哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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

April 21, 2024

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)

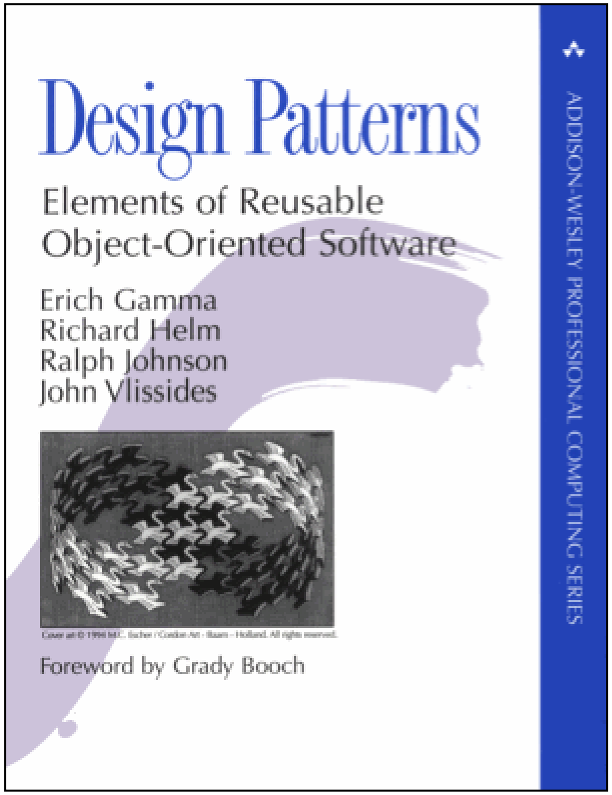
…allows the delivery of more functionality after an initial delivery (extensibility).

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
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.
  - 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.

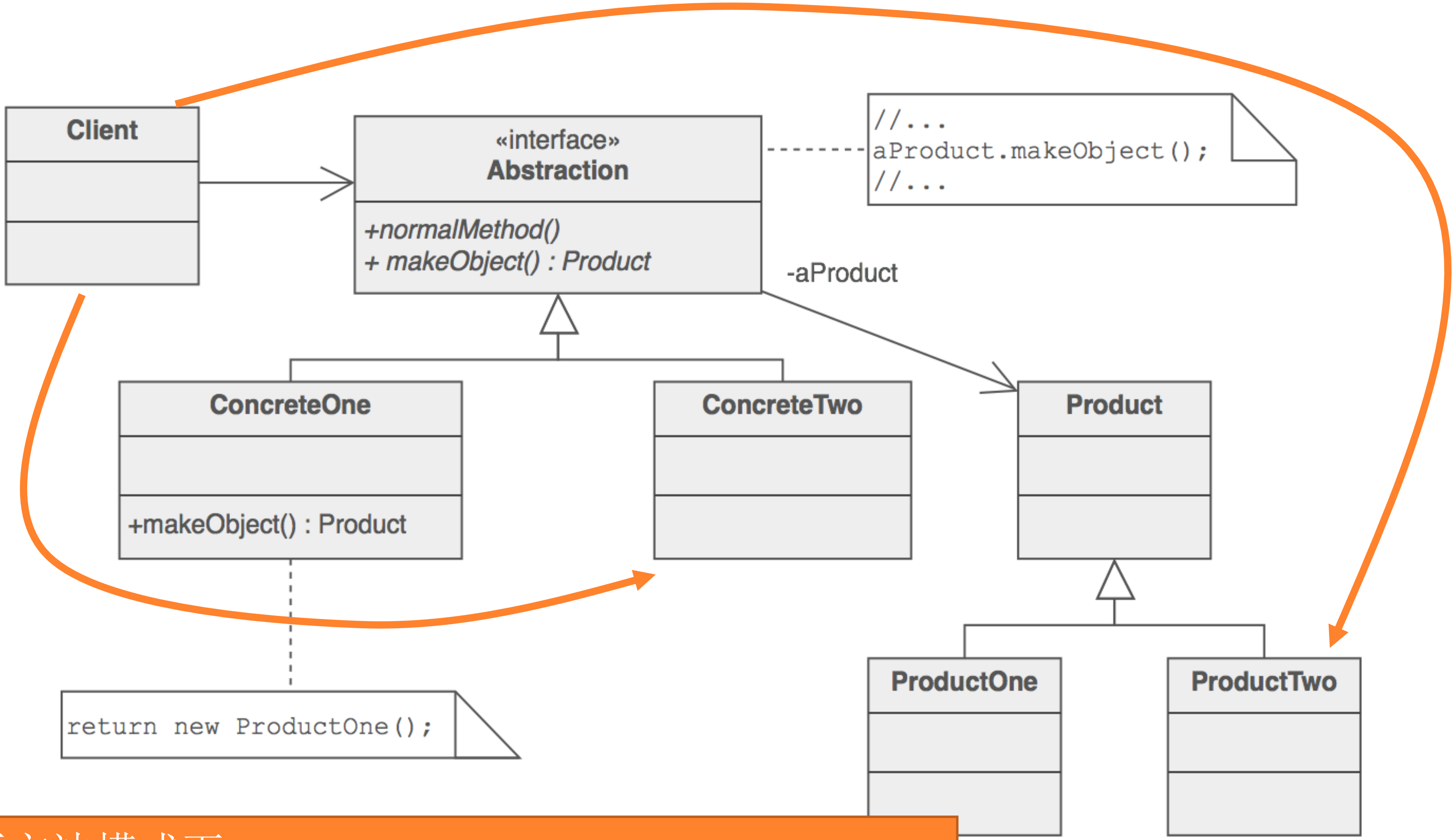
当client不知道要创建哪个具体类的实例，或者不想在client代码中指明要具体创建的实例时，用工厂方法。

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();



```
//...
aProduct.makeObject();
//...
```

```
return new ProductOne();
```

