
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 5: Maintainability-Oriented Software Construction Approaches

# 5.2 Design Patterns for Maintainability

# 面向可维护性的设计模式

Wang Zhongjie
rainy@hit.edu.cn

April 20, 2020

# Outline

- **Creational patterns**
  - Factory method pattern creates objects without specifying the exact class to create.
  - Abstract factory pattern groups object factories that have a common theme.
- **Structural patterns**
  - Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.
- **Behavioral patterns**
  - Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
  - Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.
- **Commonality and Difference of Design Patterns** 设计模式的共性和差异

# Reading

- **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**


哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

# 1 Creational patterns

关于如何“创建类的新实例”的模式




# (1) 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.
  - Factory Method lets a class defer instantiation to subclasses.
- **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.

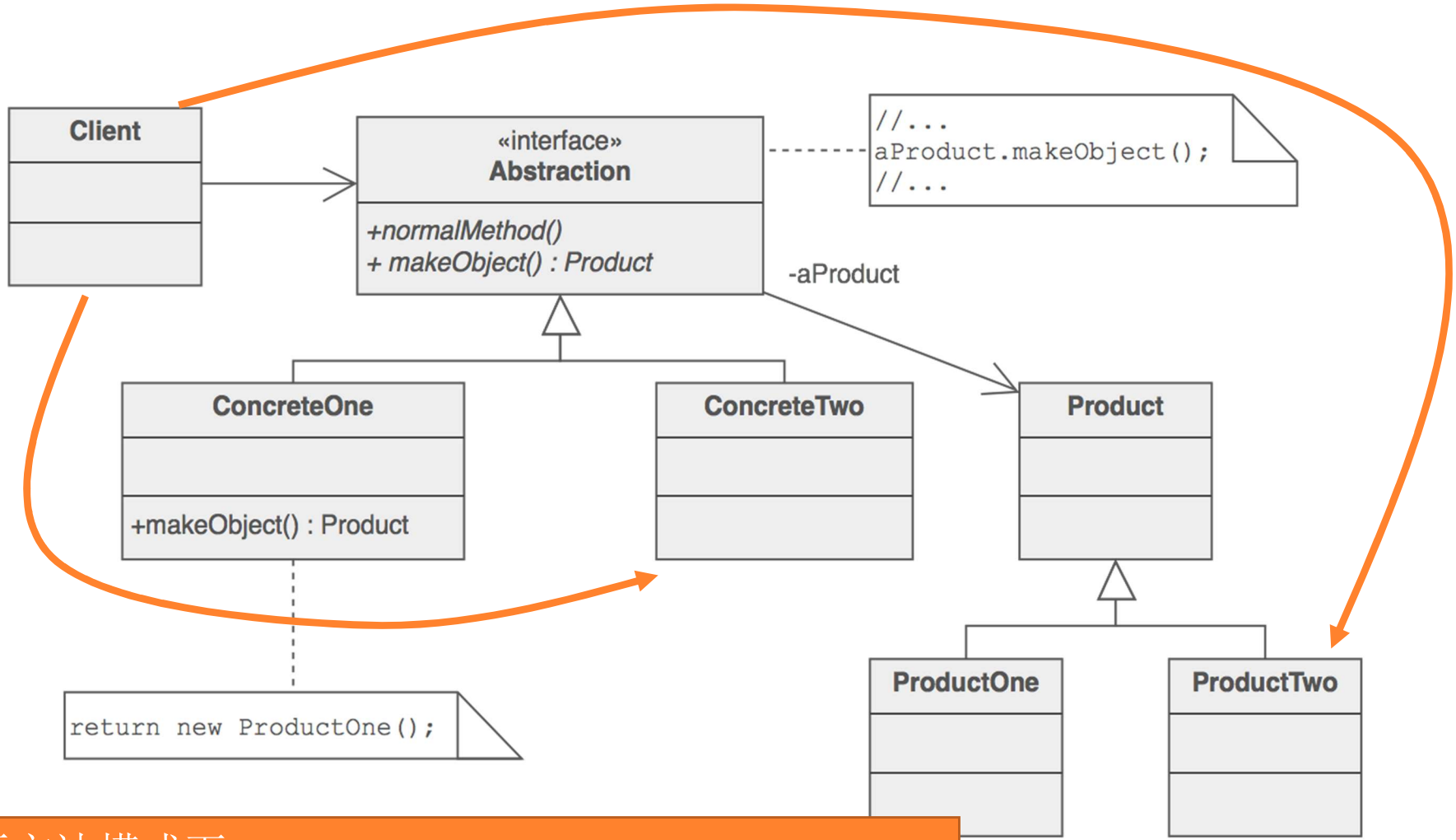
当client不知道要创建哪个具体类的实例，或者不想在client代码中指明要具体创建的实例时，用工厂方法。

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.

# Example

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

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

```
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();
    }
```

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

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

# 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 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**
  - Clients may have to make a subclass of the `Creator`, just so they can create a certain `ConcreteProduct`.
  - 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.

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




# (2) Abstract Factory

抽象工厂模式

# Abstract Factory Pattern

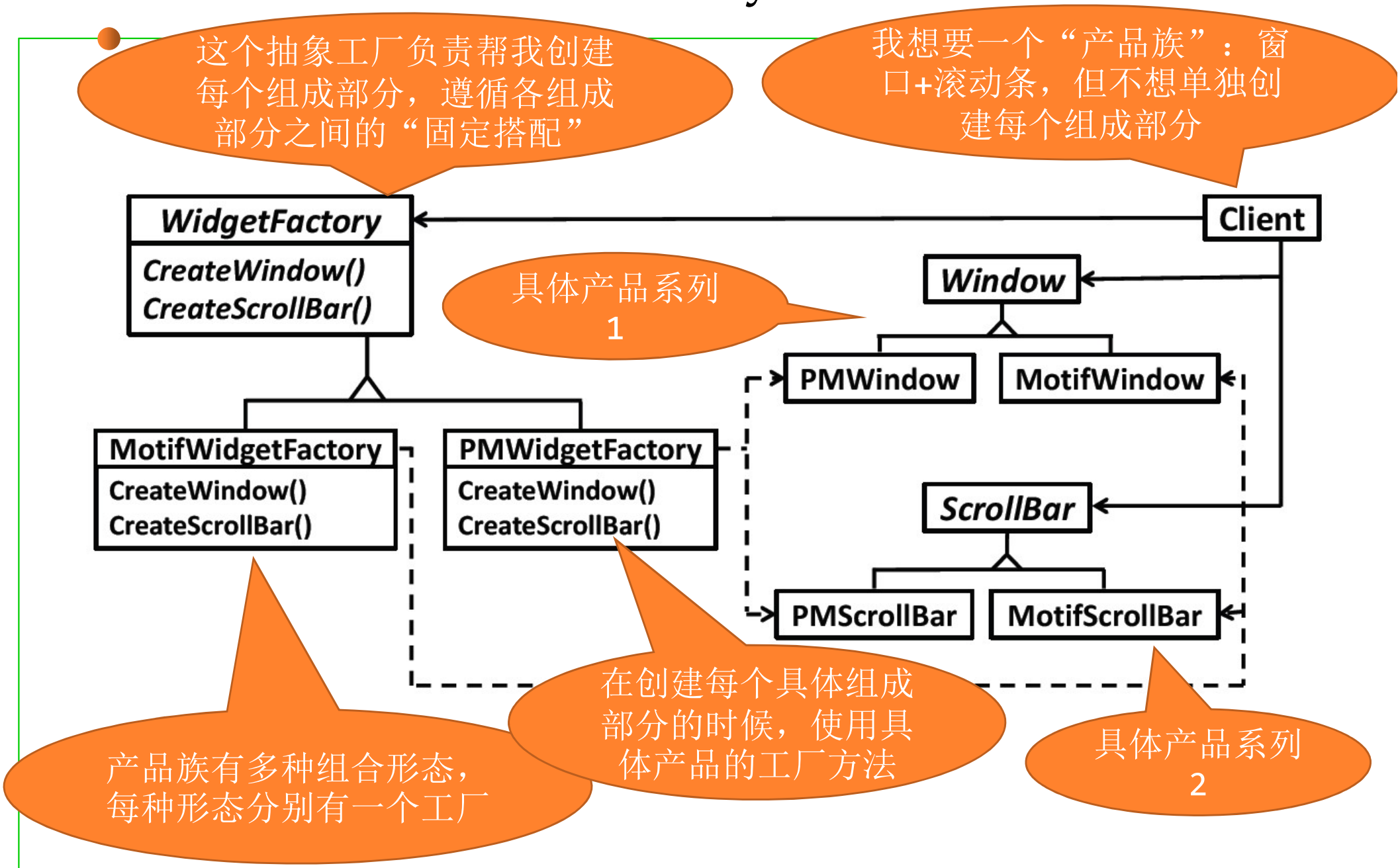
- **Example 1: Consider a user interface toolkit that supports multiple looks and feel standards for different operating systems: 一个UI，包含多个窗口控件，这些控件在不同的OS中实现不同**
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- **Example 2: Consider a facility management system for an intelligent house that supports different control systems: 一个仓库类，要控制多个设备，这些设备的制造商各有不同，控制接口有差异**
  - How can you write a single control system that is independent from the manufacturer?

- **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes. 抽象工厂模式：提供接口以创建一组相关/相互依赖的对象，但不需要指明其具体类。

# Abstract Factory pattern

- **Name**: **Abstract Factory (or Kit)**
- **Intent**: allow creation of families of related objects independent of implementation
- **Approach:** Using a factory to return factories that can be used to create sets of related objects.
- **Applicability**
  - Different families of components (products) that keep independence from Initialization or Representation
  - Must be used in mutually exclusive and consistent way
  - Hide existence of multiple families from clients
  - Manufacturer Independence
  - Cope with upcoming change

# Structure of Abstract Factory



# Example

抽象产品接口和具体产品类

```
//AbstractProduct
public interface Window{
  public void setTitle(String s);
  public void repaint();
  public void addScrollbar(…);
}
//ConcreteProductA1
public class PMWindow
             implements Window{
  public void setTitle(){…}
  public void repaint(){…}
}


//ConcreteProductA2
public class MotifWindow
             implements Window{
  public void setTitle(){…}
  public void repaint(){…}
}
```

抽象工厂接口和具体工厂类

```
//AbstractFactory
public interface AbstractWidgetFactory{
  public Window createWindow();
  public Scrollbar createScrollbar();
}
```

第一个具体产品的工厂方法

```
//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

```
//ConcreteFactory2
public class WidgetFactory2{
  public Window createWindow(){
    return new MotifWindow();
  }
  public Scrollbar createScrollbar(){B}
}
```

# Example

辅助类

Delegate到抽象工厂类

```
public class GUIBuilder{
  public void buildWindow(AbstractWidgetFactory widgetFactory){
    Window window = widgetFactory.createWindow();
    Scrollbar scrollbar = widgetFactory.createScrollbar();
    window.setTitle("New Window");
    window.addScrollbar(scrollbar);
  }
}
```

使用抽象工厂类分别创建两个具体产品

把创建的两个具体产品实例组合起来（模式之外）

定义抽象工厂接口的实例

