window，scrollbar……等接口类型，这些接口有多个实现类，例如：window 接口有实现类 PMwindow，MotifWindow，scrollbar 接口有实现类 PMscroll，MotifScrollBar。但是不希望 client 直接构造，避免表示暴露。

解决办法：

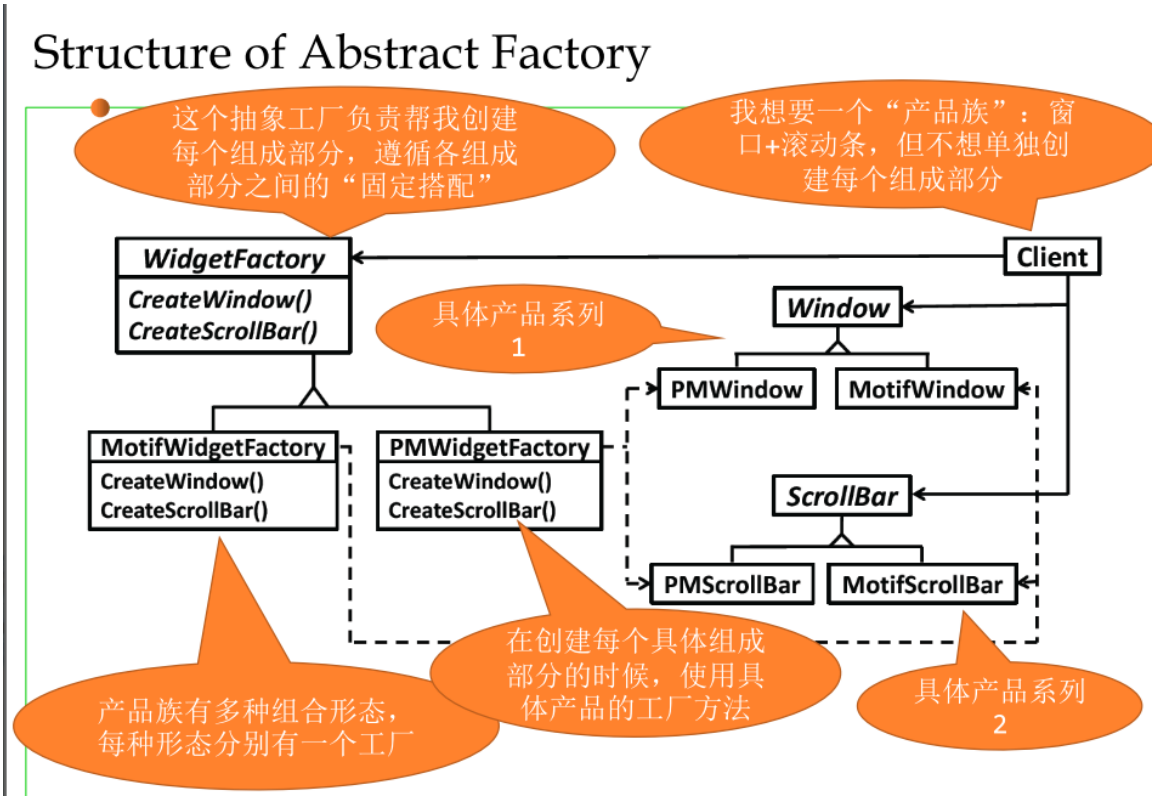
构造一个接口 widgetFactory，用于创建 window 以及 scrollbar 等接口的具体对象，构造 widgetFactory 的具体实现类 PMwidgetFactory，MptifwidgetFactory 等等，分别对应于 window 和 scrolbar 的具体实现类的组合，在每个 widgetFactory 的具体实现类中，存在方法可以返回 client 需要的具体实现类的对象。