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

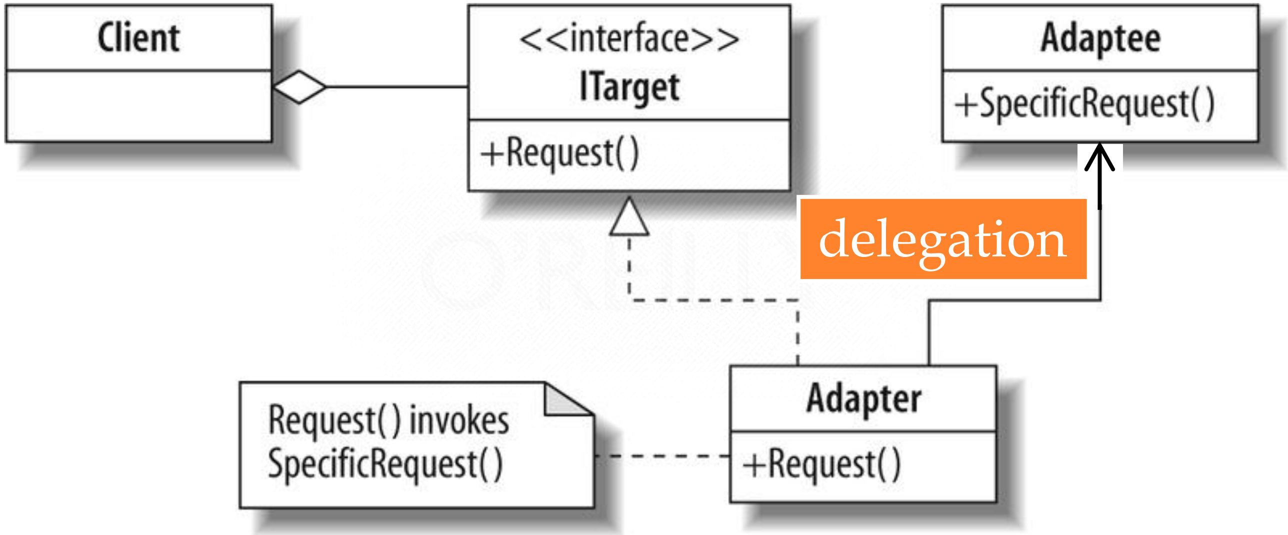
2 Structural patterns

(1) Adapter

适配器模式

Adapter Pattern

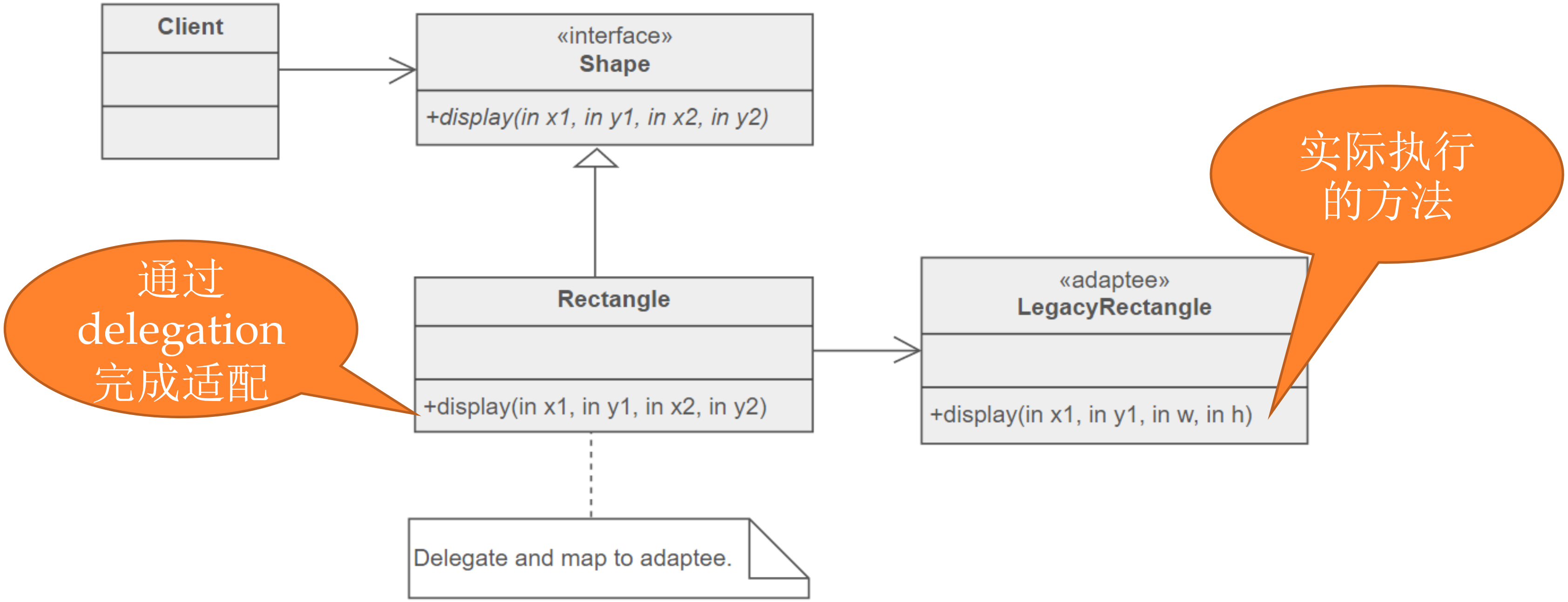
- **Intent: Convert the interface of a class into another interface that clients expect to get. 将某个类/接口转换为client期望的其他形式**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface. 通过增加一个接口，将已存在的子类封装起来，client面向接口编程，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.