```
Client
GUIBuilder builder = new GUIBuilder();
AbstractWidgetFactory widgetFactory = null;

if(“Motif”)
   widgetFactory  = new WidgetFactory2();
else
   widgetFactory  = new WidgetFactory1();

builder.buildWindow(widgetFactory);
```

根据要创建的“组合产品”的类型，构建不同的抽象工厂子类

具体构建

# Notes

- **Abstract Factory** 创建的不是一个完整产品，而是“产品族”（遵循固定搭配规则的多类产品的实例），得到的结果是：多个不同产品的**object**，各产品创建过程对**client**可见，但“搭配”不能改变。
- 本质上，**Abstract Factory**是把多类产品的**factory method**组合在一起

```
AbstractWidgetFactory widgetFactory = null;

if(“Motif”)
   widgetFactory  = new WidgetFactory2();
else
   widgetFactory  = new WidgetFactory1();

Window window = widgetFactory.createWindow();
Scrollbar scrollbar = widgetFactory.createScrollbar();
```

这是最终得到的两个产品实例

如果不用Abstract Factory，直接用多个factory method，是否能实现目的？

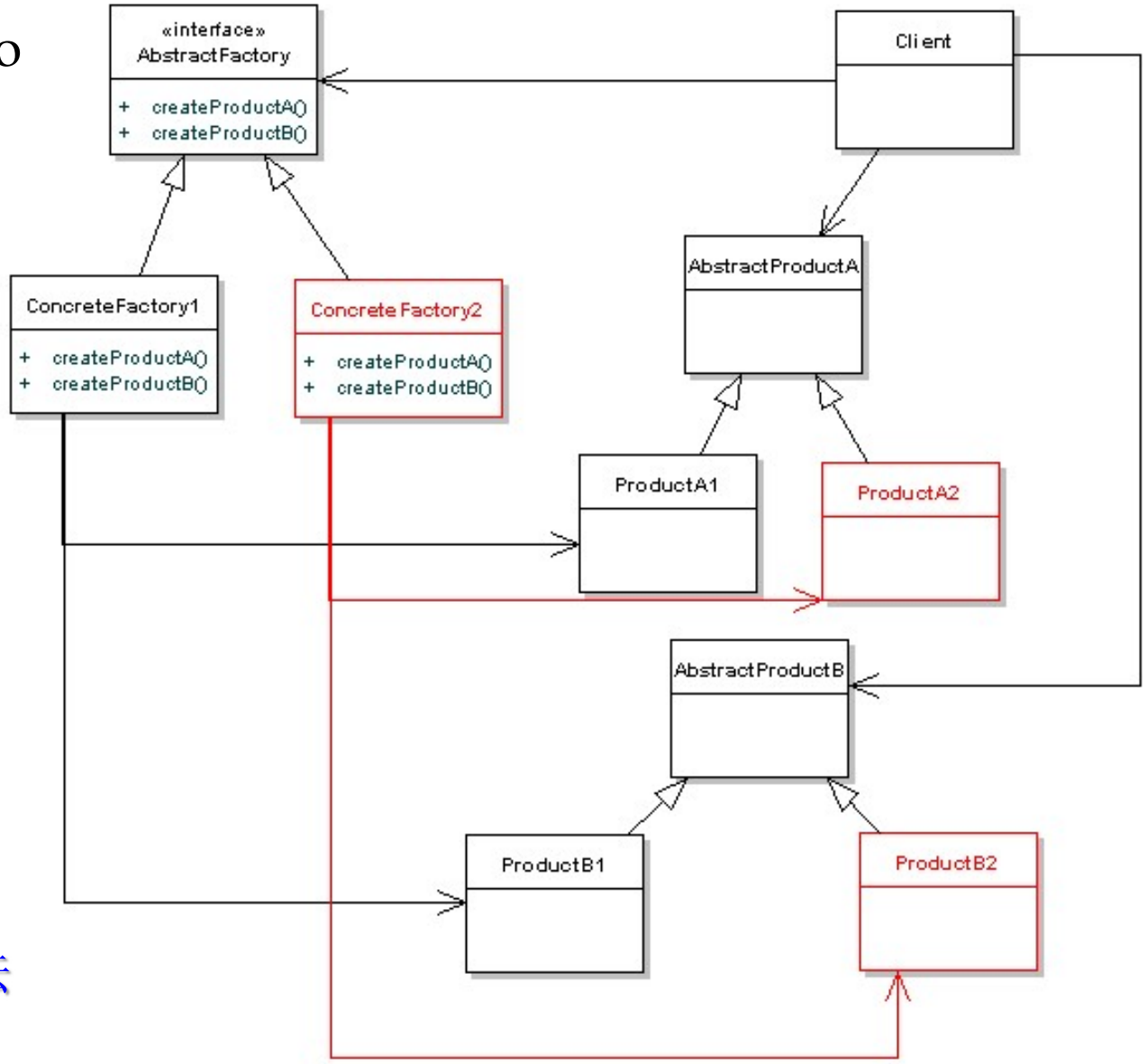
直接用factory method：client可能不知道搭配而用错工厂——牛仔裤+西装☹

# For example

- **A `StellarSystem` is composed of one `Stellar` and a list of `Planet`**
- **A `PersonalAppEcosystem` is composed of one `User` and a list of `MobileApp`**
- **In other words, the object creation of `L` and `PhysicalObject` is closely related but should not be independent. L和PhysicalObject的类型，要有固定搭配，不能随意组合**

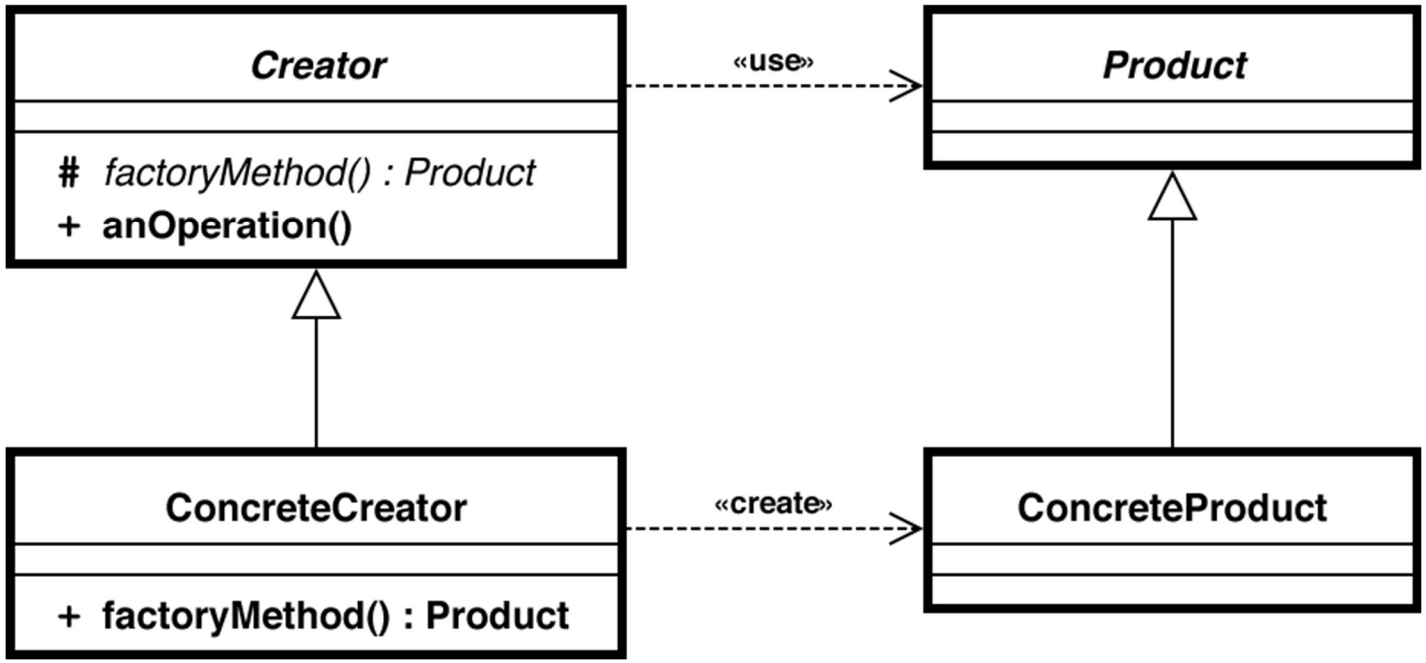
# Abstract Factory vs Factory Method

- **Factory Method** is used to create one product only but **Abstract Factory** is about creating families of related or dependent products. 创建一个对象 vs 创建多个类型的对象
- **Factory Method** pattern exposes a method to the client for creating the object whereas **Abstract Factory** exposes a family of related objects which consist of these Factory methods. 一个factory方法 vs 多个factory方法