注：

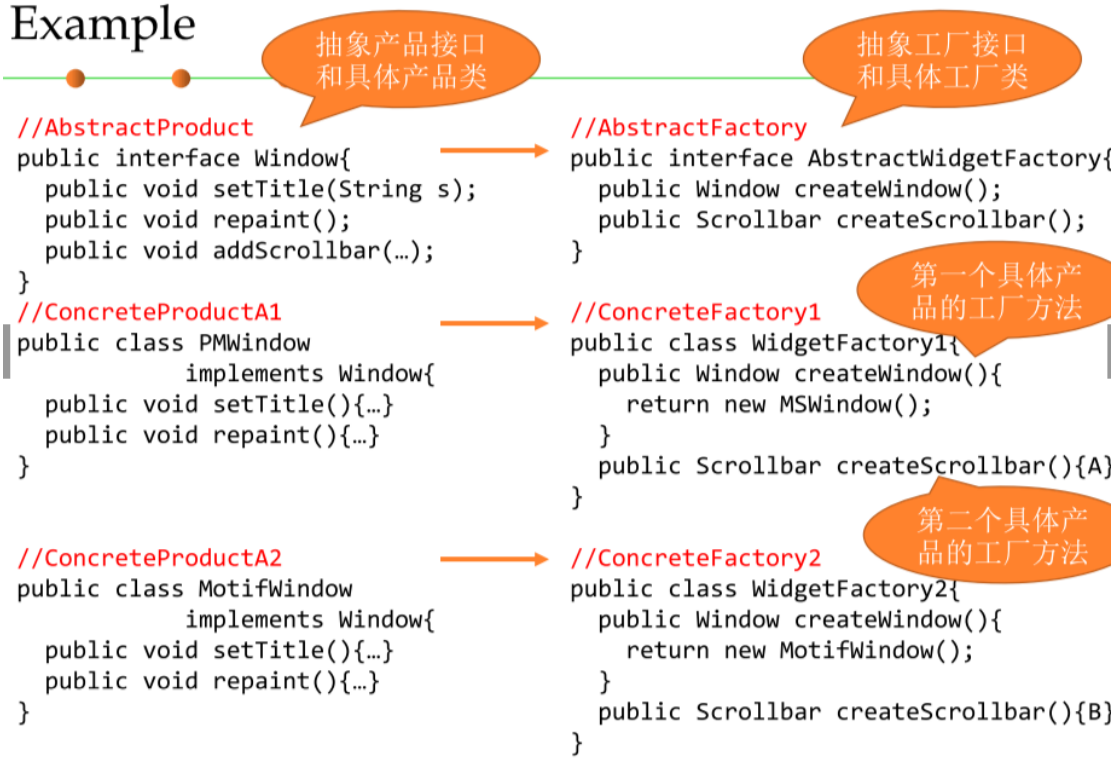
抽象工厂模式与传统工厂模式的区别在于构造一组对象还是一个对象。

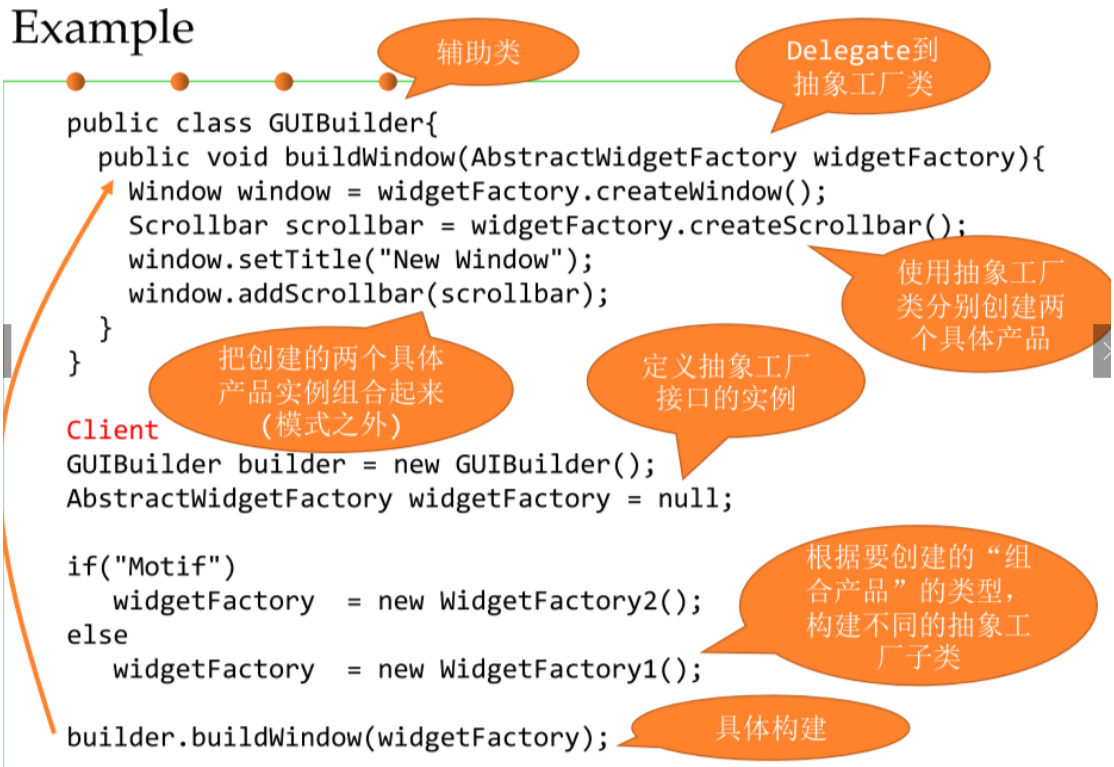
当然你可以认为，可以使用一组传统工厂模式的类来实现一个抽象工厂模式的类，但是这仍然有所不同，因为可能实现接口的某些类具有特定的联系，存在某些不可以共同构建，存在某些必须共同构建。一旦出现这种问题，抽象模式会很好的提供这种联系，而传统工厂模式则需要依赖于 client 的对于规约的遵守。

形象图：


Structure of Abstract Factory
这个抽象工厂负责帮我创建每个组成部分，遵循各组成部分之间的“固定搭配”
我想要一个“产品族”：窗口+滚动条，但不想单独创建每个组成部分
WidgetFactory
CreateWindow()
CreateScrollBar()
Client
具体产品系列1
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateWindow()
CreateScrollBar()
PMWidgetFactory
CreateWindow()
CreateScrollBar()
ScrollBar
PMScrollBar
MotifScrollBar
产品族有多种组合形态，每种形态分别有一个工厂
在创建每个具体组成部分的时候，使用具体产品的工厂方法
具体产品系列2

Ex:


Example
抽象产品接口和具体产品类
抽象工厂接口和具体工厂类
//AbstractProduct
public interface Window{
public void setTitle(String s);
public void repaint();
public void addScrollbar(…);
}
//AbstractFactory
public interface AbstractWidgetFactory{
public Window createWindow();
public Scrollbar createScrollbar();
}
第一个具体产品的工厂方法
//ConcreteProductA1
public class PMWindow
implements Window{
public void setTitle(){…}
public void repaint(){…}
}
//ConcreteFactory1
public class WidgetFactory1{
public Window createWindow(){
return new MSWindow();
}
public Scrollbar createScrollbar(){A}
}
第二个具体产品的工厂方法
//ConcreteProductA2
public class MotifWindow
implements Window{
public void setTitle(){…}
public void repaint(){…}
}
//ConcreteFactory2
public class WidgetFactory2{
public Window createWindow(){
return new MotifWindow();
}
public Scrollbar createScrollbar(){B}
}