实际执行的方法

通过delegation完成适配

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);
  }
}
```

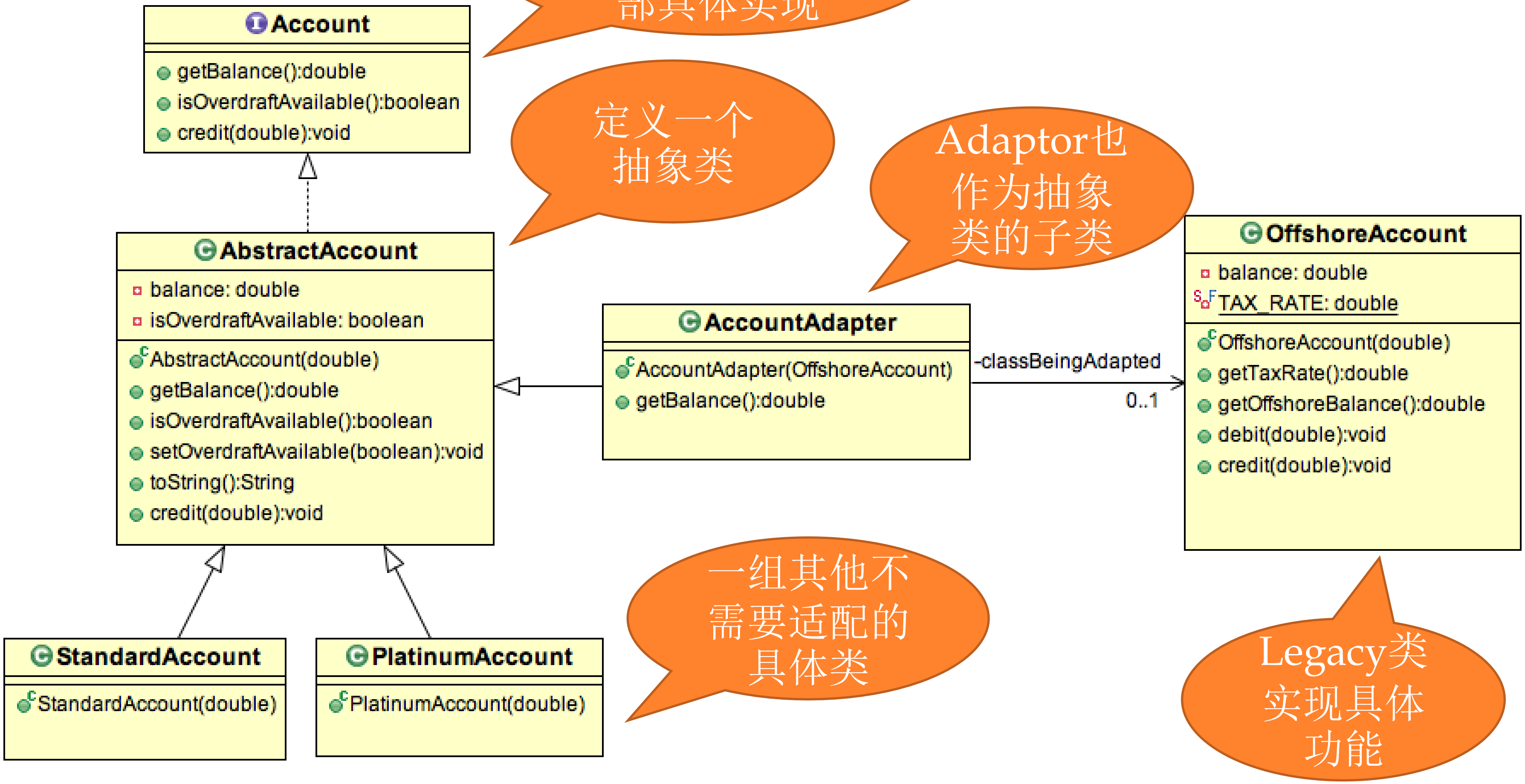
Adaptor类实现抽象接口

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`?



Account

getBalance():double
isOverdraftAvailable():boolean
credit(double):void

AbstractAccount

balance: double
isOverdraftAvailable: boolean
AbstractAccount(double)
getBalance():double
isOverdraftAvailable():boolean
setOverdraftAvailable(boolean):void
toString():String
credit(double):void

AccountAdapter

AccountAdapter(OffshoreAccount)
getBalance():double

-classBeingAdapted
0..1

OffshoreAccount

balance: double
TAX_RATE: double
OffshoreAccount(double)
getTaxRate():double
getOffshoreBalance():double
debit(double):void
credit(double):void

StandardAccount

StandardAccount(double)

PlatinumAccount

PlatinumAccount(double)

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

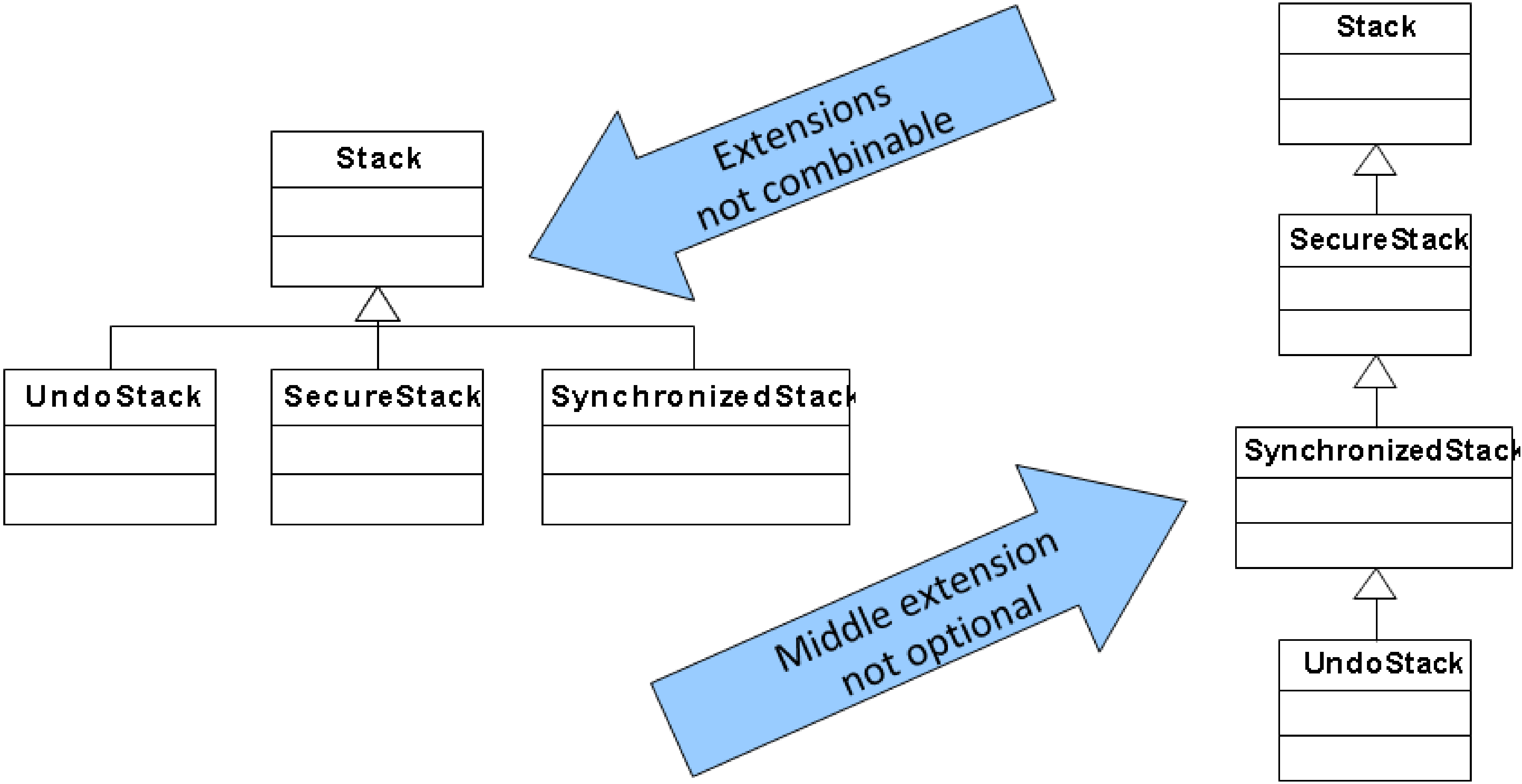