# Abstract Factory vs Factory Method

- **Abstract Factory** pattern uses composition to delegate responsibility of creating object to another class while **Factory Method** pattern uses inheritance and relies on derived class or sub class to create object. 使用组合/委派 vs 使用继承/子类型






# 2 Structural patterns




# (1) Proxy

代理模式

# Proxy Pattern Motivation

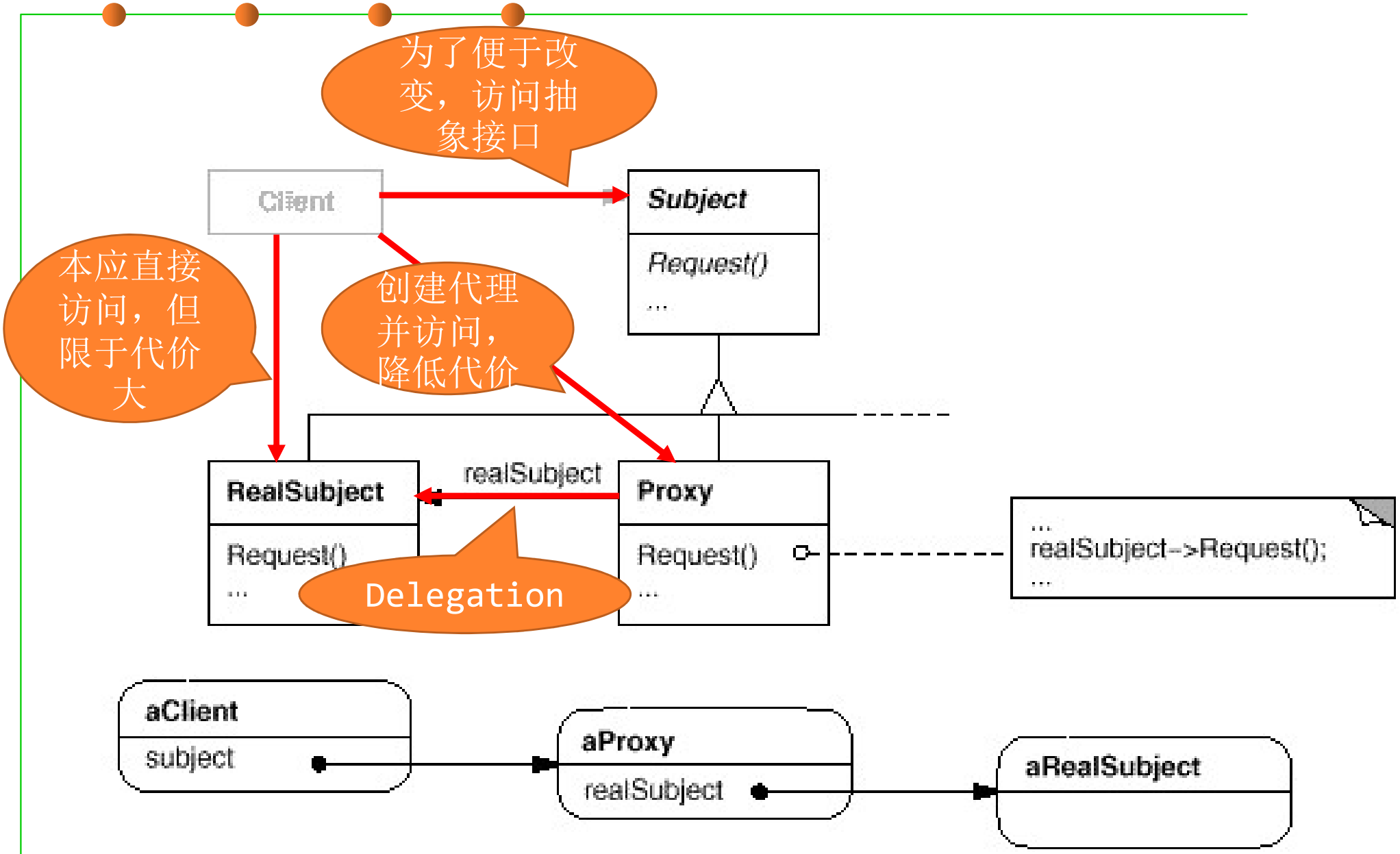
- **Goal:**
  - Prevent an object from being accessed directly by its clients
  - Allow for object level access control by acting as a pass through entity or a placeholder object.
- **Solution:**
  - Use an additional object, called a proxy
  - Clients access to protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

- **某个对象比较“敏感”/“私密”/“贵重”，不希望被client直接访问到，故设置proxy，在二者之间建立防火墙。**

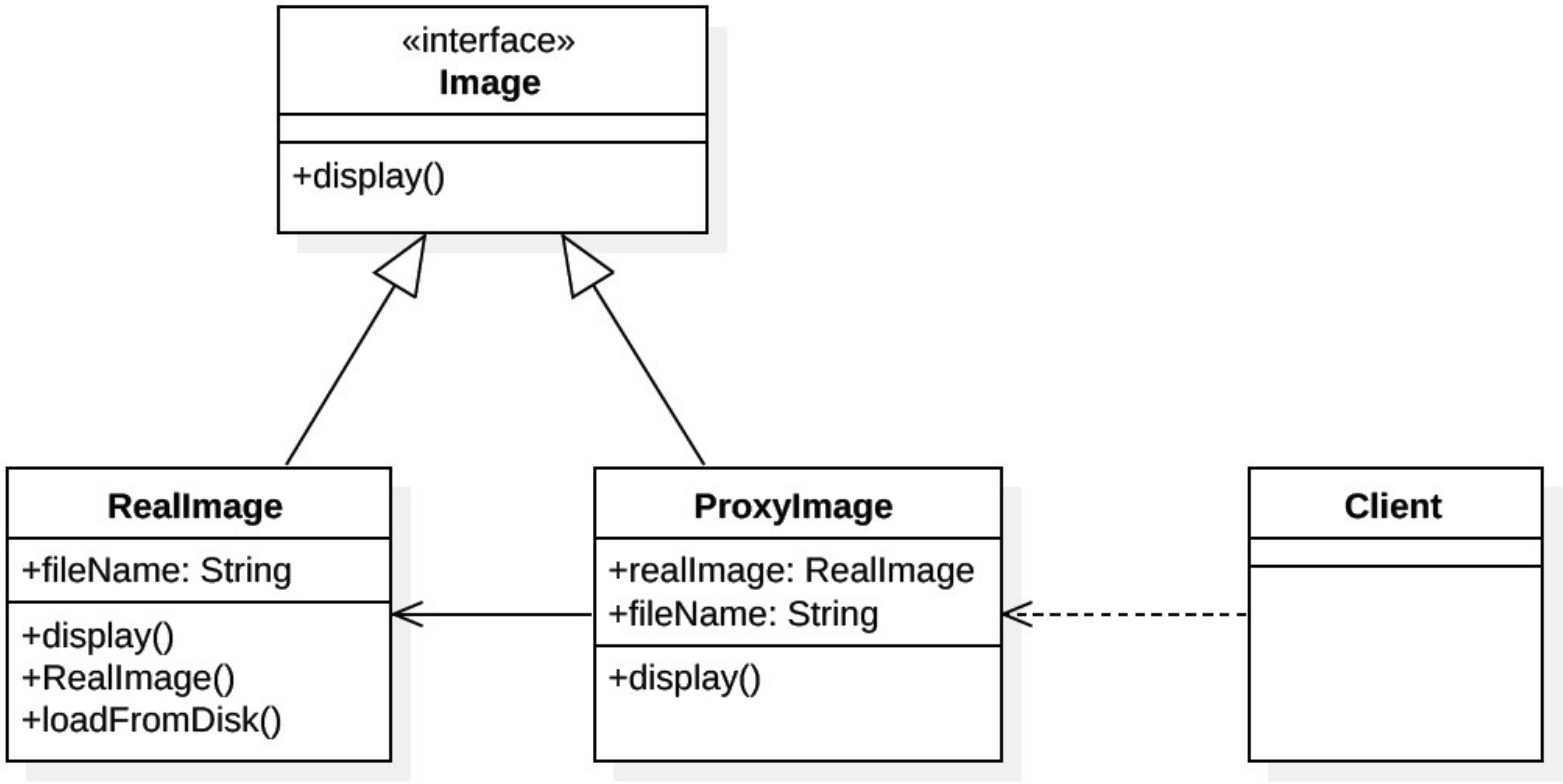
# The Proxy Pattern: 3 Types

- **Caching of information ("Remote Proxy")**
  - The Proxy object is a local representative for an object in a different address space
  - Good if information does not change too often
- **Standin ("Virtual Proxy")**
  - Object is too expensive to create or too expensive to download.
  - Good if the real object is not accessed too often
- **Access control ("Protection Proxy")**
  - The proxy object provides protection for the real object
  - Good when different actors should have different access and viewing rights for the same object
  - Example: Grade information accessed by administrators, teachers and students.

# Proxy Pattern Class Diagram



# Example


«interface»
Image
+display()
RealImage
+fileName: String
+display()
+RealImage()
+loadFromDisk()
ProxyImage
+realImage: RealImage
+fileName: String
+display()
Client

# 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