注：在该例子中，使用了比抽象工厂模式更为高级的 builder 工厂模式，其将工厂类的方法进行整合。

5.2.2 Structural patterns（结构模式）

Proxy（代理模式）

适用场合：

某个对象存在某些情况例如（对象创建的开销很大，或者某些操作需要安全控制，或者需要进程外的访问），不希望直接被 client 访问。

注：无法理解代理模式如何降低对象创建的开销。

形式化定义：

由于某些问题，不希望 client 直接访问到 realsubject 类。

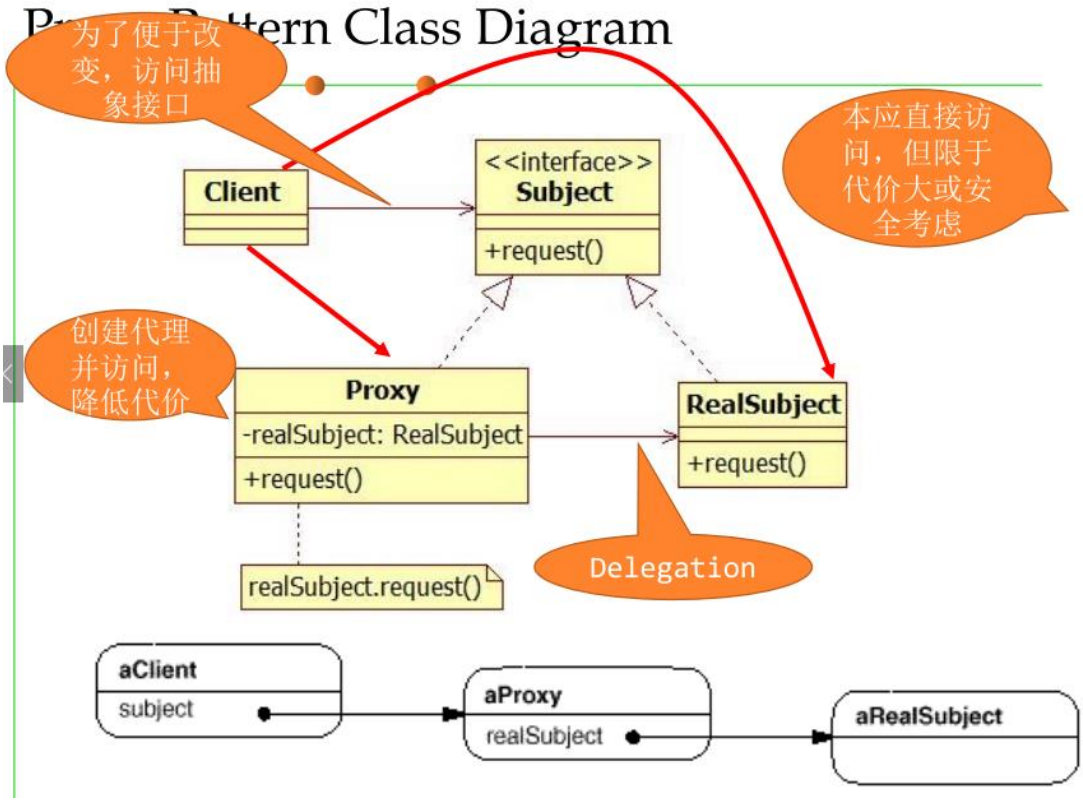
解决办法：

构造一个接口 subject，其中包含所有的 client 希望调用的 realsubject 的方法签名以及规约，创建一个 subject 的实现类 Proxy，在该实现类中，其对于方法的实现委托于 realsubject。

Client 只需要定义一个 subject 的变量，以 Proxy 进行初始化，调用 Proxy 中的接口即可。

形象图：

Pattern Class Diagram



Ex: 这里可能算是一个减少开销的问题，但是如果只是某个方法开销大，创建开销并不是 很大，何必使用代理，如果将这种形式应用于多个开销大的方法，如果 client 调用这些方法，岂不又是得不偿失，造成更多次对象的创建，而对象创建的开销很大。

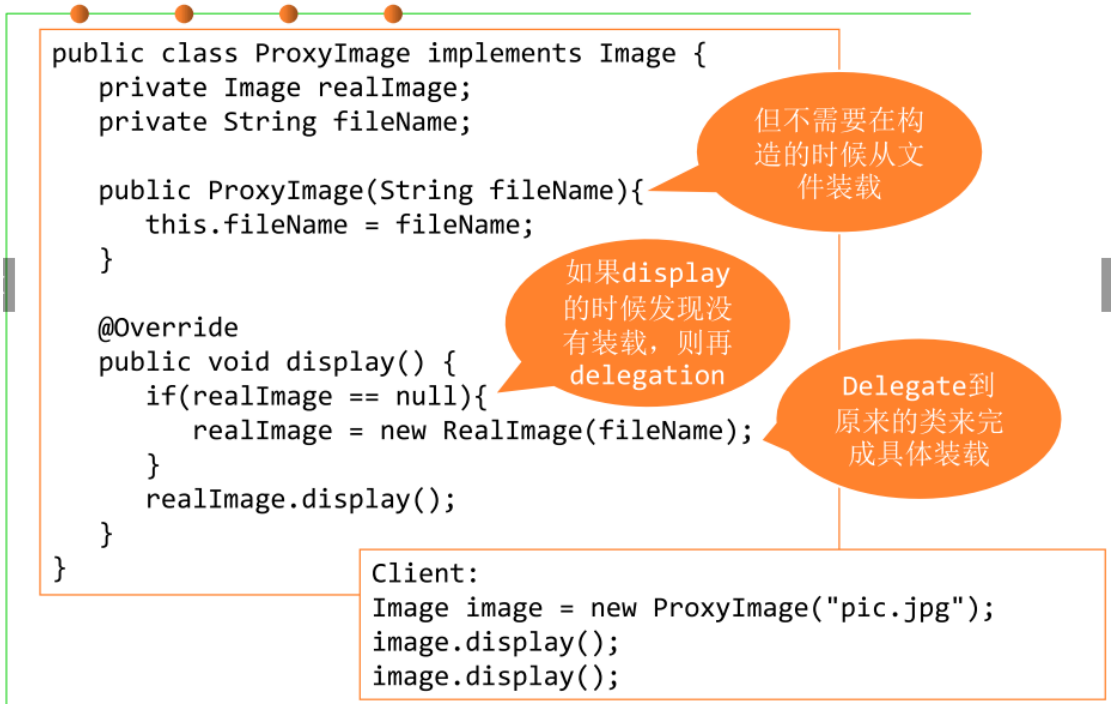
Example

```
public interface Image {
   void display();
}
```

```
public class RealImage implements Image {

   private String fileName;
   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {…}

   private void loadFromDisk(String fileName){…}
}
```

每次创建都要从磁盘装载，代价高

Example



5.2.3 Behavioral patterns（行为模式）

Observer（观察者模式）

适用场合：

在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。

形式化定义：

当 ConcreteSubject 类发生状态变化时，需要 CooncreteObserverA, CooncreteObserverB……都得到通知。

解决办法：

观察者模式的主要角色如下。

抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

具体主题（Concrete Subject）角色：也叫具体目标类。。

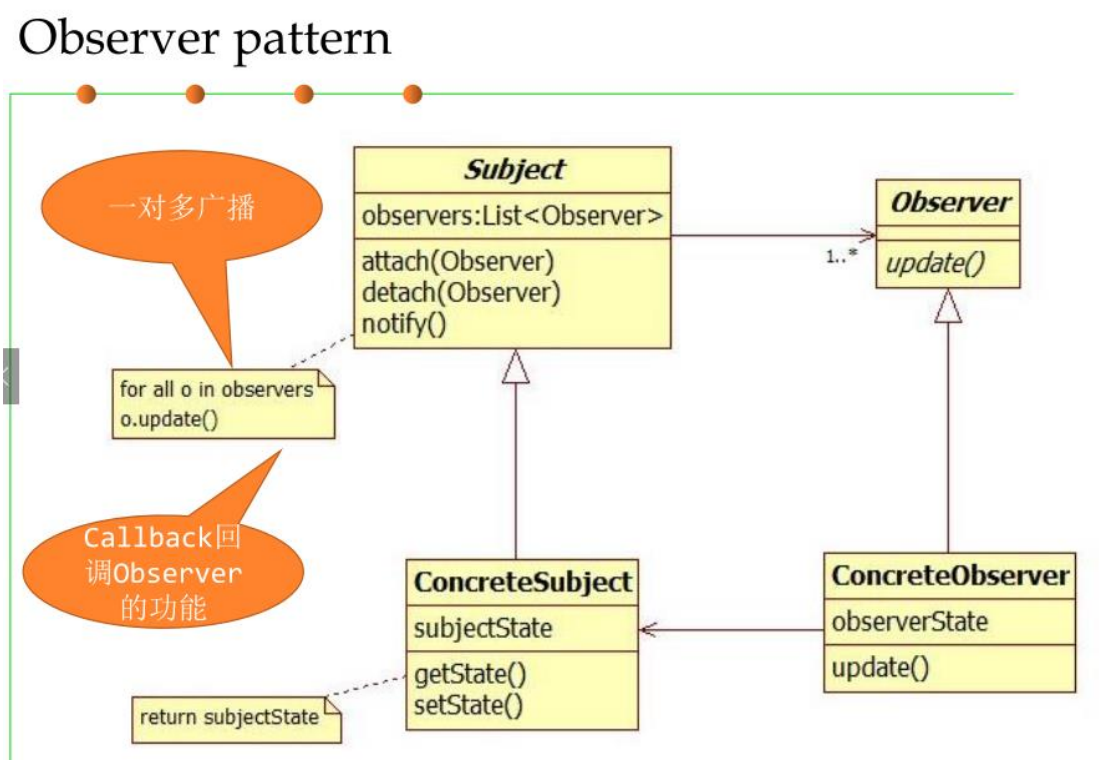
抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法。