- **Suppose you want various extensions of a `Stack` data structure…**
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses
  - 用每个子类实现不同的特性

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`: …

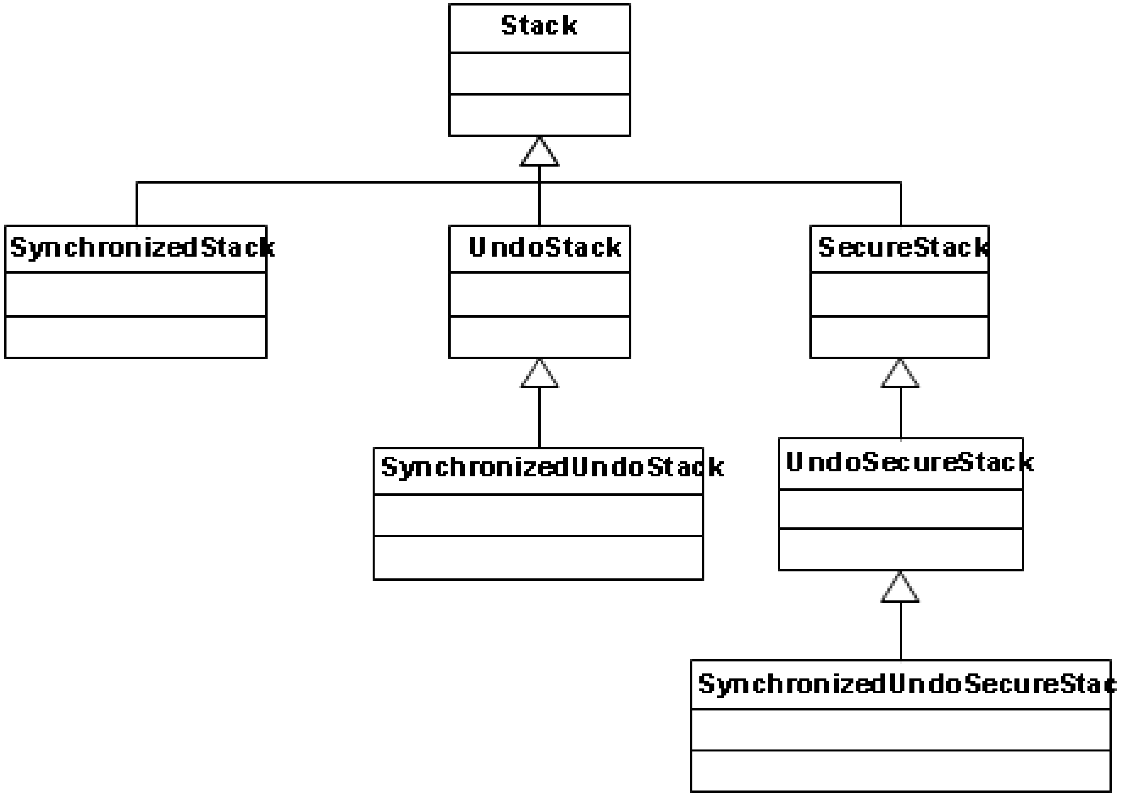
Inheritance
hierarchies?
Multi-Inheritance?

Limitations of inheritance



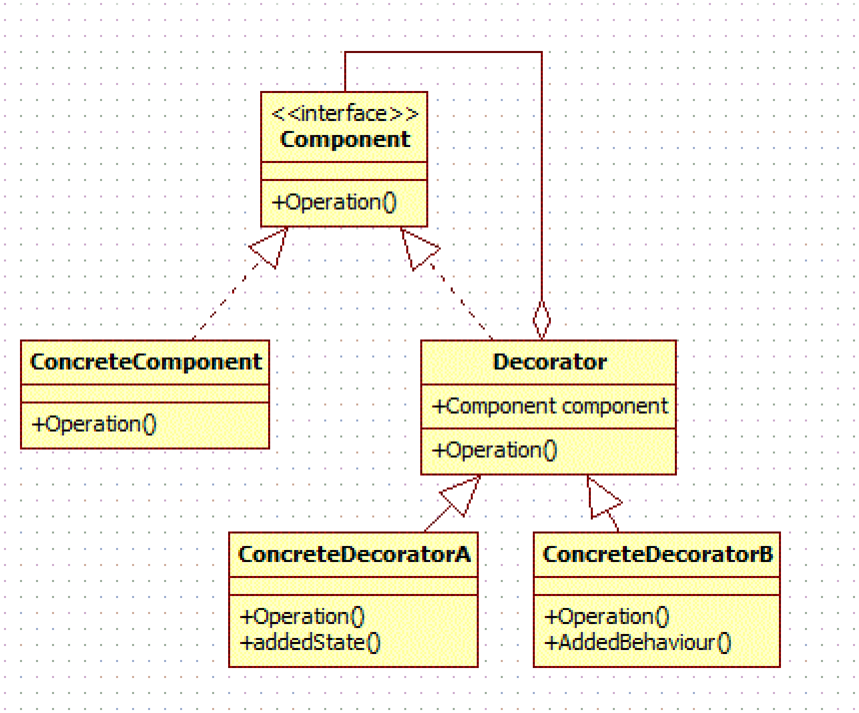
Limitations of inheritance

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



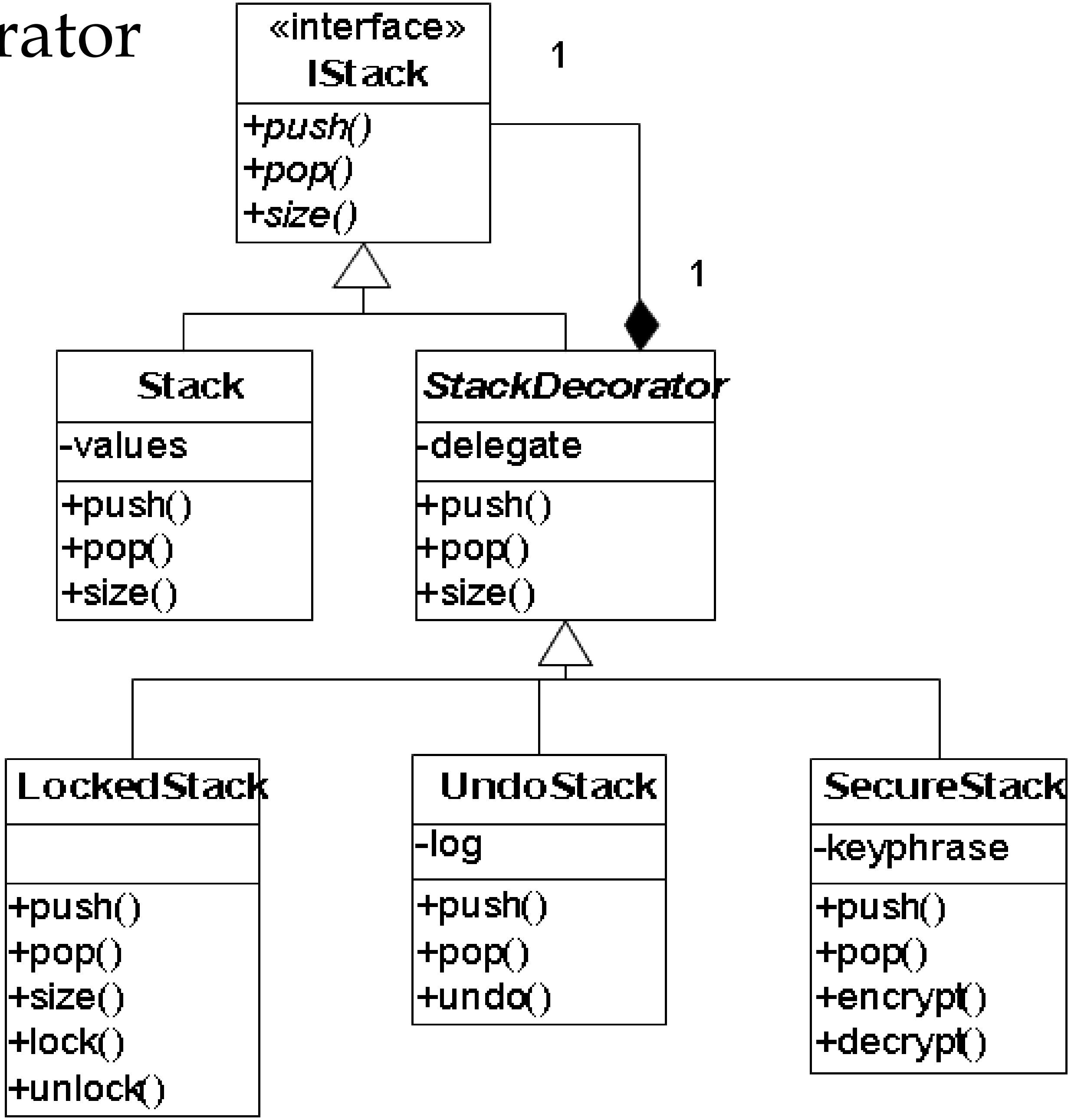
Decorator

- **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

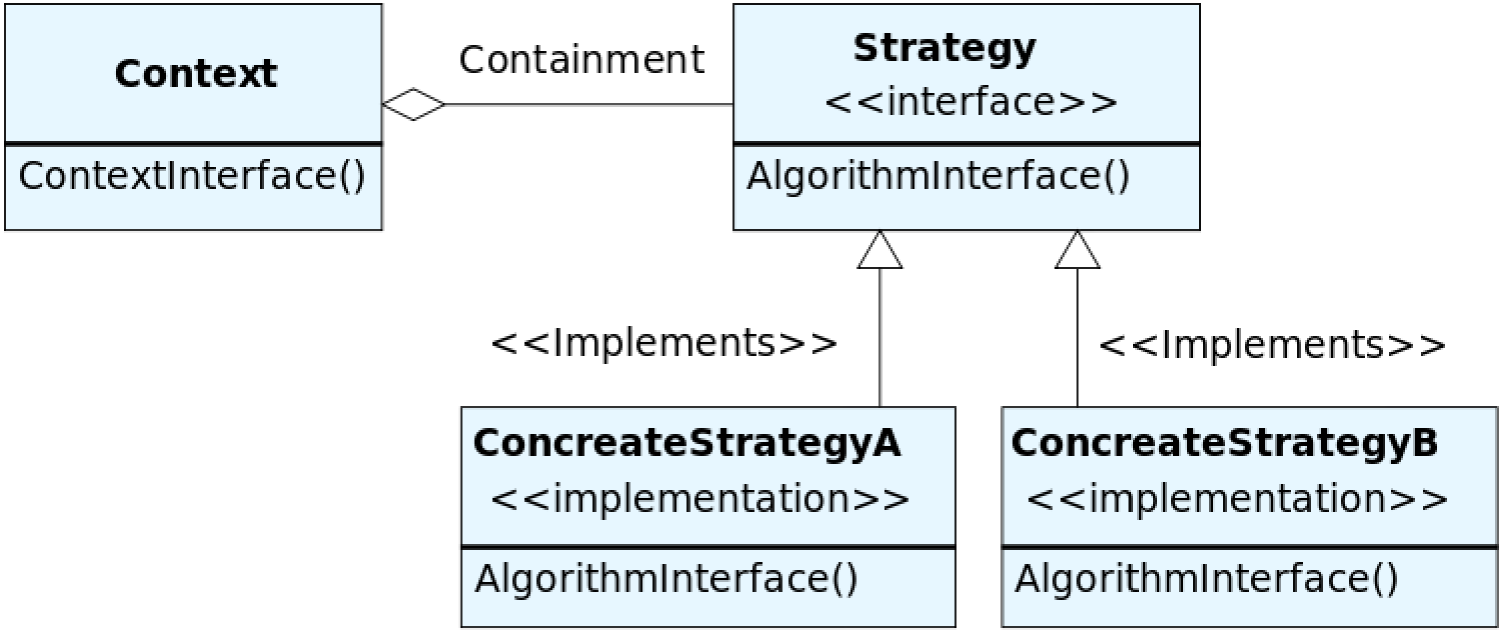
(1) Strategy

策略模式

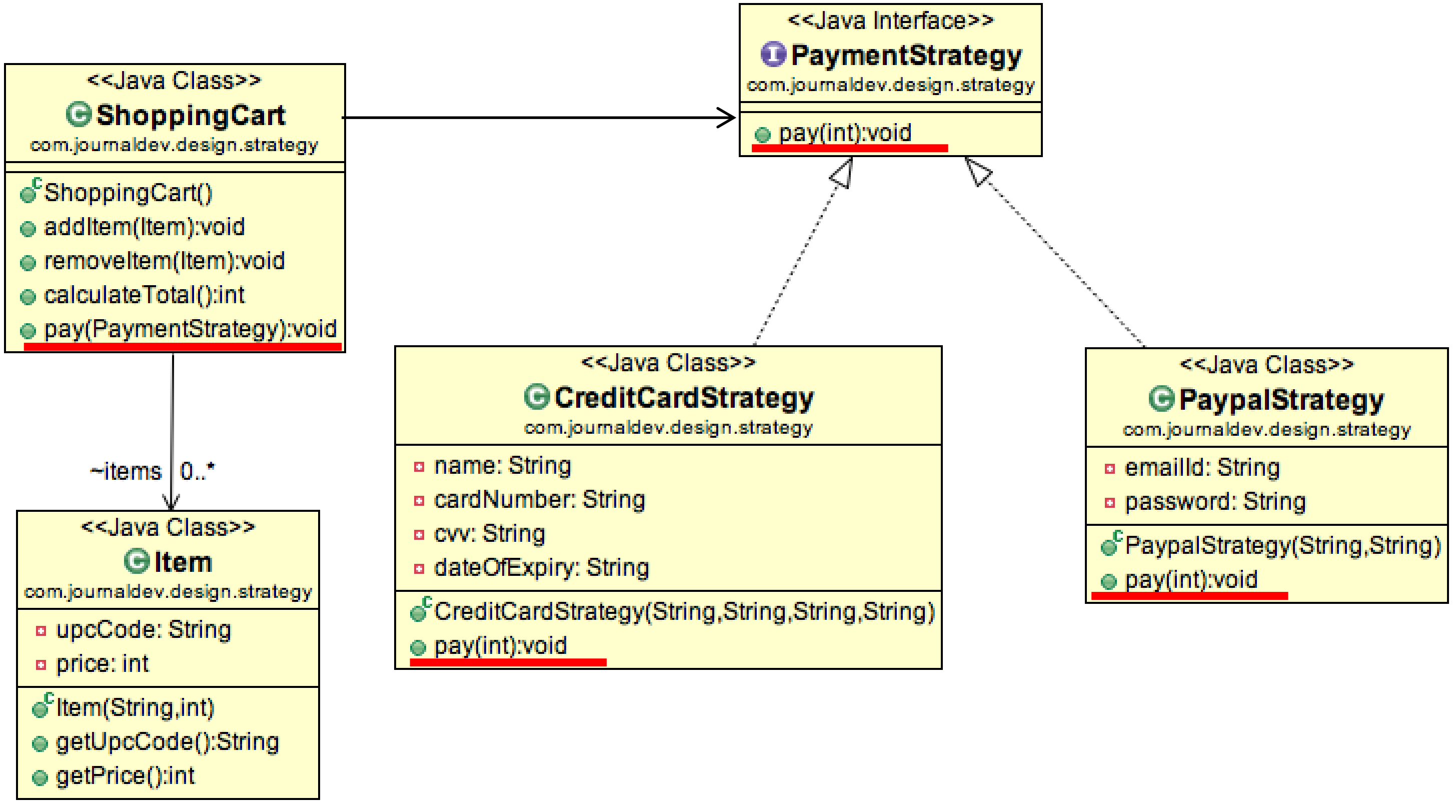
Strategy Pattern

- **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

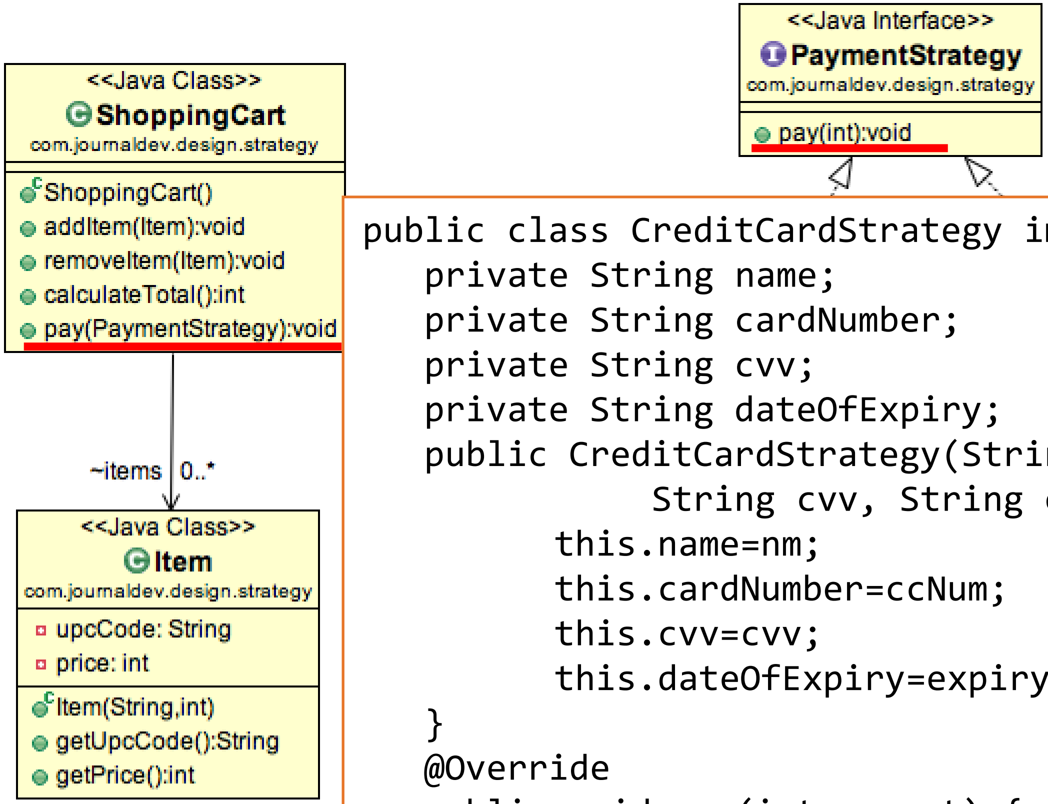


Code example



Code example

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

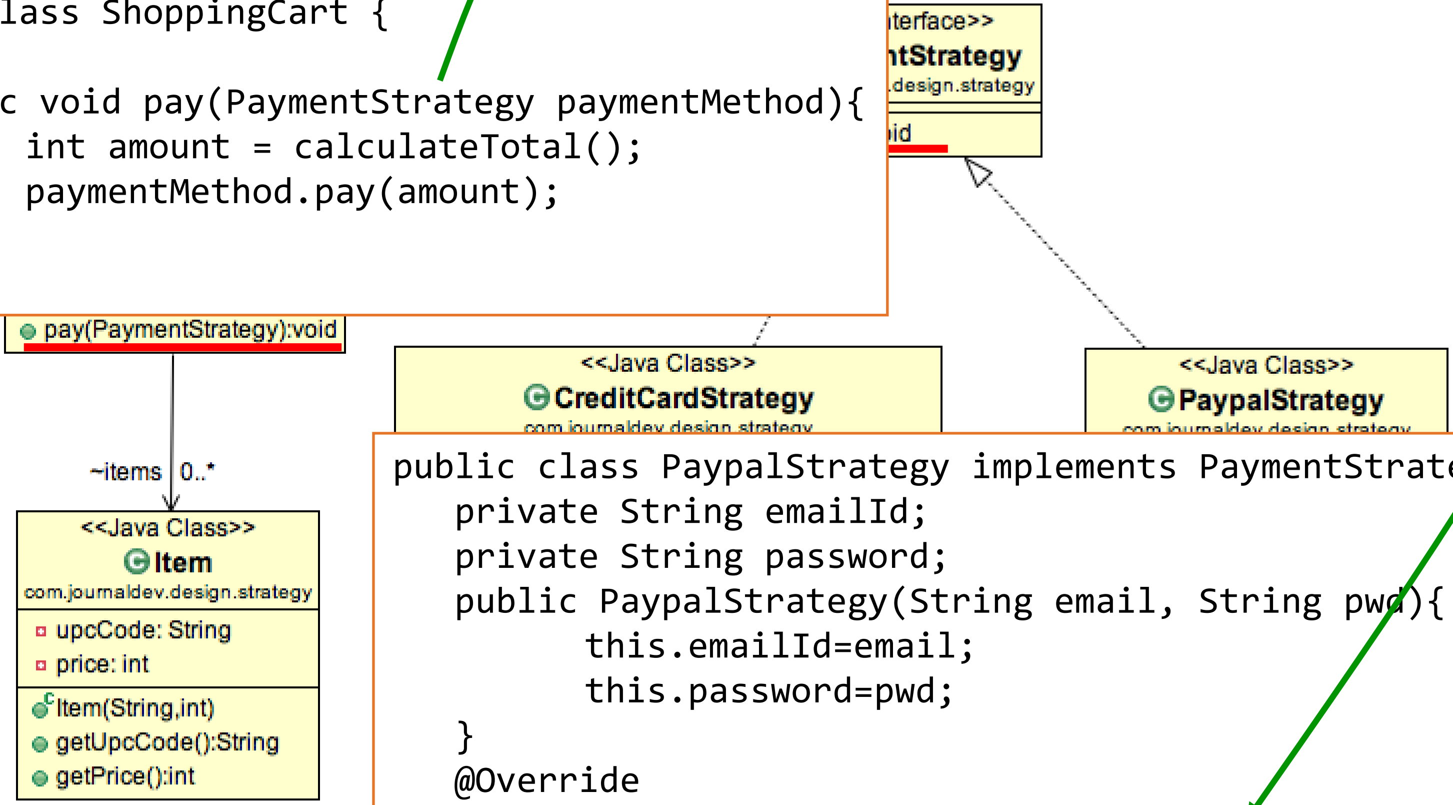