```
public class ProxyImage implements Image {
   private Image realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

但不需要在构造的时候从文件装载

如果display的时候发现没有装载，则再delegation

Delegate到原来的类来完成具体装载

```
Client:
Image image = new ProxyImage(“pic.jpg");
image.display();
image.display();
```

# Proxy vs. Adaptor

- **Adapter: structural pattern, and the purpose is to change the interface of class/library A to the expectations of client B. 目的：消除不兼容，目的是B以客户端期望的统一的方式与A建立起联系。**
  - The typical implementation is a wrapper class or set of classes.
  - The purpose is not to facilitate future interface changes, but current interface incompatibilities.
- **Proxy: behavioral pattern, also uses wrapper classes, but the purpose is to create a stand-in for a real resource. 目的：隔离对复杂对象的访问，降低难度/代价，定位在“访问/使用行为”**
  - The real resource resides on a remote computer (the proxy facilitates the interaction with the remote resource)
  - The real resource is expensive to create (the proxy ensures the cost is not incurred unless/until really needed)
  - A proxy provides a drop-in replacement for the real resource it is a stand-in for, so it must provide the same interface.




# 3 Behavioral patterns



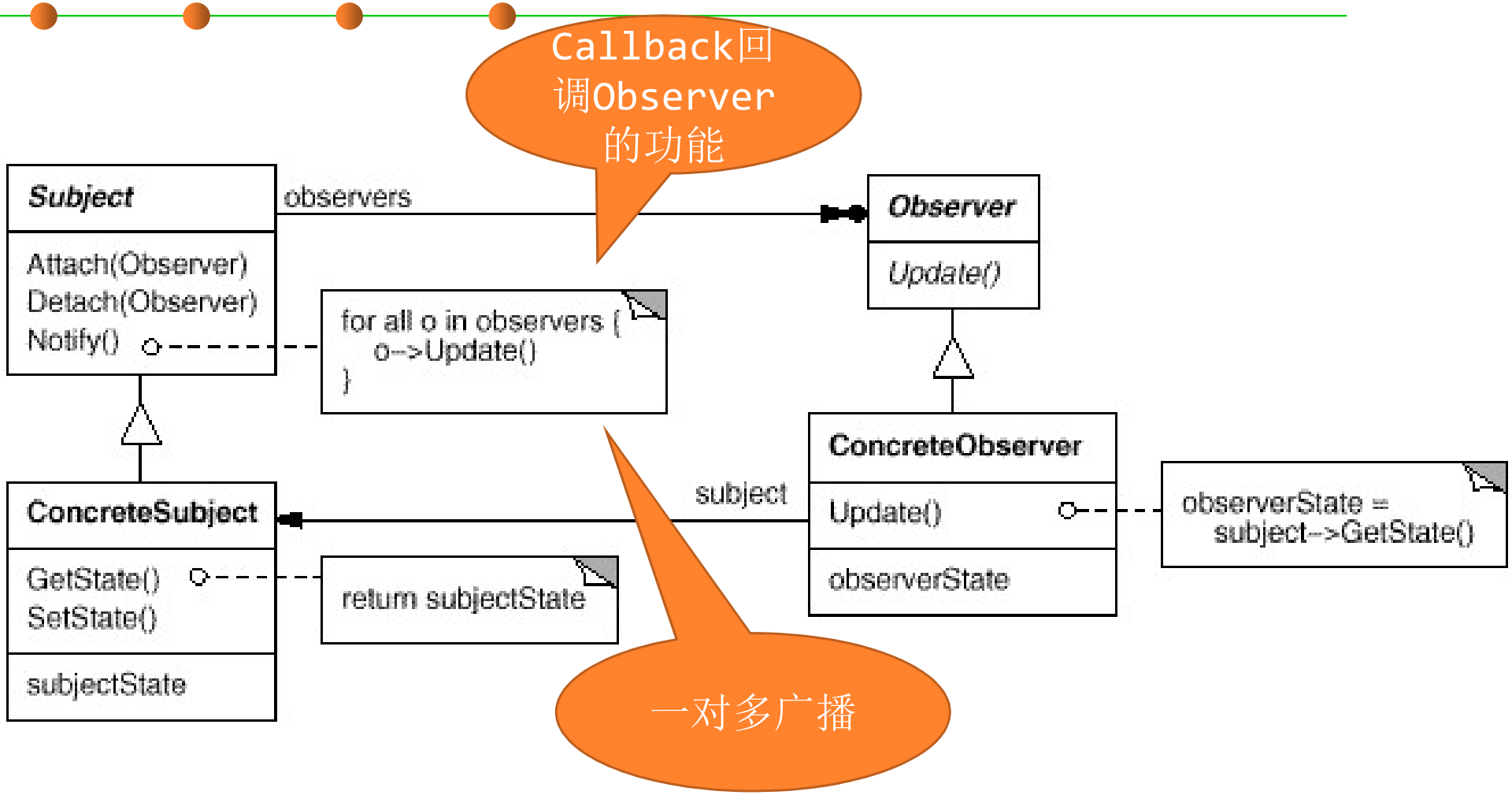

# (1) Observer

# Observer pattern

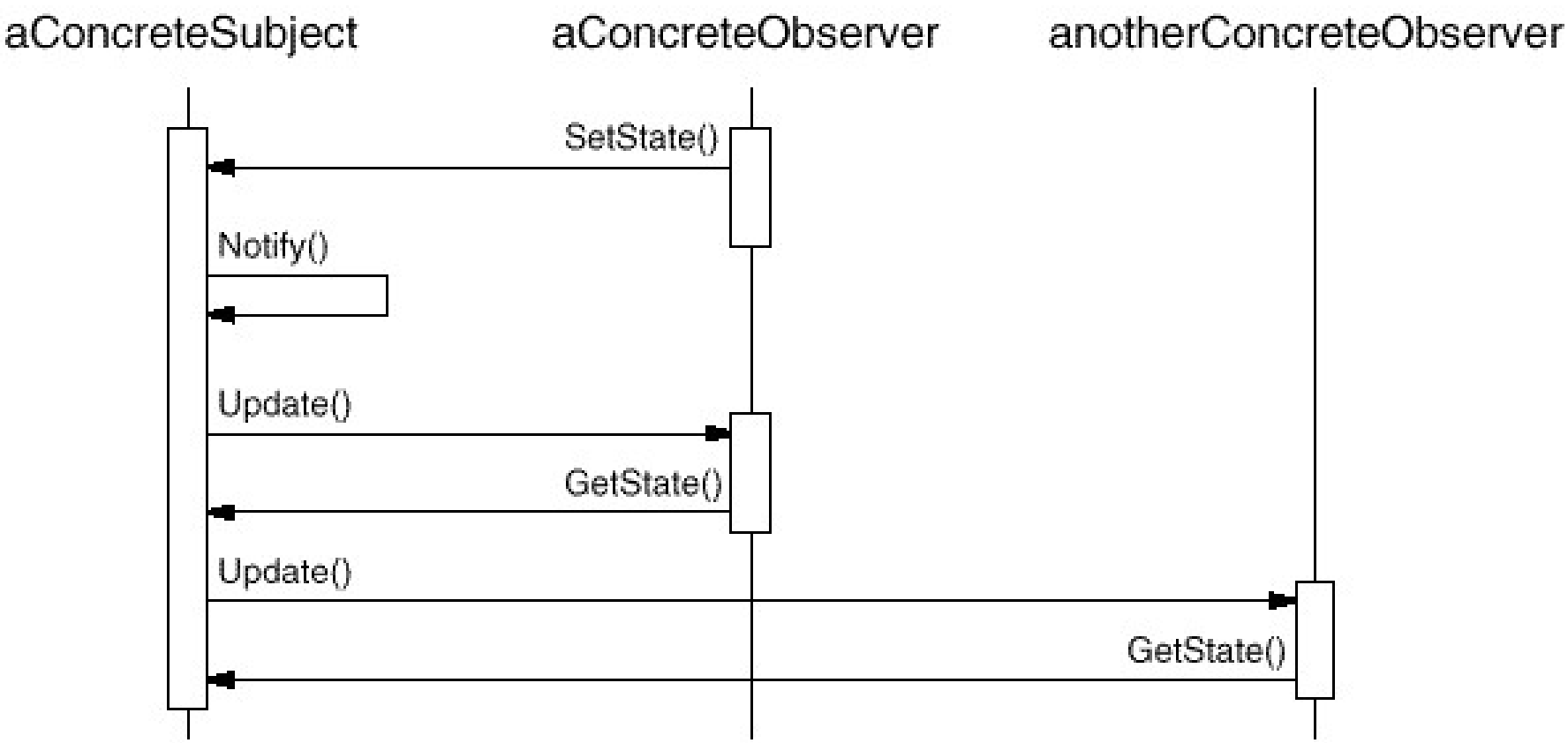
- **Problem: Dependent's state must be consistent with master's state**
- **Solution: Define four kinds of objects:**
  - Abstract subject: maintain list of dependents; notifies them when master changes
  - Abstract observer: define protocol for updating dependents
  - Concrete subject: manage data for dependents; notifies them when master changes
  - Concrete observers: get new subject state upon receiving update message

- **“粉丝”对“偶像”感兴趣，希望随时得知偶像的一举一动**
- **粉丝到偶像那里注册，偶像一旦有新闻发生，就推送给已注册的粉丝（回调callback粉丝的特定功能）**

# Observer pattern



# Use of Observer pattern



# Example

```
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();
   }

   public void attach(Observer observer){observers.add(observer);}

   private void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

维持一组“对自己感兴趣的”对象

在自己状态变化时，通知所有“粉丝”

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

```
public abstract class Observer {
    protected Subject subject;
    public abstract void update();
}
```

“粉丝”的抽象接口

构造时，指定自己的“偶像”subject，把自己注册给它
这是相反方向的 delegation

```
public class BinaryObserver extends Observer{

    public BinaryObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Binary String: " +
                Integer.toBinaryString(
                        subject.getState() ) );
    }
}
```

注意：这个方法是被“偶像”回调的

当“偶像”有状态变化时，调用 subject.getState() 获取最新信息

不同子类的“粉丝”，其行为可定制

# Example

```
public class ObserverPatternDemo {

   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

偶像一枚

粉丝三枚

偶像有新闻了

并没有直接调用粉丝行为的代码！但其内部隐藏着对粉丝行为的delegation

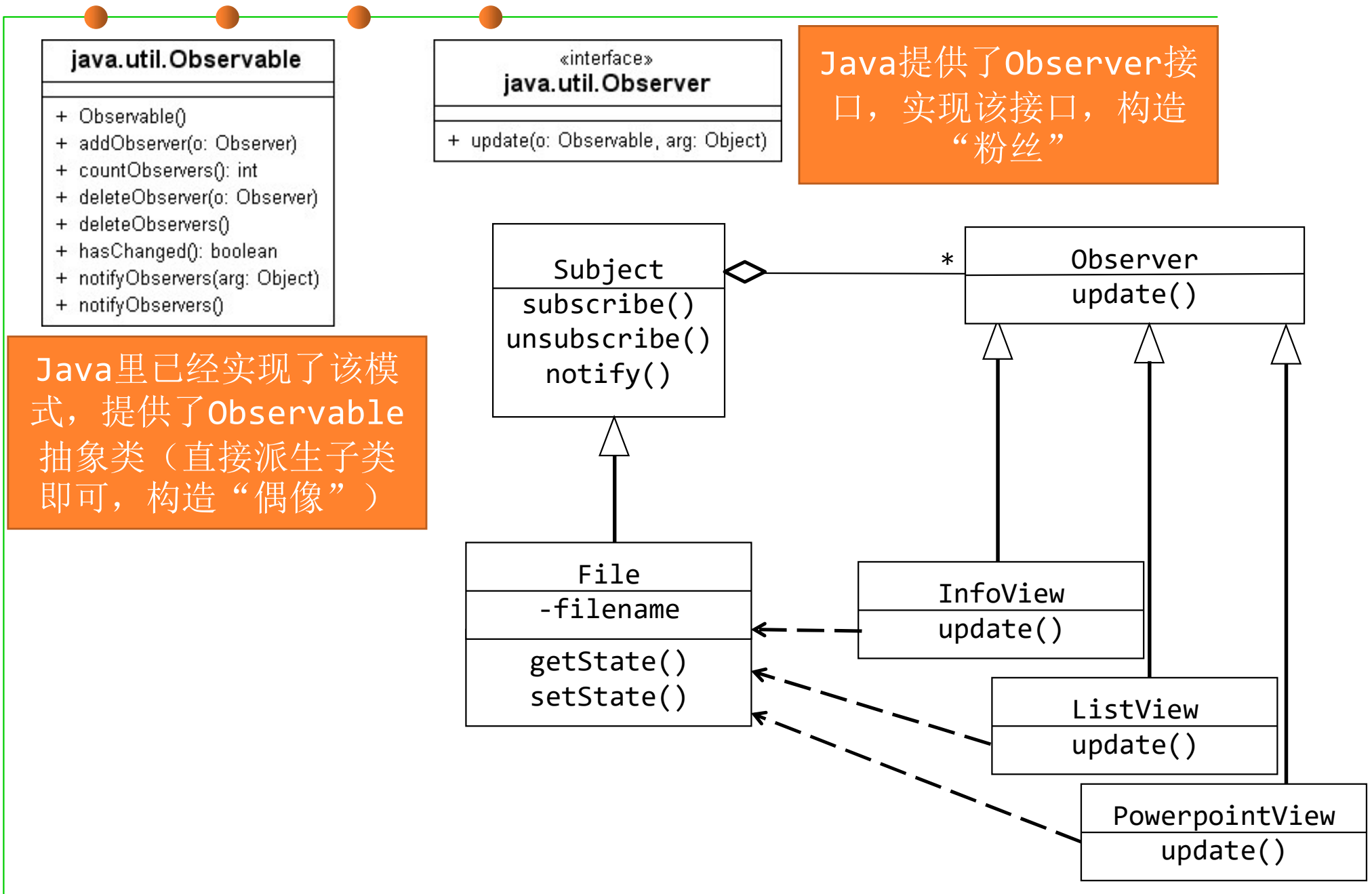
# Observer Pattern

- **Models a 1-to-many dependency between objects**
  - Connects the state of an observed object, the subject with many observing objects, the observers
- **Usage:**
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- **Three variants for maintaining the consistency:**
  - Push Notification: Every time the state of the subject changes, all the observers are notified of the change
  - Push-Update Notification: The subject also sends the state that has been changed to the observers
  - Pull Notification: An observer inquires about the state the of the subject
- **Also called Publish-ubscribe.**