优点和缺点：

而且在客户端中，完全可以使用接口来作为被观察者变量的类型，这样就在某种程度上实现了抽象耦合，耦合程度低。

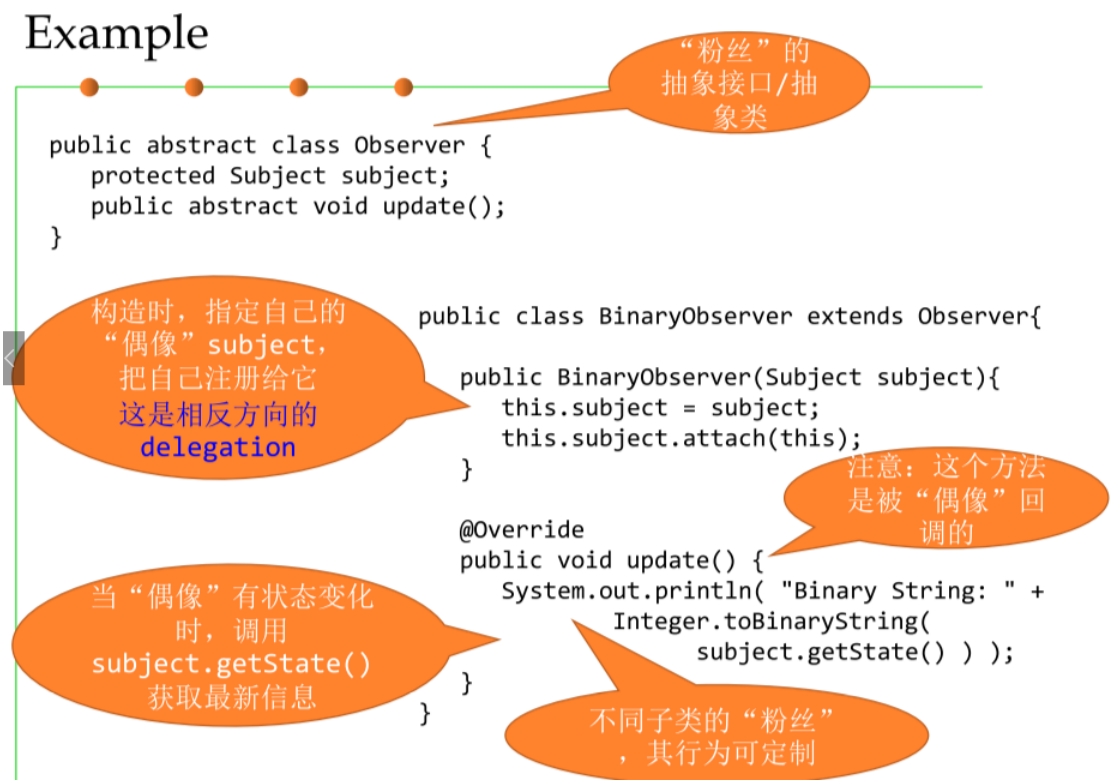
形象图:


Observer pattern
一对多广播
Subject
observers:List<Observer>
attach(Observer)
detach(Observer)
notify()
Observer
update()
1..*
for all o in observers
o.update()
Callback回调Observer的功能
ConcreteSubject
subjectState
getState()
setState()
return subjectState
ConcreteObserver
observerState
update()

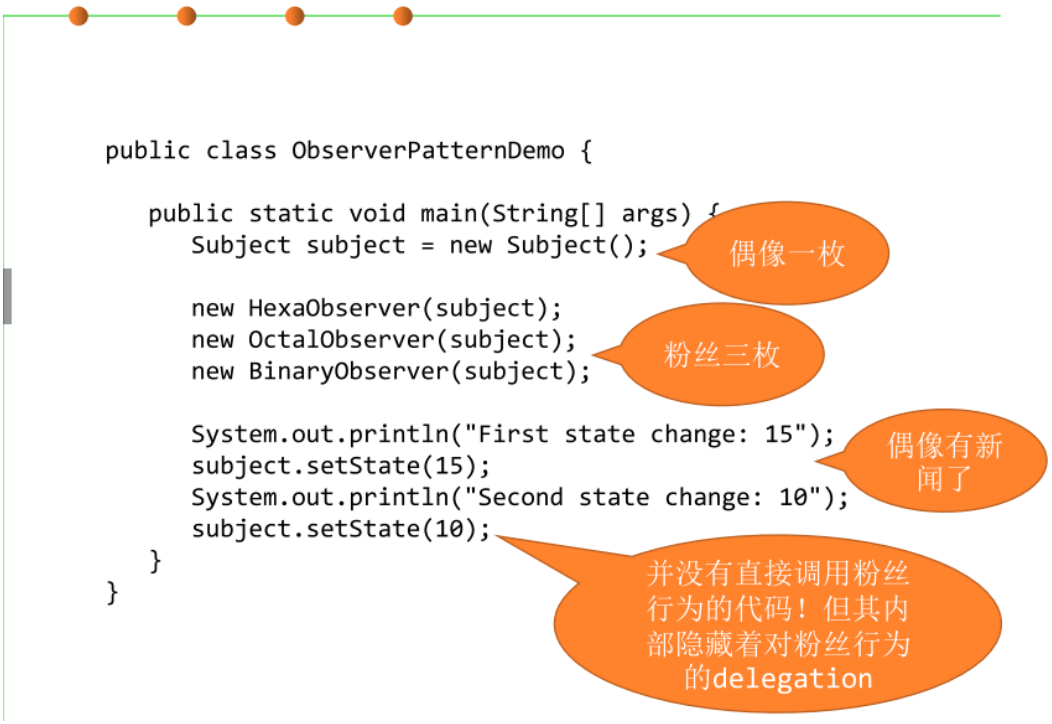
Ex:


5.2 Design Patterns for Maintainability
Example
此例中只有一个具体Subject类，没有设计抽象Subject接口
维持一组“对自己感兴趣的”对象
public class Subject {
private List<Observer> observers = new ArrayList<Observer>();
private int state;
public int getState() {return state;}
public void setState(int state) {
this.state = state;
notifyAllObservers();
}
在自己状态变化时，通知所有“粉丝”
public void attach(Observer observer){observers.add(observer);}
private void notifyAllObservers(){
for (Observer observer : observers) {
observer.update();
}
}
}
允许“粉丝”调用该方法向自己注册，将其加入队列，即建立delegation关系
callbck调用“粉丝”的update操作，向粉丝“广播”自己的变化，实际执行delegation

Example



Example



Visitor（访问者模式）

适用场合：

可以通过接口类来进行对多个类的数据的访问，进而定义在其上的操作，实现数据与作用于数据上的操作的分离。

也可以这么讲，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 本身的情况下在需要时通过 delegation 接入 ADT。这与上段话的意思是一致的。

形式化定义：

存在某个接口 Element，其有多种实现方式 ConcreteElementA, ConcreteElementB……

解决办法：

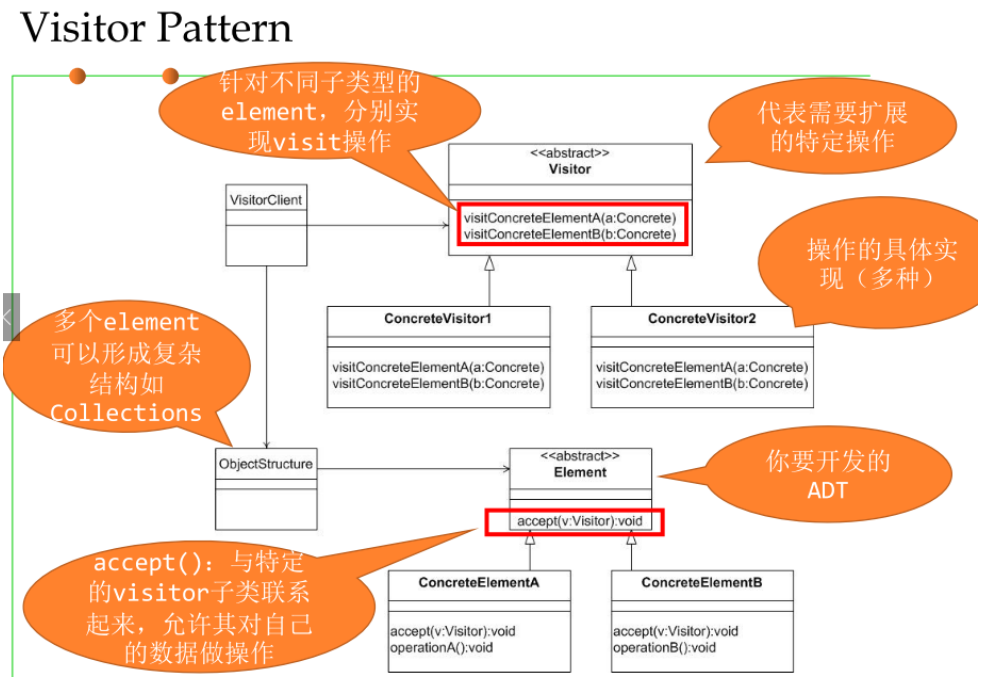
定义一个接口 visitor，其内存在对于每个接口 Element 的实现类的单独访问（这里可以使用方法重载，以保证签名相同），另外，在接口 Element 中定义方法 accept（visitor），使得通过该方法可以访问其内元素。

客户端可以定义接口 visitor 的实现类，对接口 Element 内进行数据访问，定义个人需要的方法，进行解绑。

注：存在双向委托，一个为 accpt（），一个为简单的调用

注：这种访问模式类似于黑盒框架，即利用了一个接口来实现访问内部元素。

形象图：



Ex：

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);
  }
}
```

声明自己是可以被visit的

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

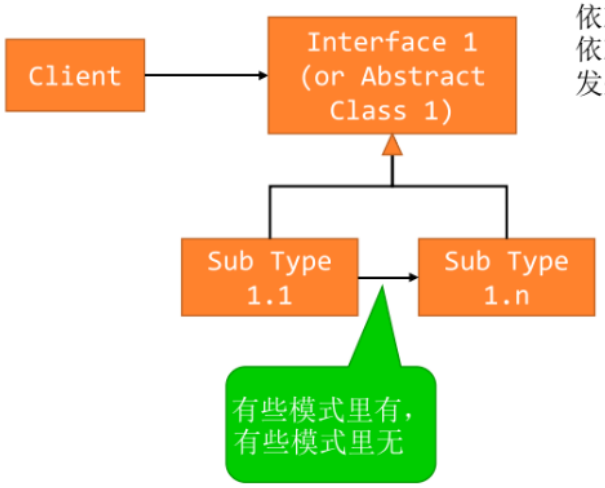
```
public class ShoppingCartClient {

   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

5.2.4 关于 5.2 以及 4.3 中设计模式的总结

共性样式 1:

单继承树，匹配 OCP 原则或者 DIP 原则

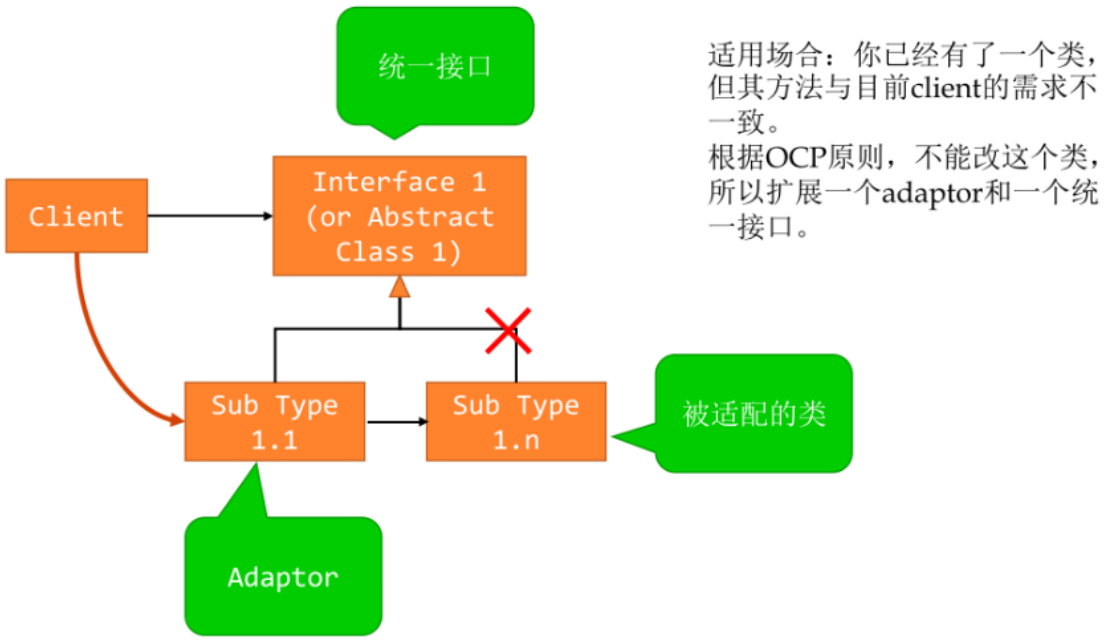


只使用“继承”，不使用“delegation”
核心思路：OCP/DIP
依赖倒置，客户端只依赖“抽象”，不能依赖于“具体”
发生变化时最好是“扩展”而不是“修改”