```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
              String cvv, String expiryDate){
         this.name=nm;
         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);
   }
}
```



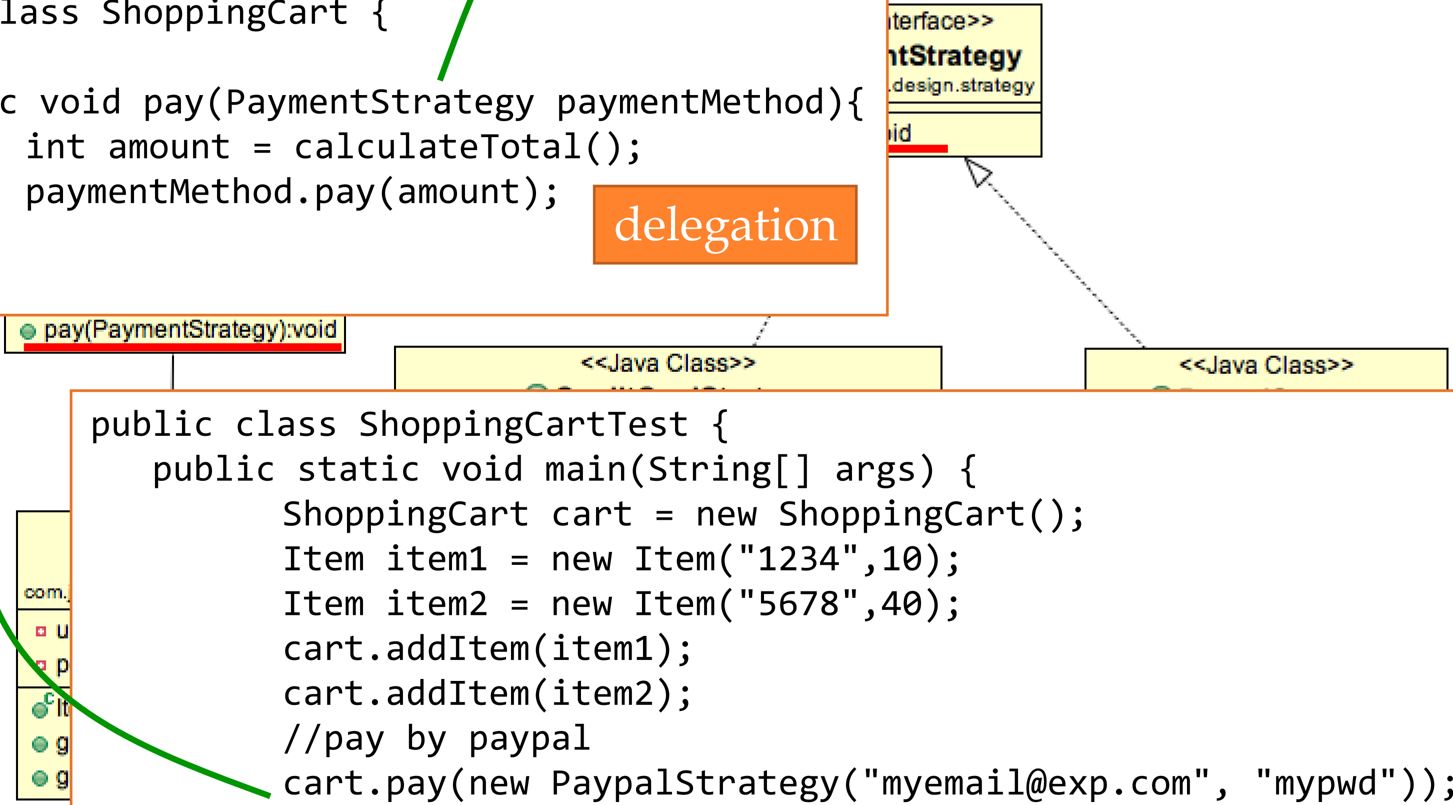
```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
         this.emailId=email;
         this.password=pwd;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount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