# Observer Pattern

- **Advantage:**
  - Low coupling between subject and observers: `Subject` unaware of dependents
  - Support for broadcasting: Dynamic addition and removal of `observers`
  - Unexpected updates: No control by the subject on computations by `observers`
- **Implementation issues**
  - Storing list of observers: typically in `subject`
  - Observing multiple subjects: typically add parameters to `update()`
  - Who triggers update: state-setting operations of subject

# Example: to Maintain Consistency across Views



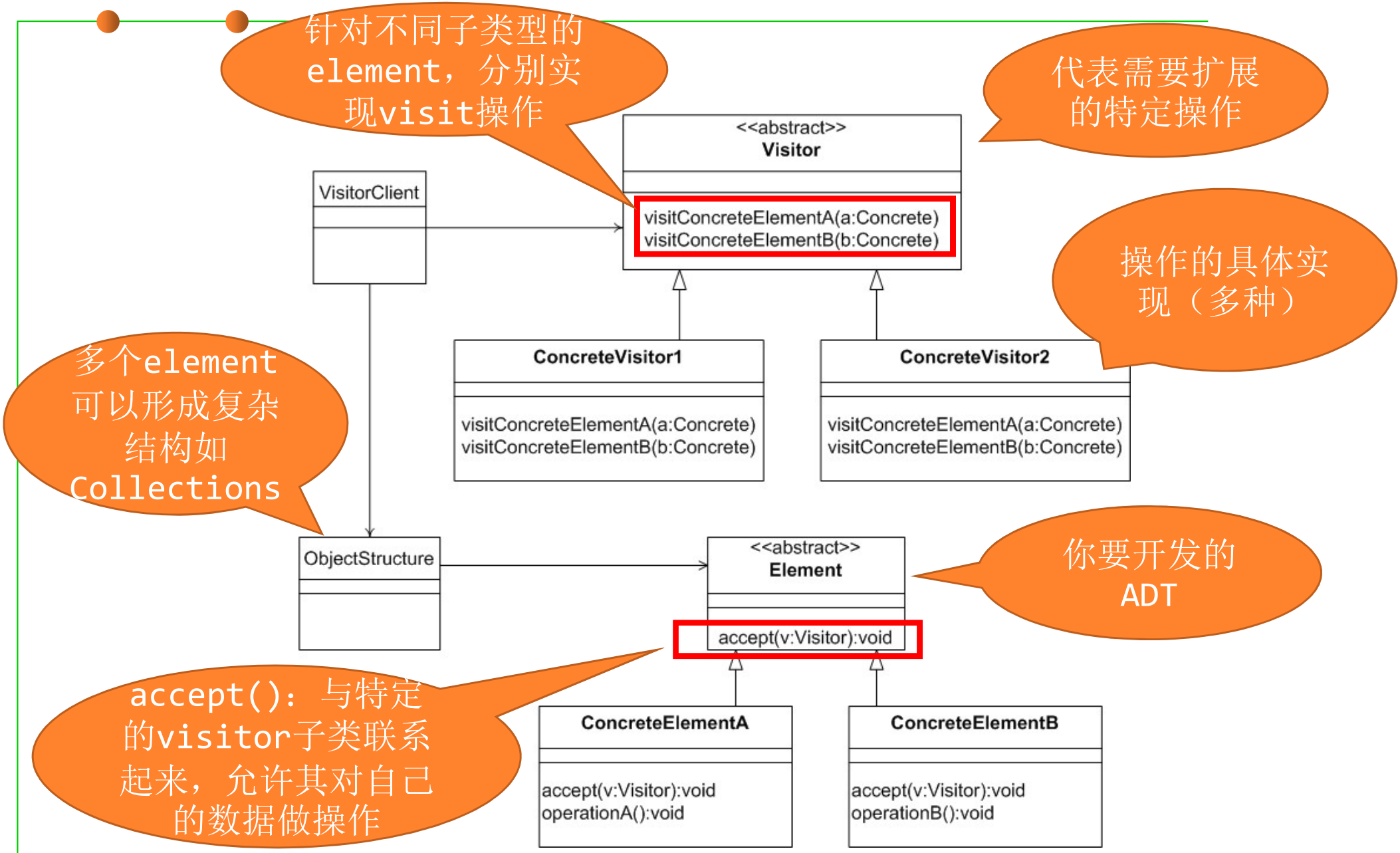



# (2) Visitor

# 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



# 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);
  }