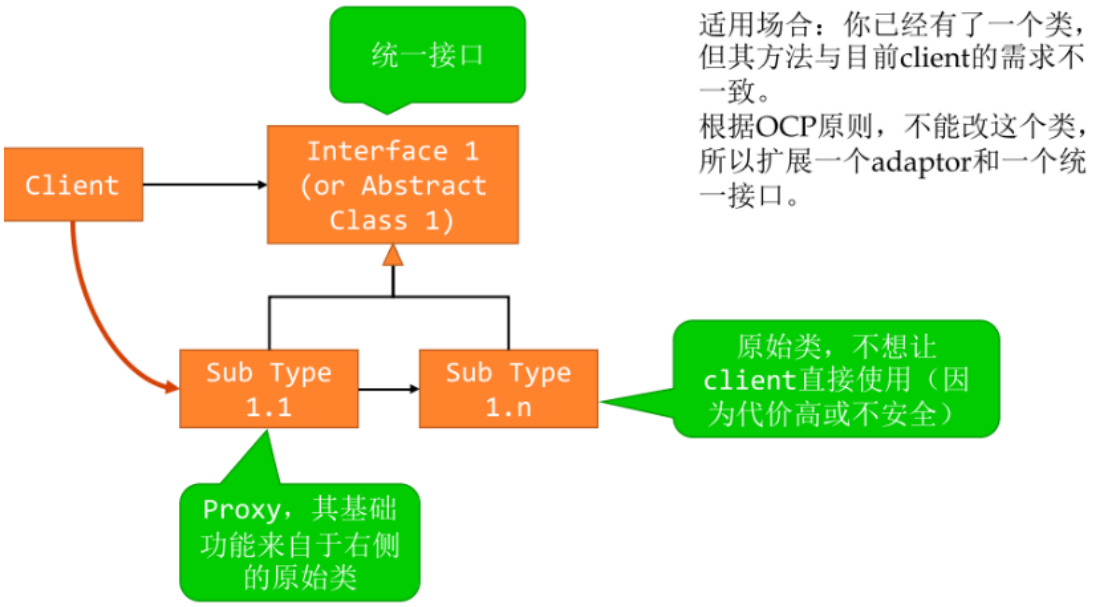
注：虽然此图片中解释：共性样式 1 只使用继承而不是用委托。但代理模式属于该结构，使用了委托

此类的模式：

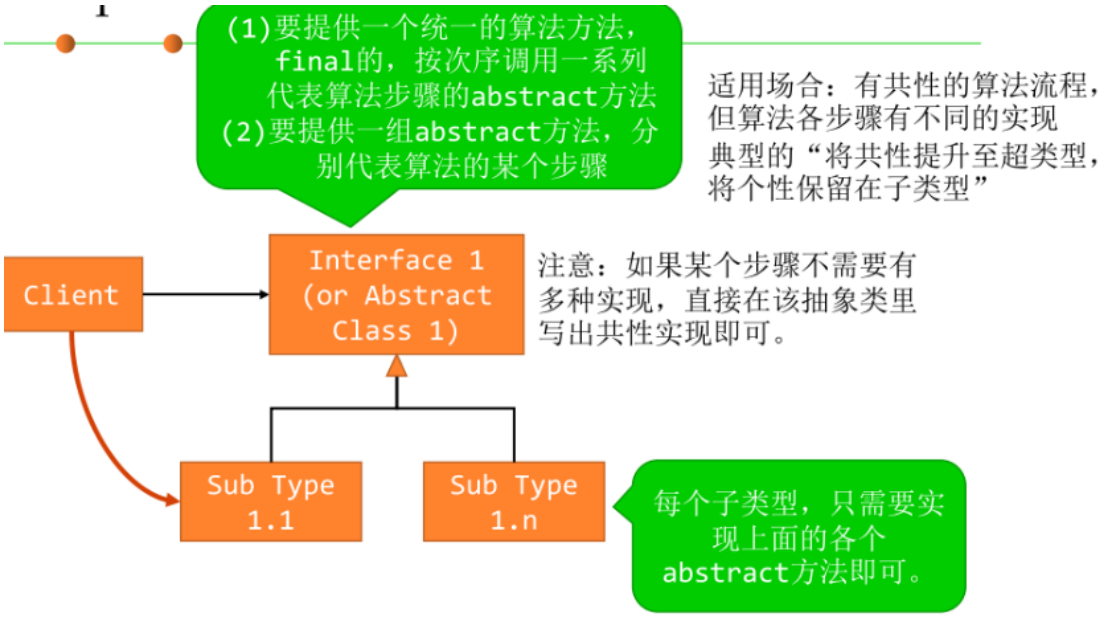
Adaptor(装配器模式)（有点牵强）



Proxy(代理模式)