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
- **Examples:**
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types

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

- **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.

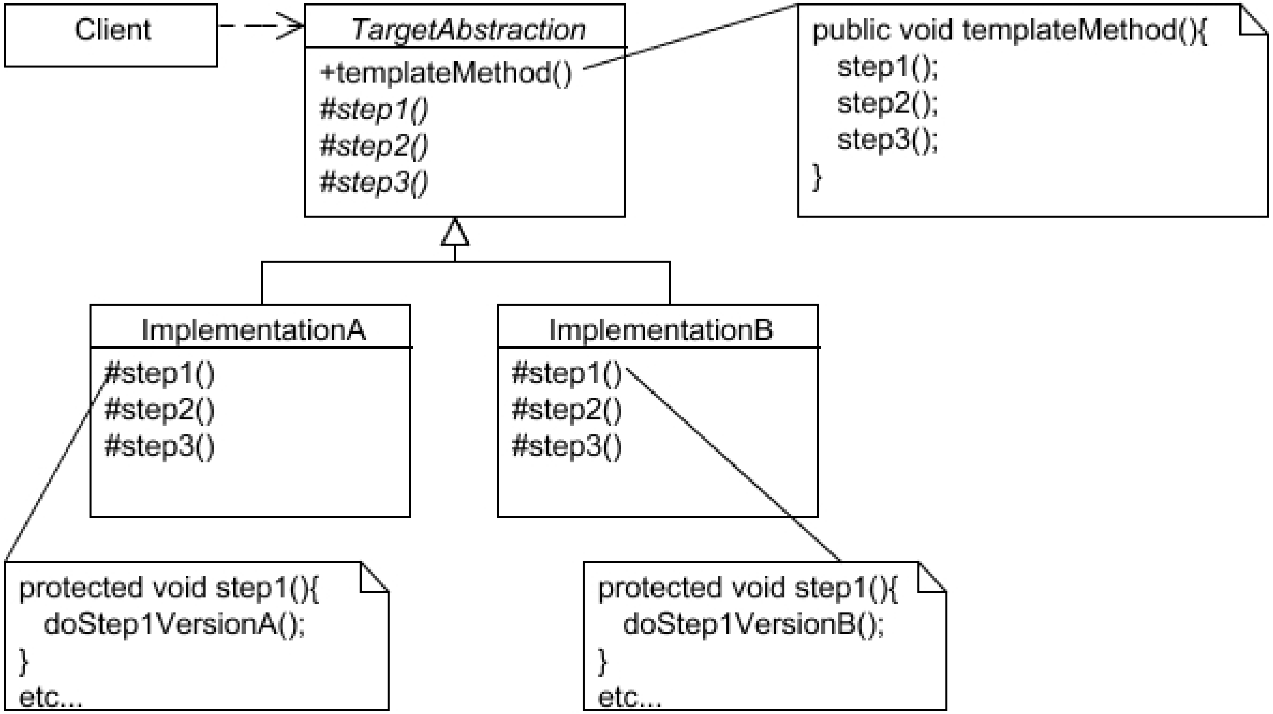
Template Method Pattern

- **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

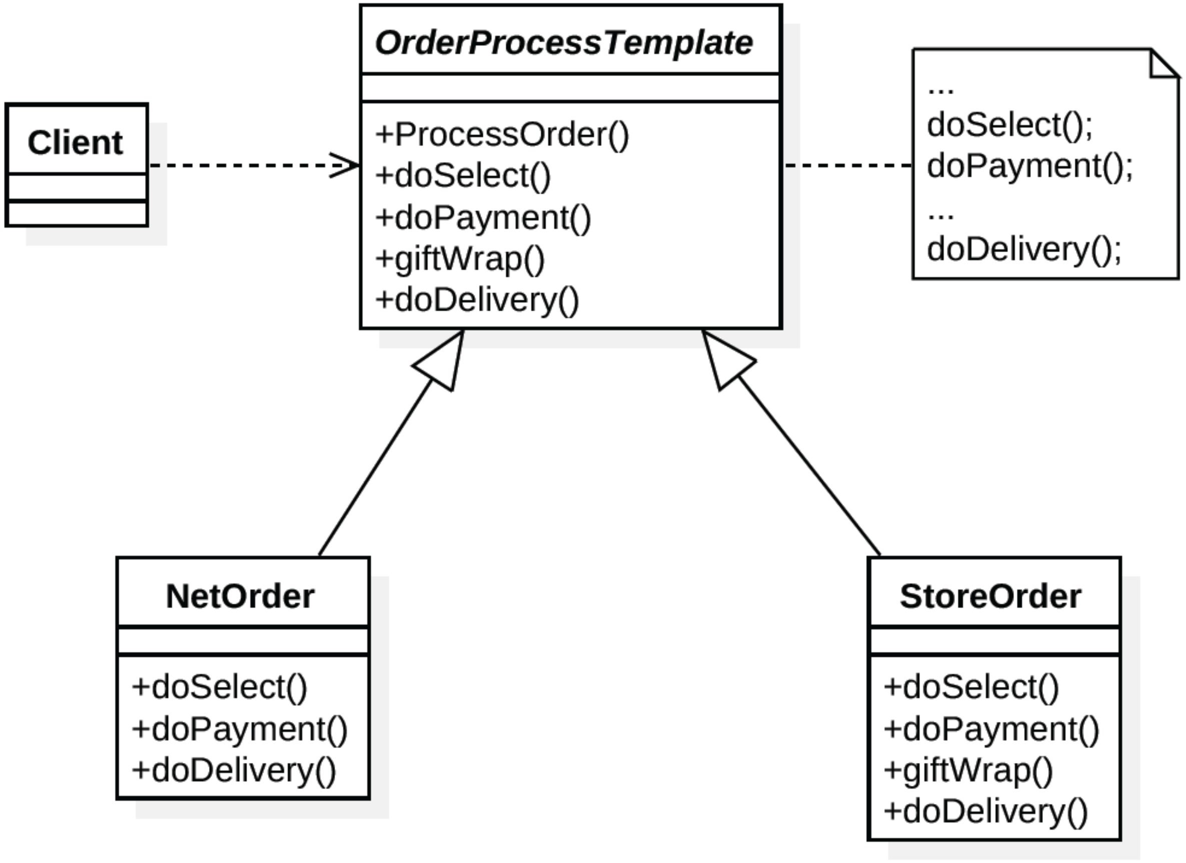


```
public void templateMethod(){
   step1();
   step2();
   step3();
}
```

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

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

Example



Client

OrderProcessTemplate

+ProcessOrder()
+doSelect()
+doPayment()
+giftWrap()
+doDelivery()

...
doSelect();
doPayment();
...
doDelivery();

NetOrder

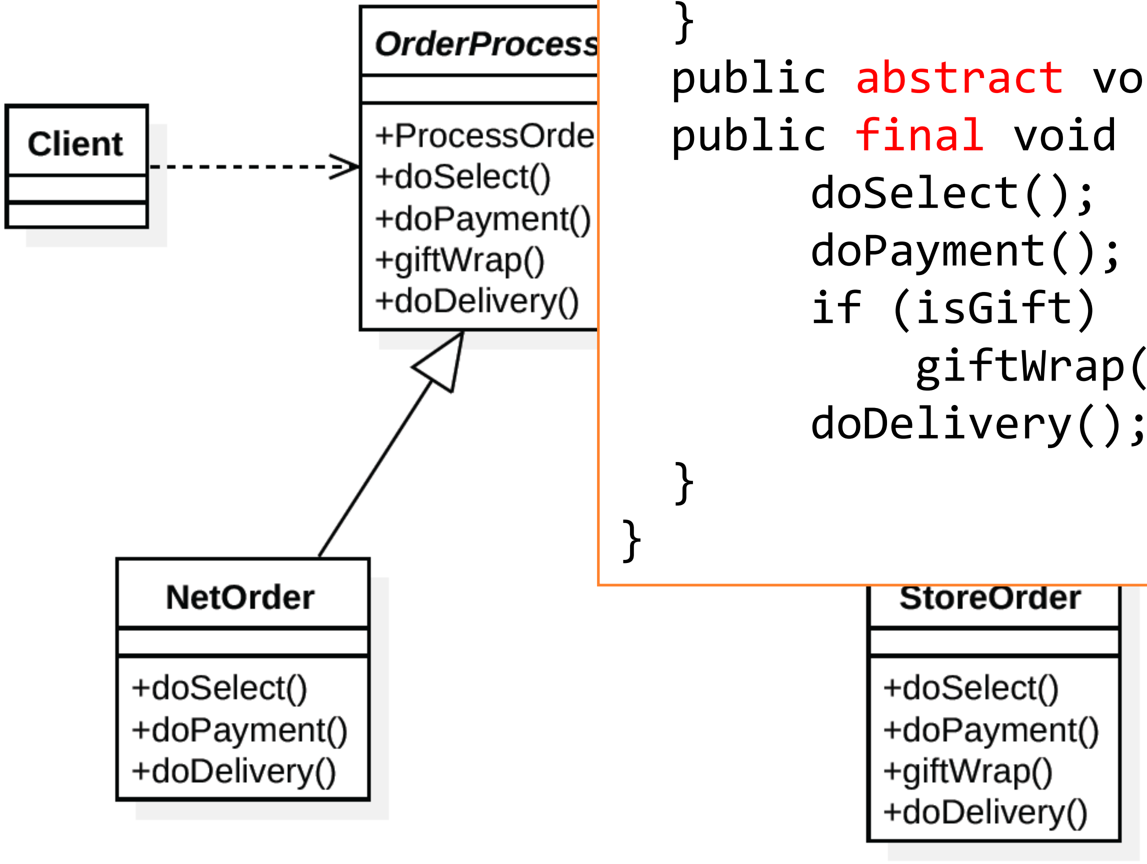
+doSelect()
+doPayment()
+doDelivery()