}
```

# 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

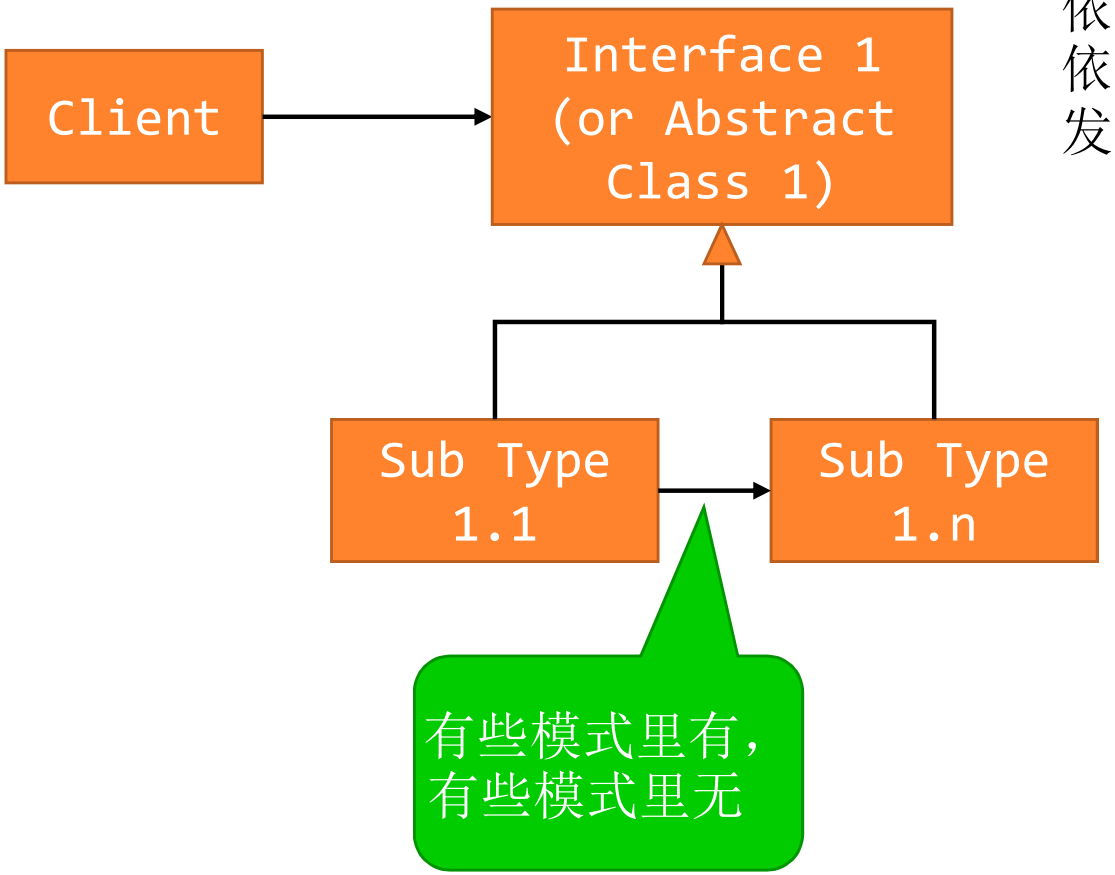
# Strategy vs visitor

- **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



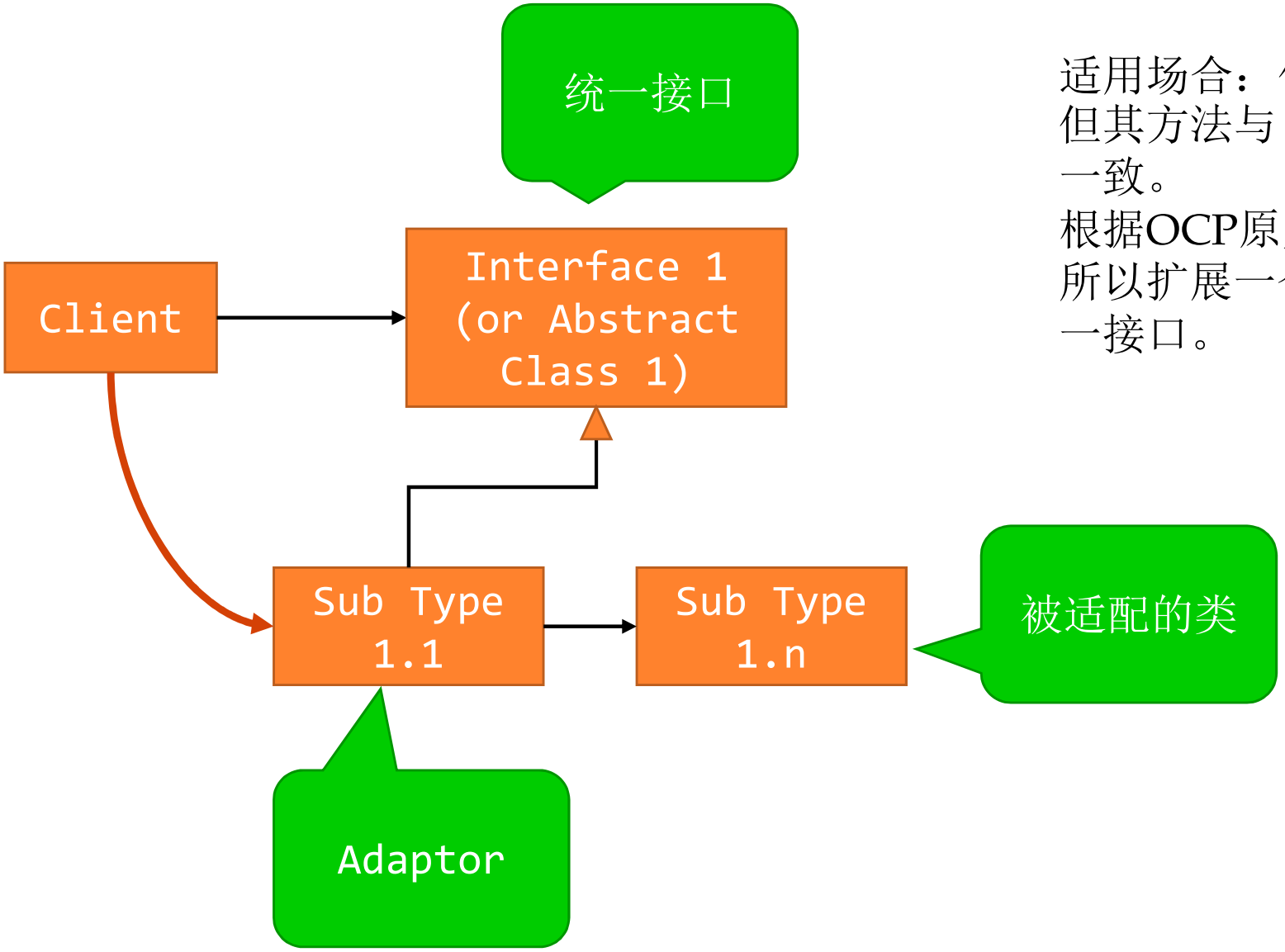

只使用“继承”，不使用“delegation”

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”

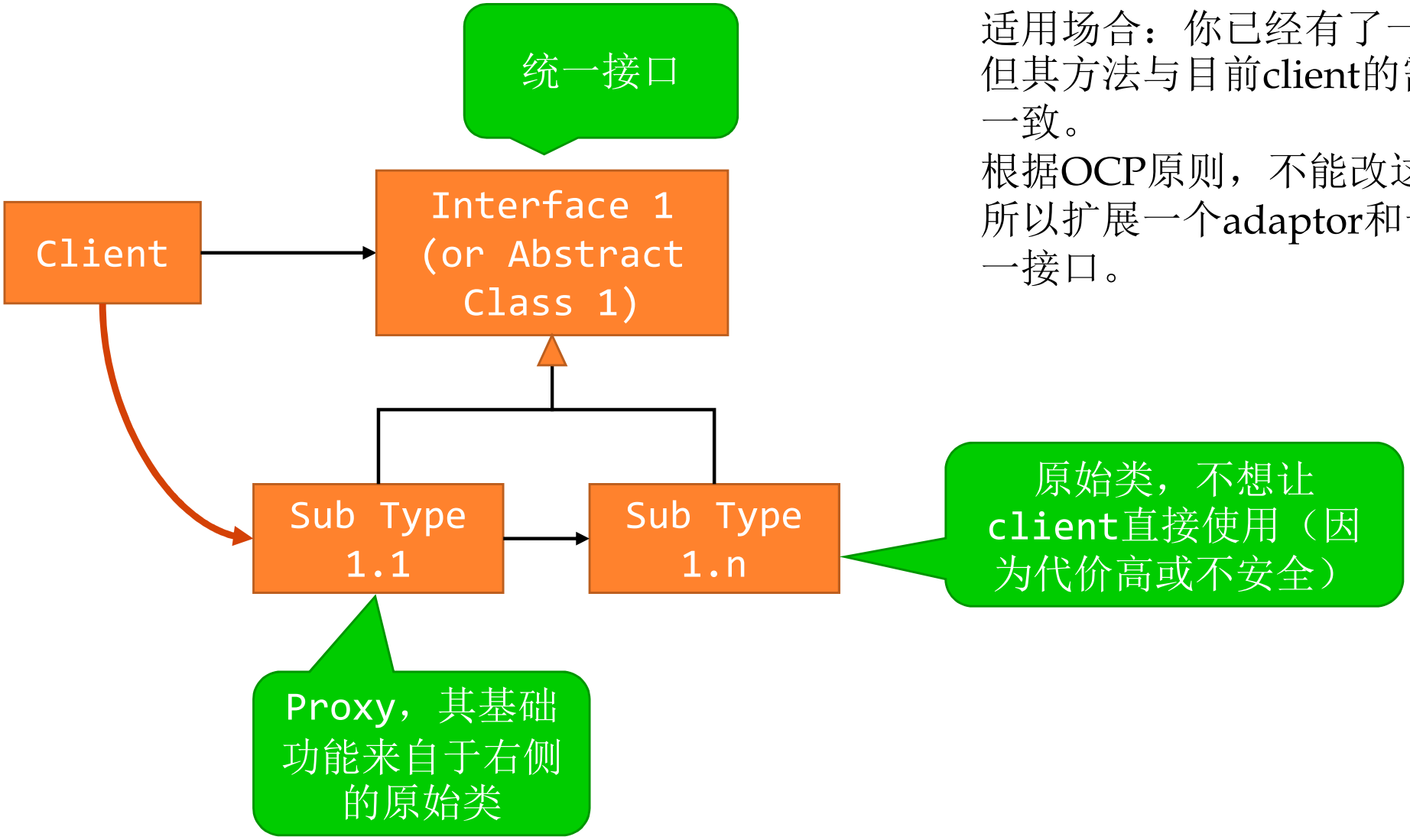
发生变化时最好是“扩展”而不是“修改”

# 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。

# Proxy

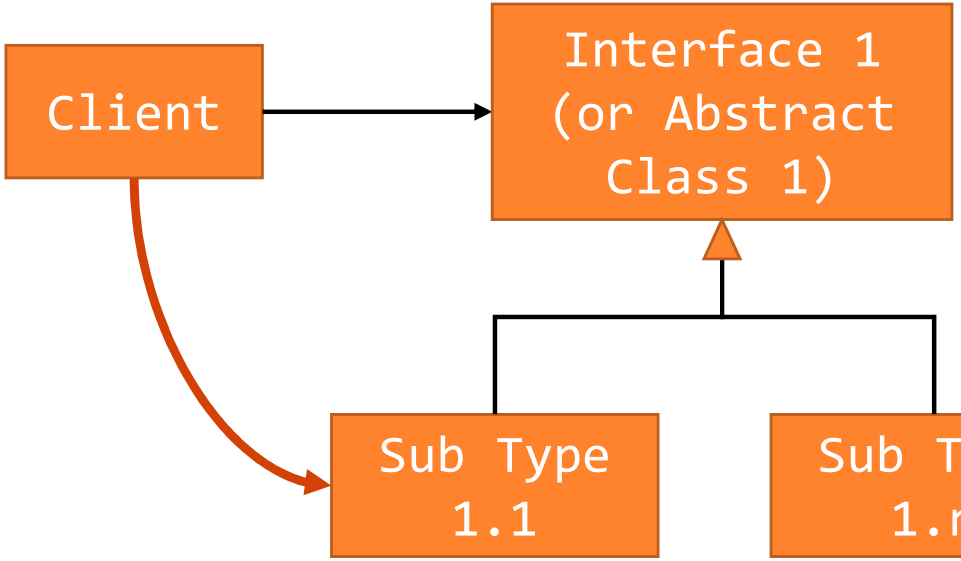



适用场合：你已经有了一个类，但其方法与目前client的需求不一致。
根据OCP原则，不能改这个类，所以扩展一个adaptor和一个统一接口。