Template（模板模式）

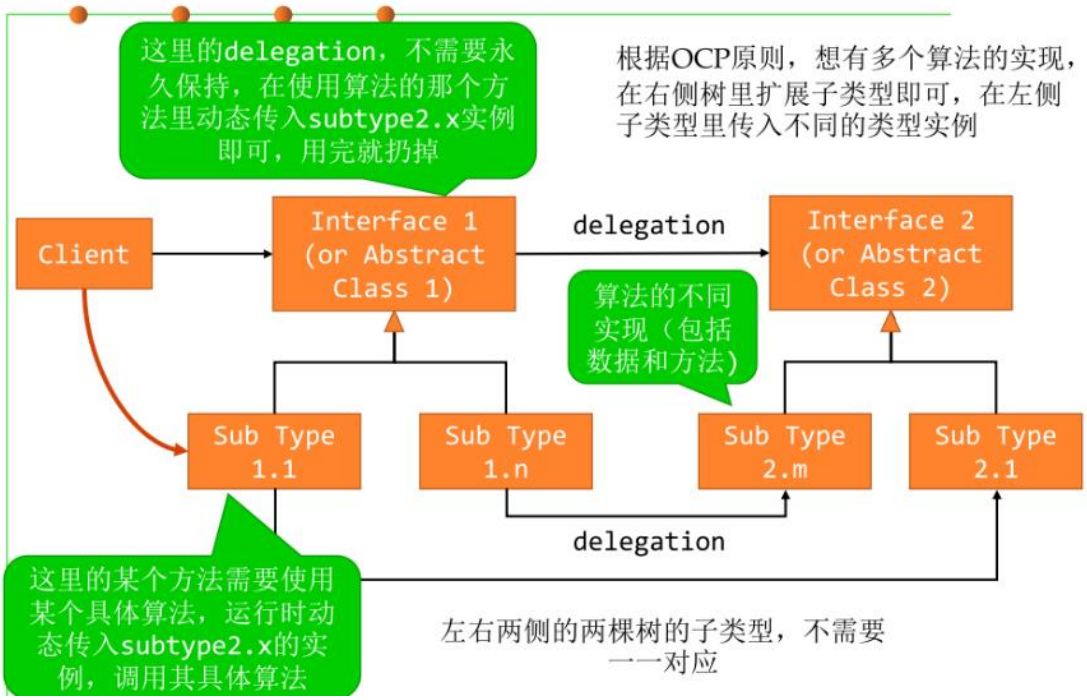


共性样式 2:

多继承树，两个层次的委托

此类的模式：

桥接模式



拓展：桥接模式。

桥接模式的类型极其类似于抽象工厂模式与策略模式的有机组合。

使用场合：

存在多个接口，每个接口有多种实现方式，clinet 会调用这些接口，其每个接口的实现方式存在限制，不可以随意变化。

形式化定义：

存在接口 interface1，其存在实现类 sybtype1.1，subtype1.2……存在与 interface1 相类似的接口 interface2，仅指结构，与功能无关。Client 需要有机调用 interface1，interface2