StoreOrder

+doSelect()
+doPayment()
+giftWrap()
+doDelivery()

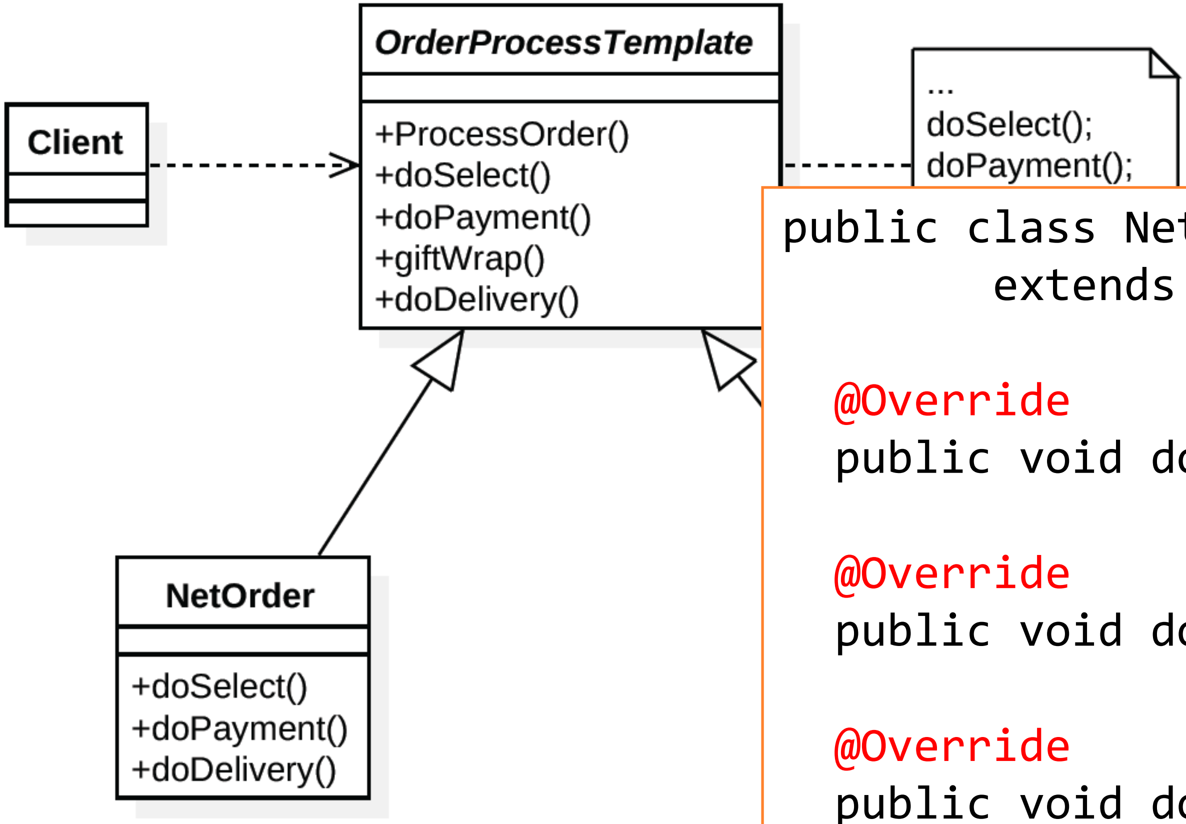
Example

```
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

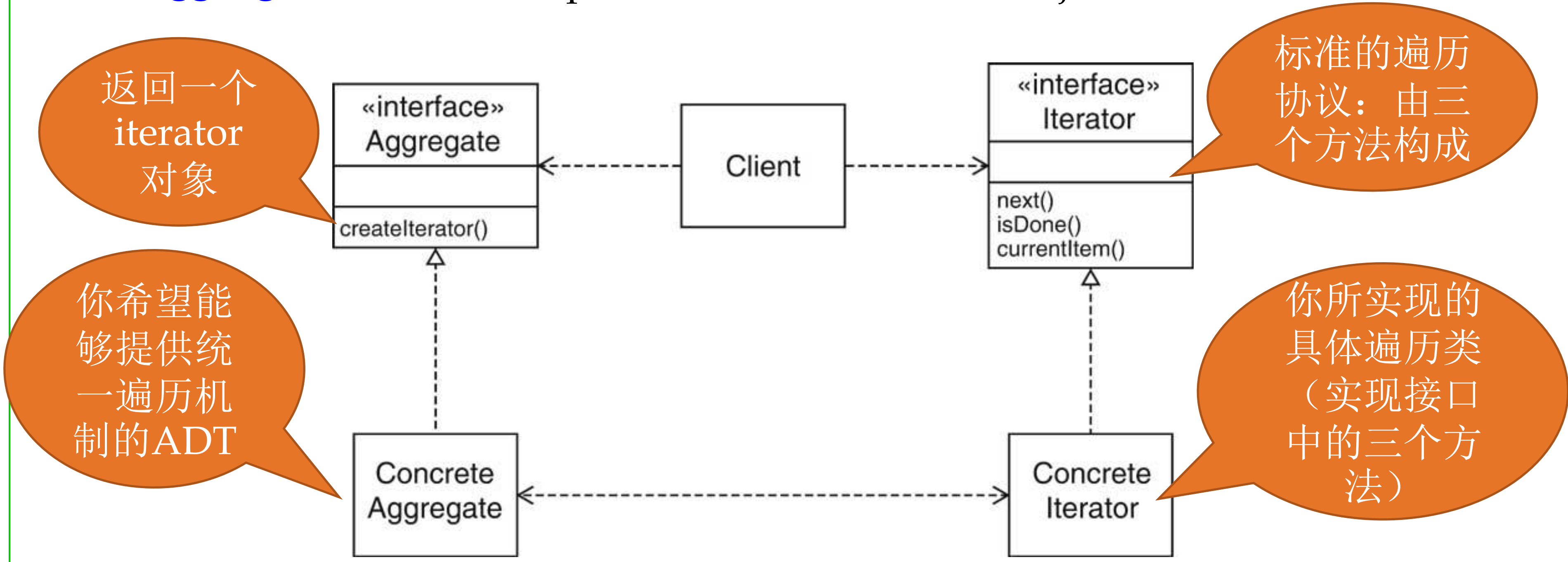
(3) Iterator

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 instance creates instances of Iterator objects
  - Aggregate instance keeps reference to Iterator object



返回一个iterator对象

标准的遍历协议：由三个方法构成

你希望能够提供统一遍历机制的ADT

你所实现的具体遍历类（实现接口中的三个方法）

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();
      }
   }
}
```