# 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”

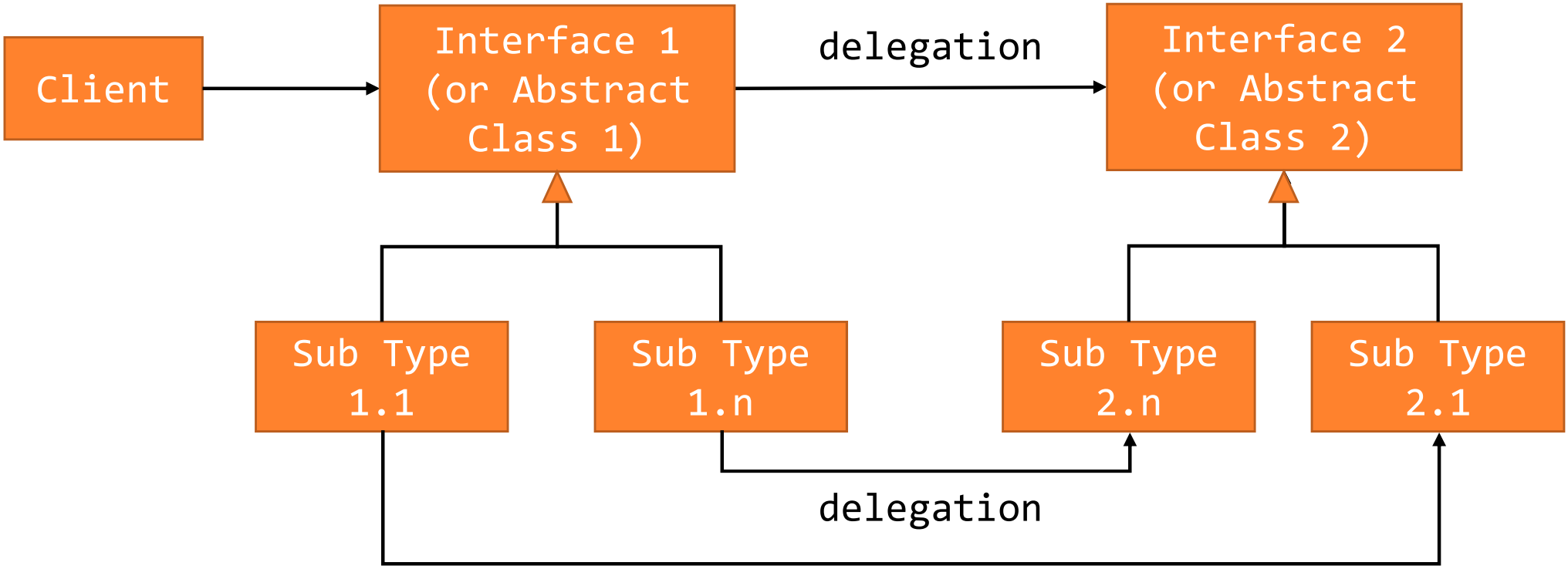



注意：如果某个步骤不需要有多种实现，直接在该抽象类里写出共性实现即可。

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

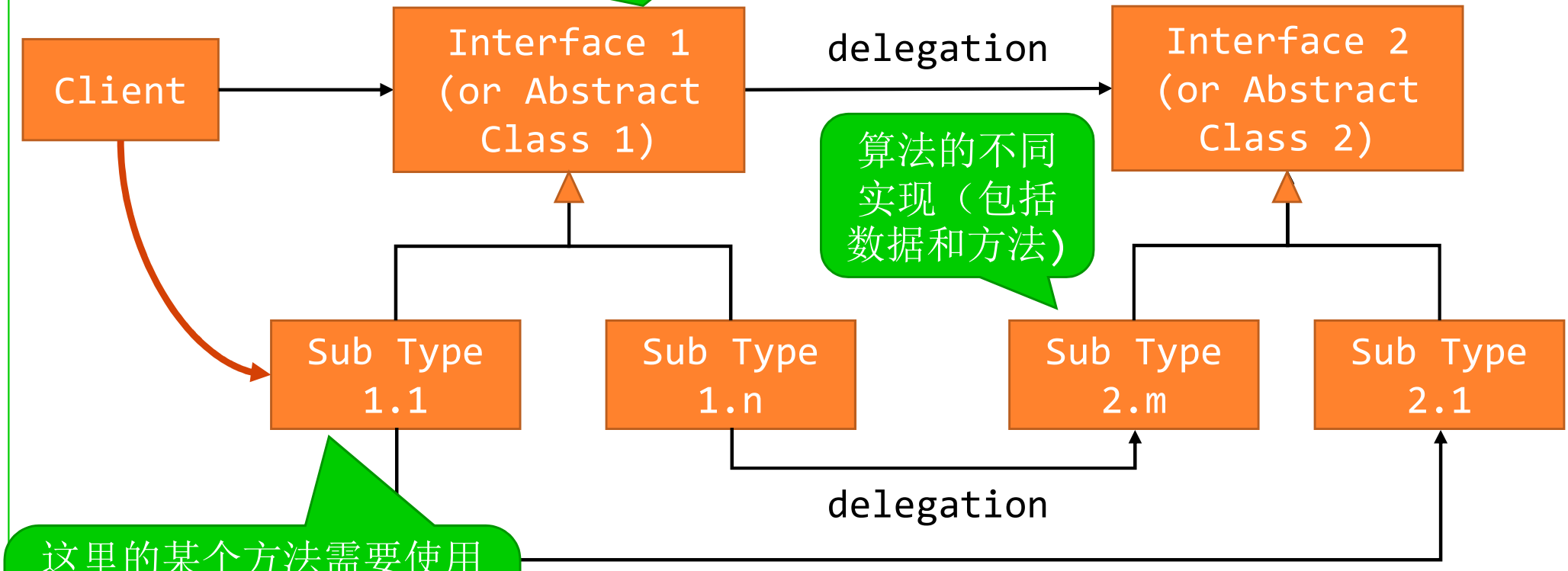
两棵“继承树”，两个层次的“delegation”



# 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

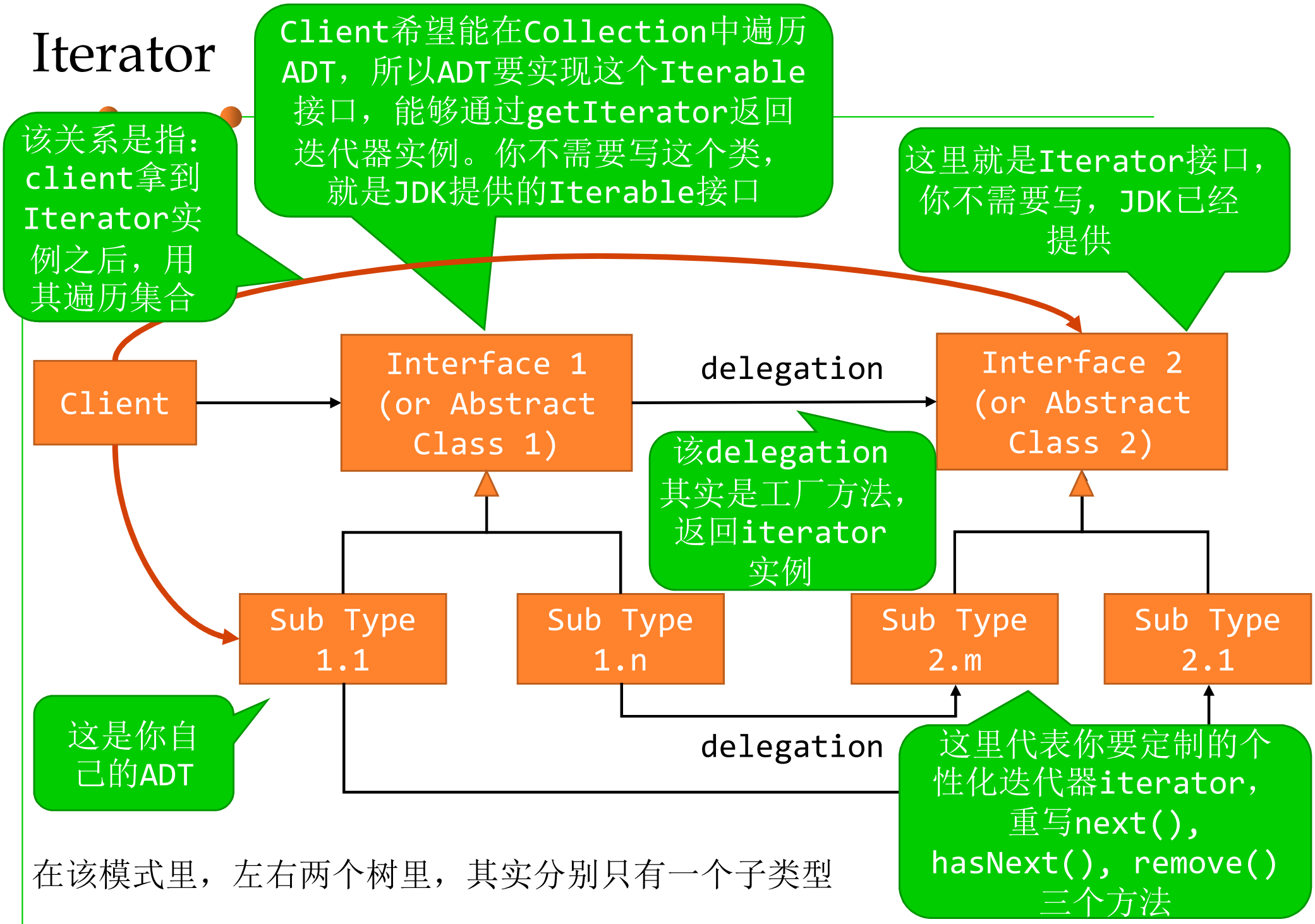
根据OCP原则，想有多个算法的实现，在右侧树里扩展子类型即可，在左侧子类型里传入不同的类型实例




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

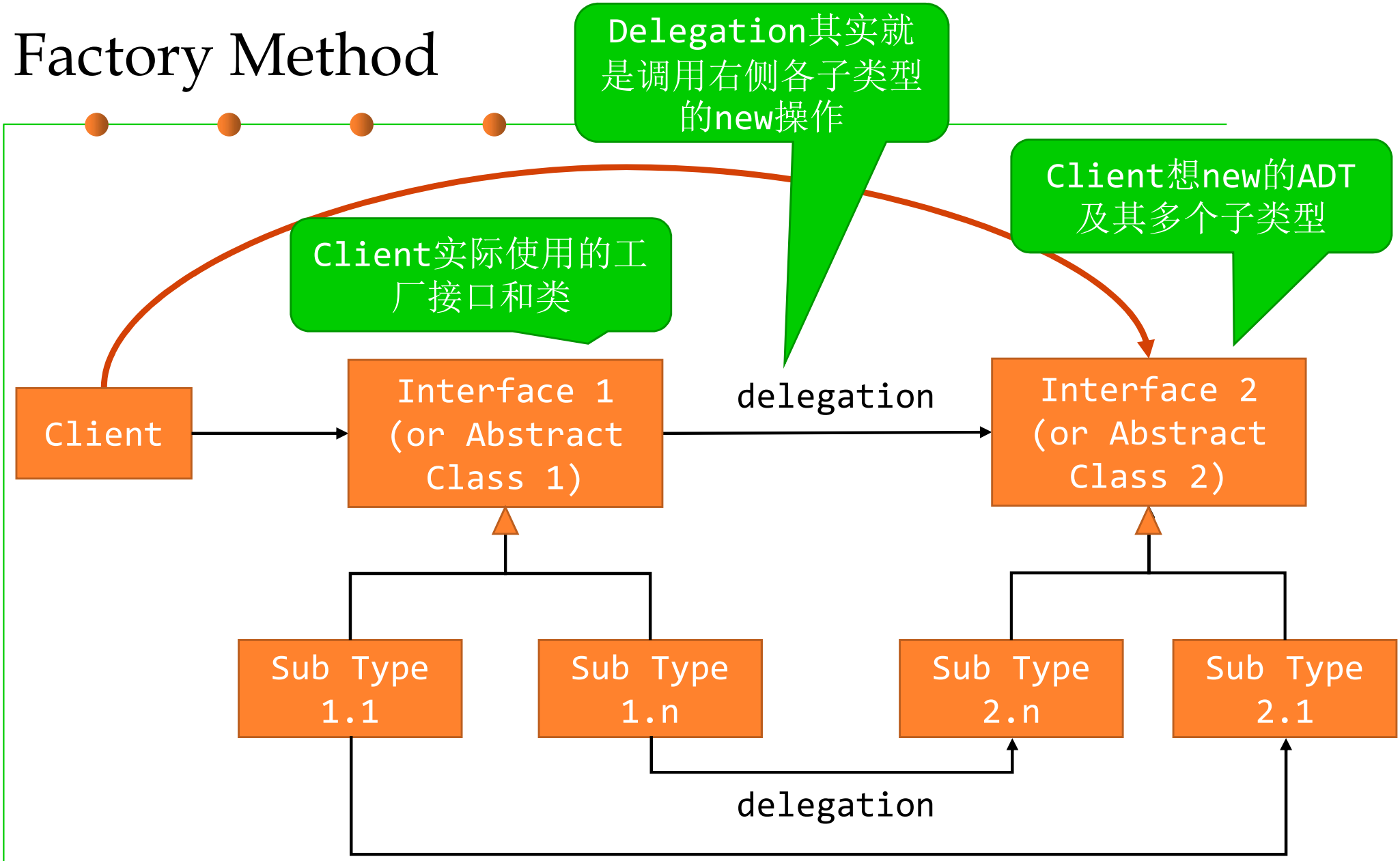
左右两侧的两棵树的子类型，不需要一一对应

# Iterator



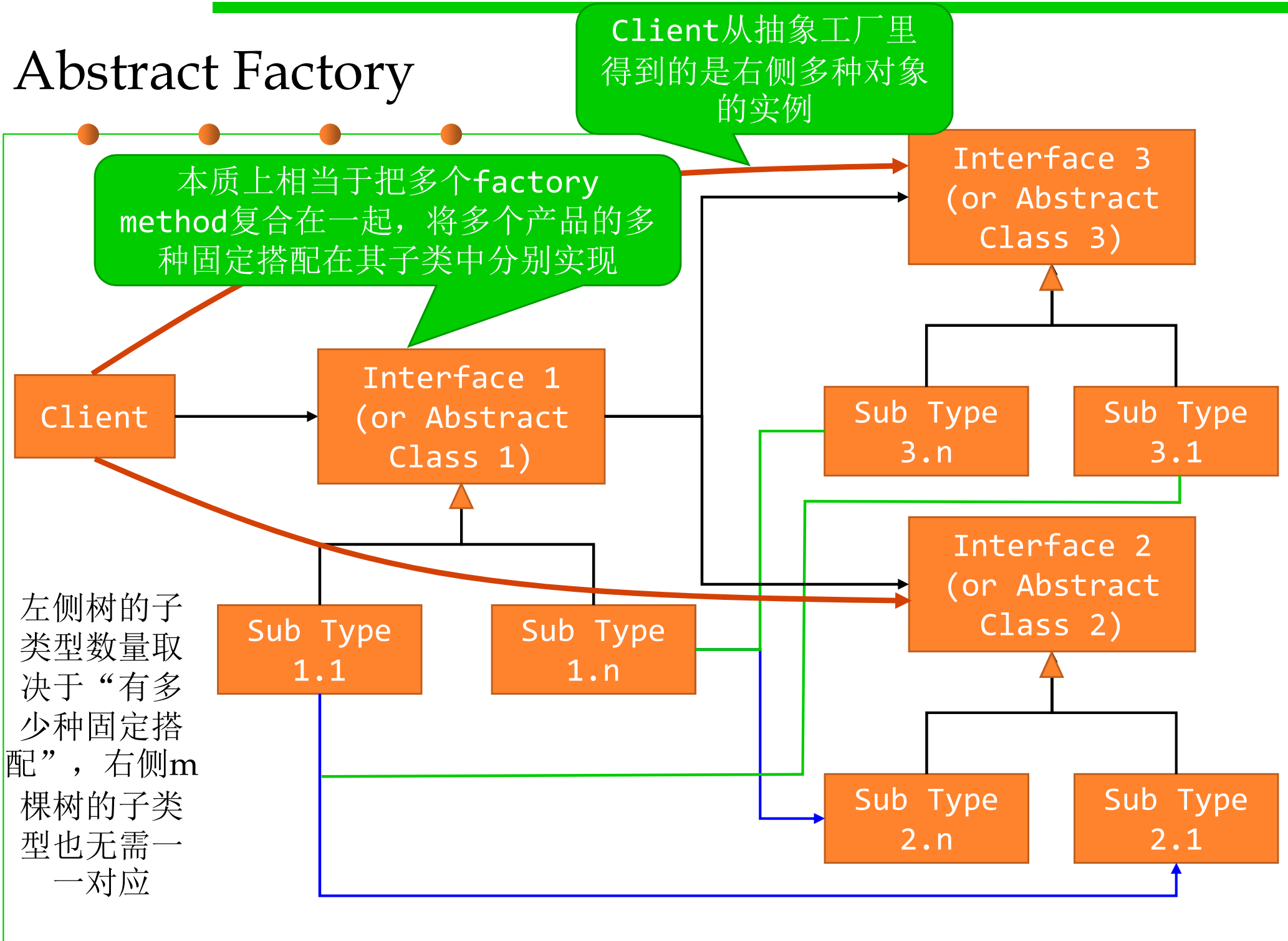

在该模式里，左右两个树里，其实分别只有一个子类型

# Factory Method

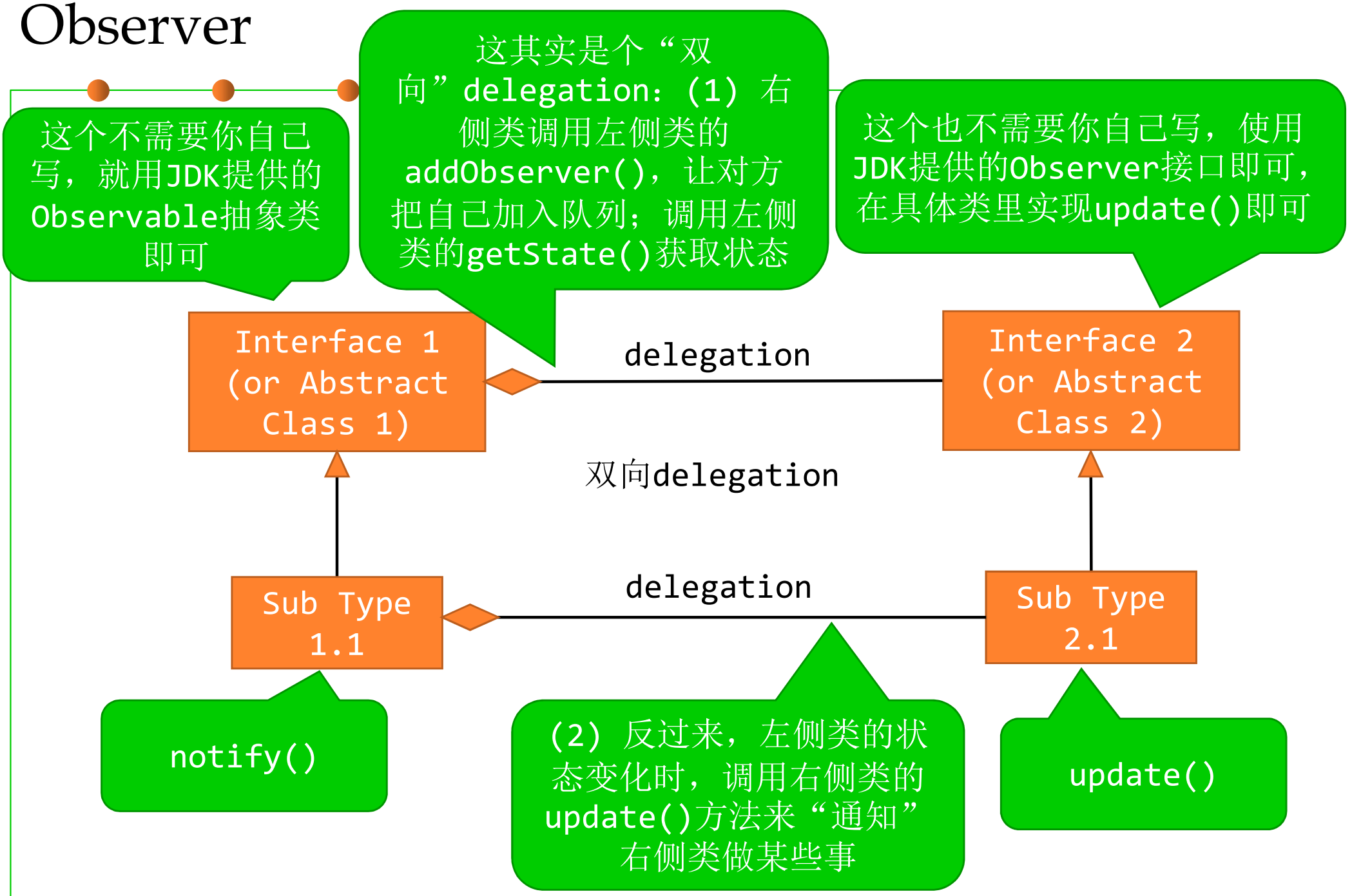