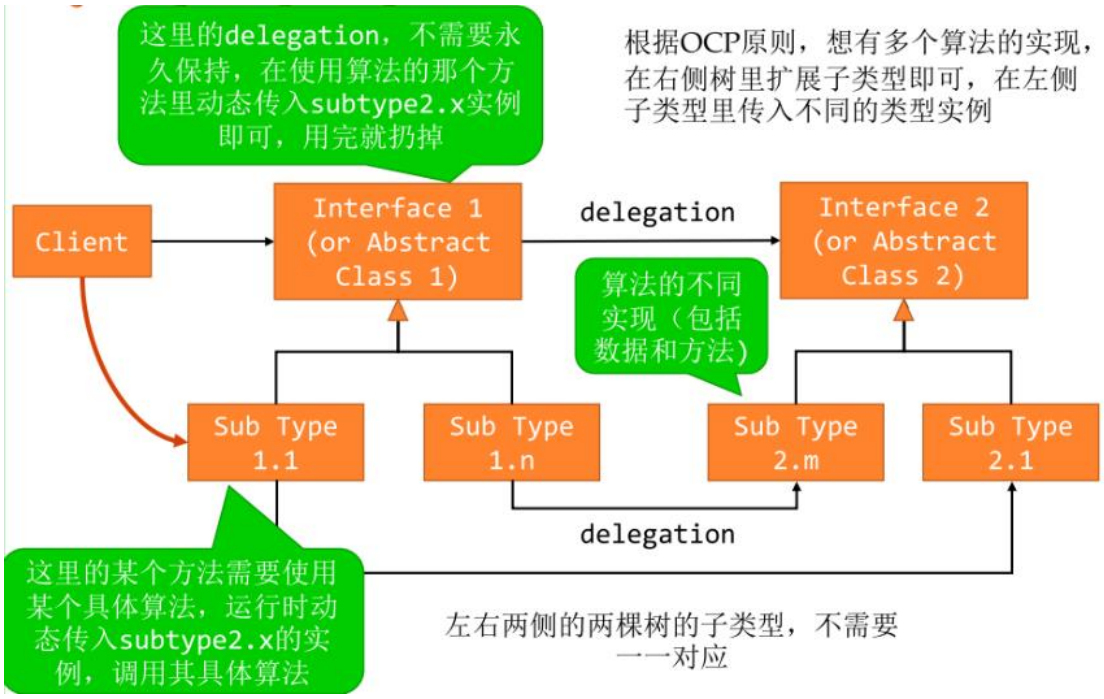
解决办法：

在 interface1 中添加某个方法，该方法委托于 interface2，实现 interface2 的功能。

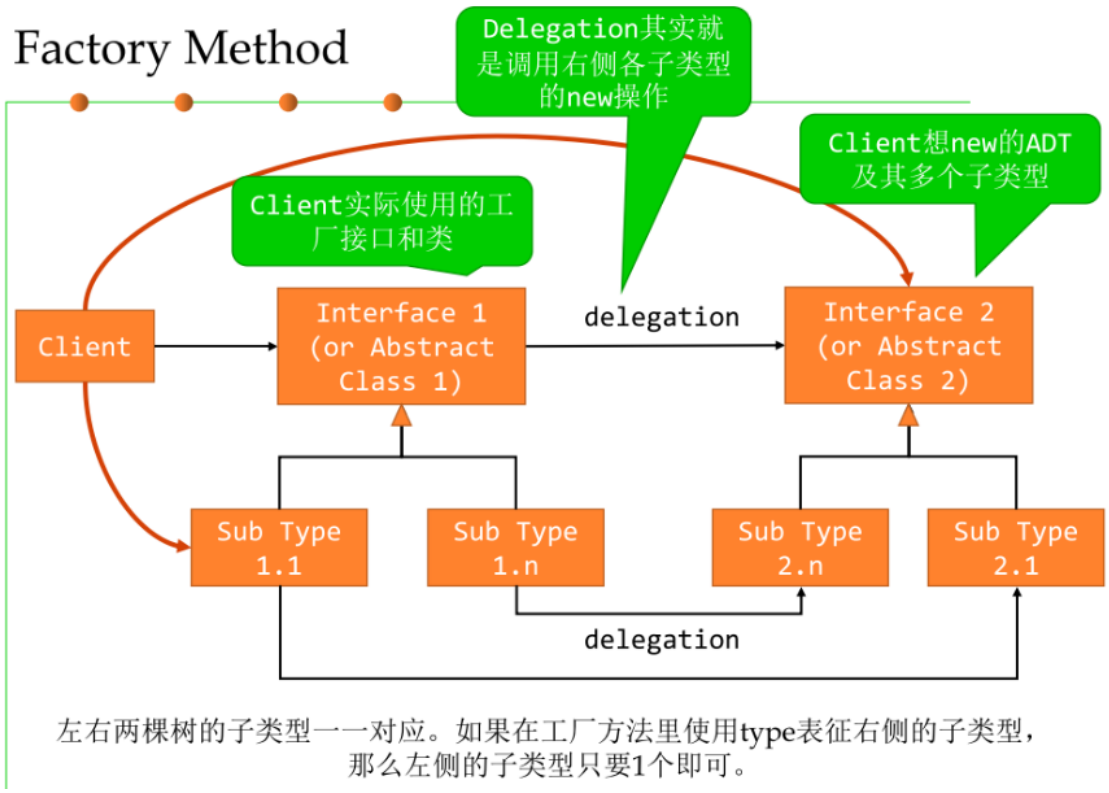
特点：

隔离了 interface2 与 cilent 的直接调用，而是使用 interface 作为桥梁。

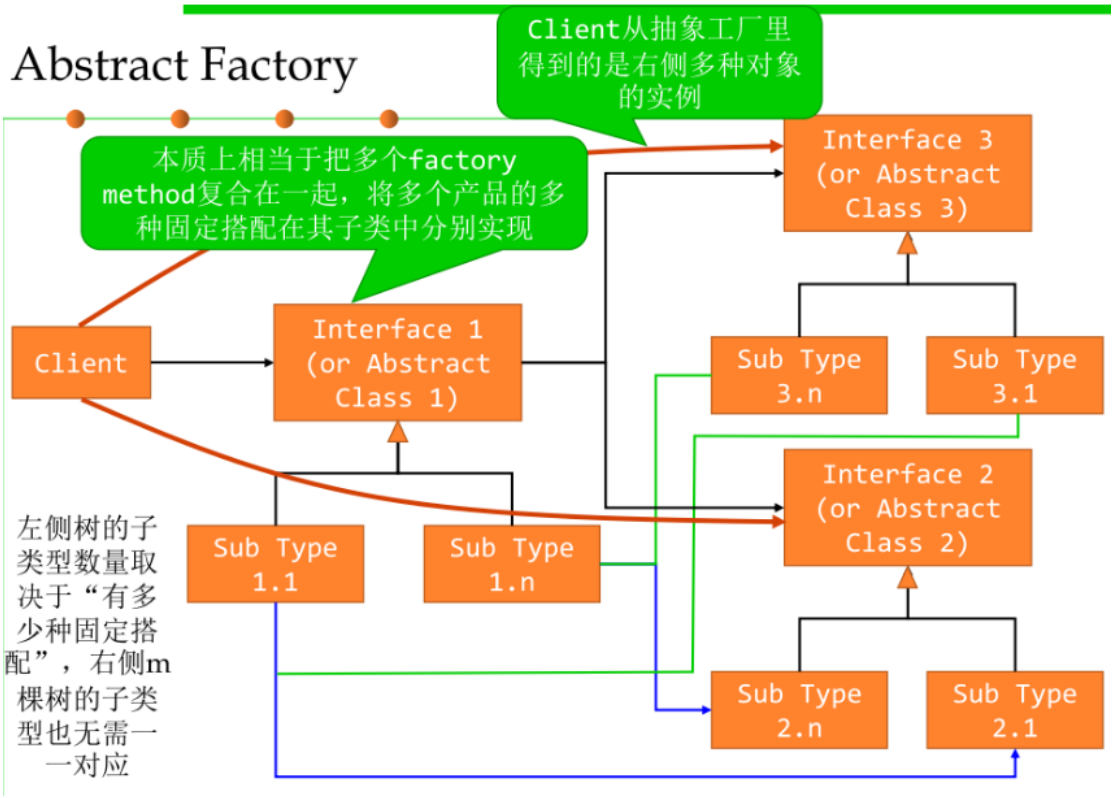
迭代器模式


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
Client
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
算法的不同实现（包括数据和方法）
Sub Type 1.1
Sub Type 1.n
Sub Type 2.m
Sub Type 2.1
delegation
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

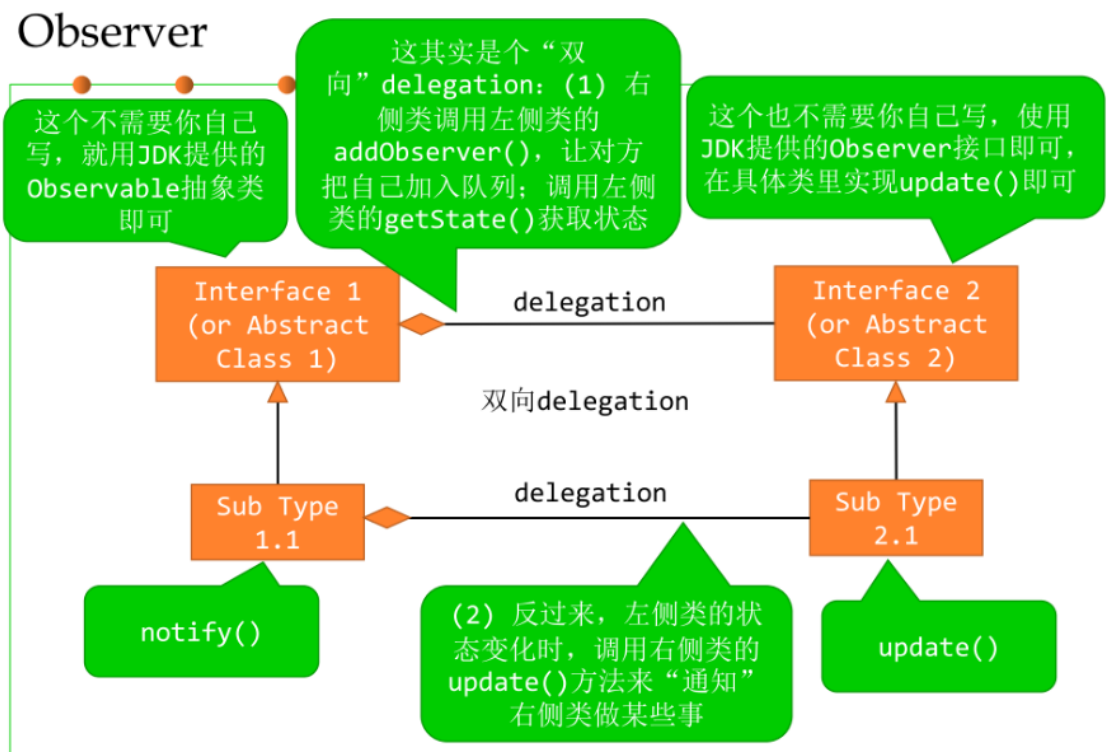
工厂模式


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。

抽象工厂模式


Abstract Factory
Client从抽象工厂里得到的是右侧多种对象的实例
本质上相当于把多个factory method复合在一起，将多个产品的多种固定搭配在其子类中分别实现
Client
Interface 1 (or Abstract Class 1)
Interface 3 (or Abstract Class 3)
Interface 2 (or Abstract Class 2)
Sub Type 1.1
Sub Type 1.n
Sub Type 3.n
Sub Type 3.1
Sub Type 2.n
Sub Type 2.1
左侧树的子类型数量取决于“有多少种固定搭配”，右侧m棵树的子类型也无需一一对应

观察者模式：



Visitor 模式