左右两棵树的子类型一一对应。如果在工厂方法里使用type表征右侧的子类型，那么左侧的子类型只要1个即可。

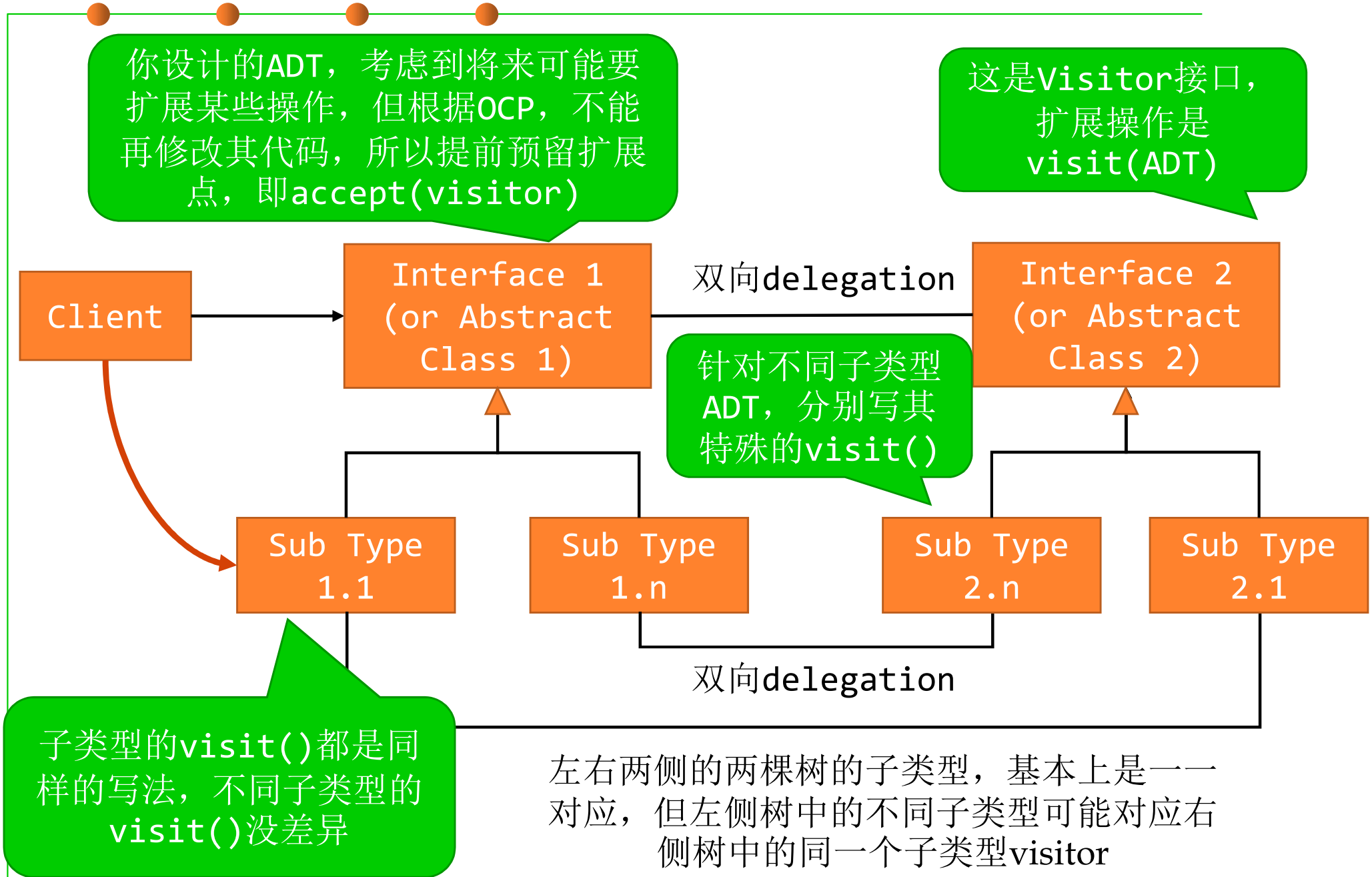
# Abstract Factory



# Observer


这个不需要你自己写，就用JDK提供的Observable抽象类即可
这其实是个“双向”delegation：(1) 右侧类调用左侧类的addObserver()，让对方把自己加入队列；调用左侧类的getState()获取状态
这个也不需要你自己写，使用JDK提供的Observer接口即可，在具体类里实现update()即可
Interface 1 (or Abstract Class 1)
delegation
Interface 2 (or Abstract Class 2)
双向delegation
Sub Type 1.1
delegation
Sub Type 2.1
notify()
(2) 反过来，左侧类的状态变化时，调用右侧类的update()方法来“通知”右侧类做某些事
update()

# 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, Abstract factory
- **Structural patterns**
  - Proxy
- **Behavioral patterns**
  - Observer, Visitor




# The end

April 20, 2020