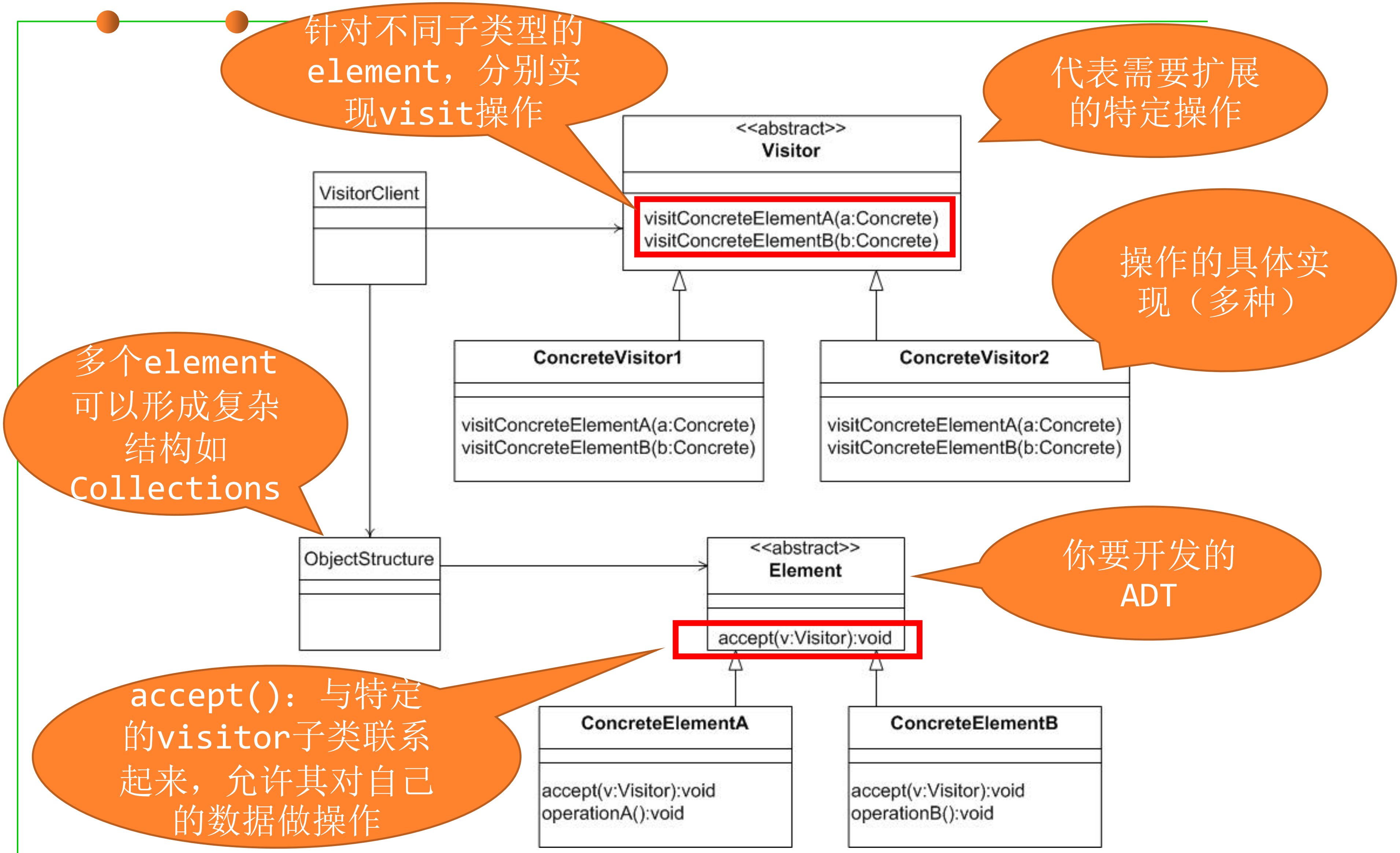
```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

(4) Visitor

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

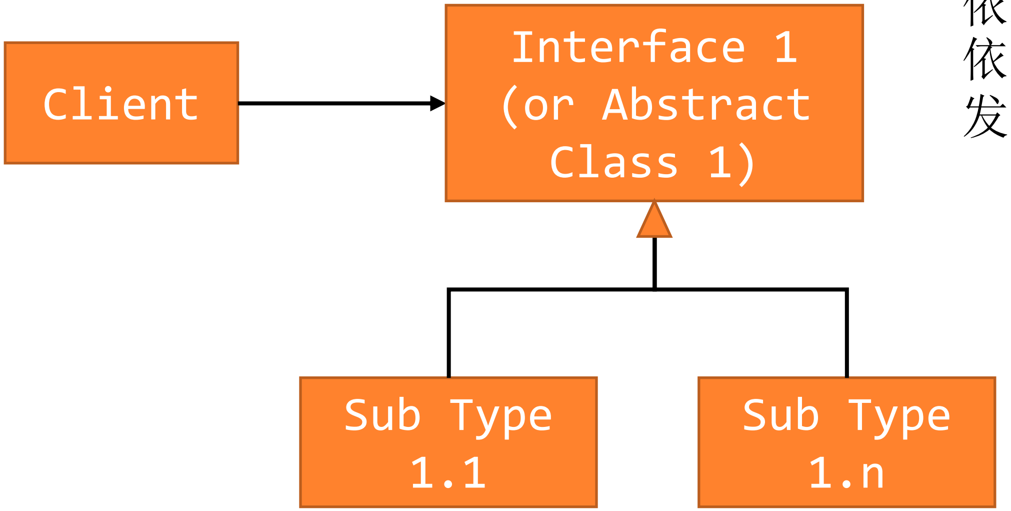
Strategy vs visitor

- **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”

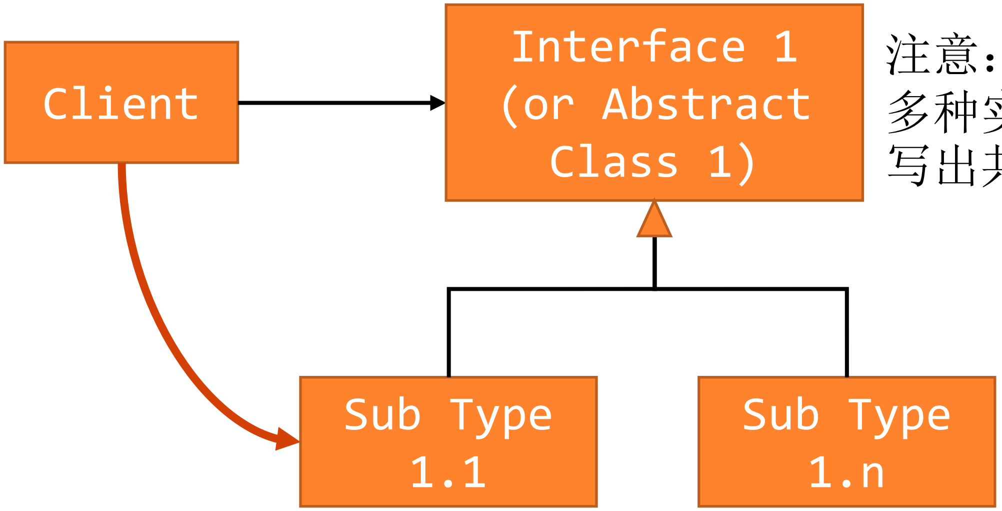
发生变化时最好是“扩展”而不是“修改”

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”

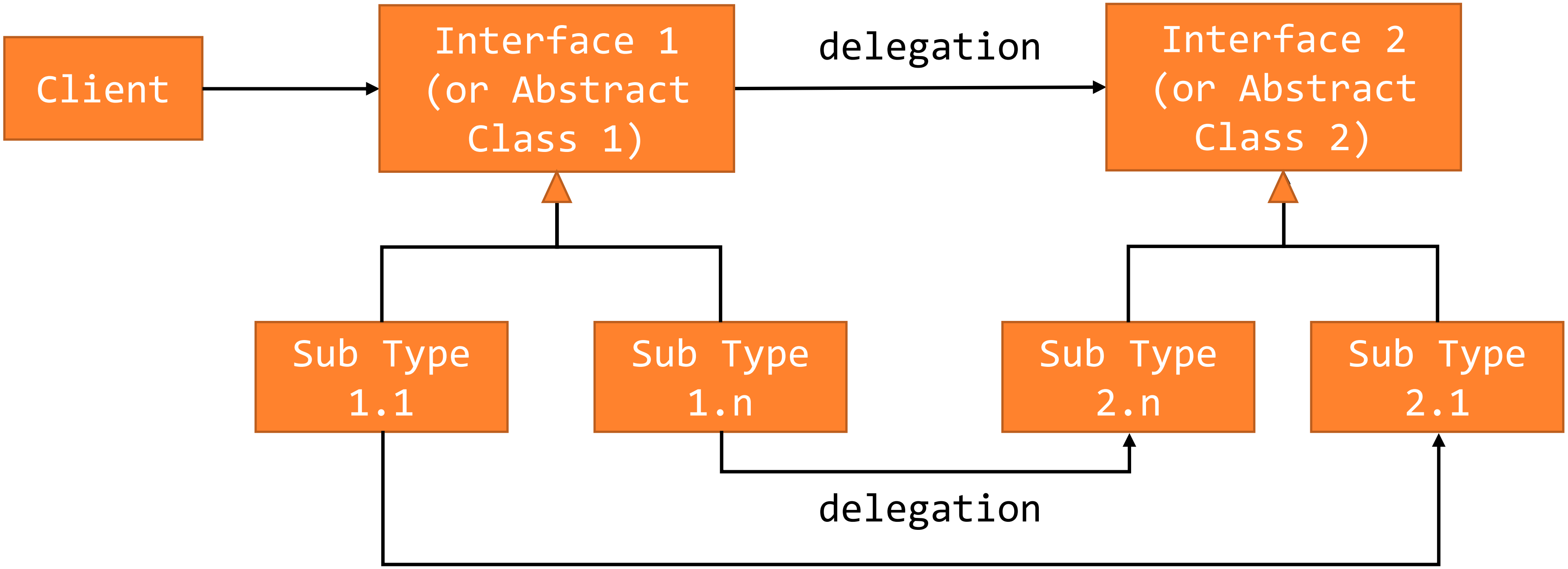


注意：如果某个步骤不需要有多种实现，直接在该抽象类里写出共性实现即可。

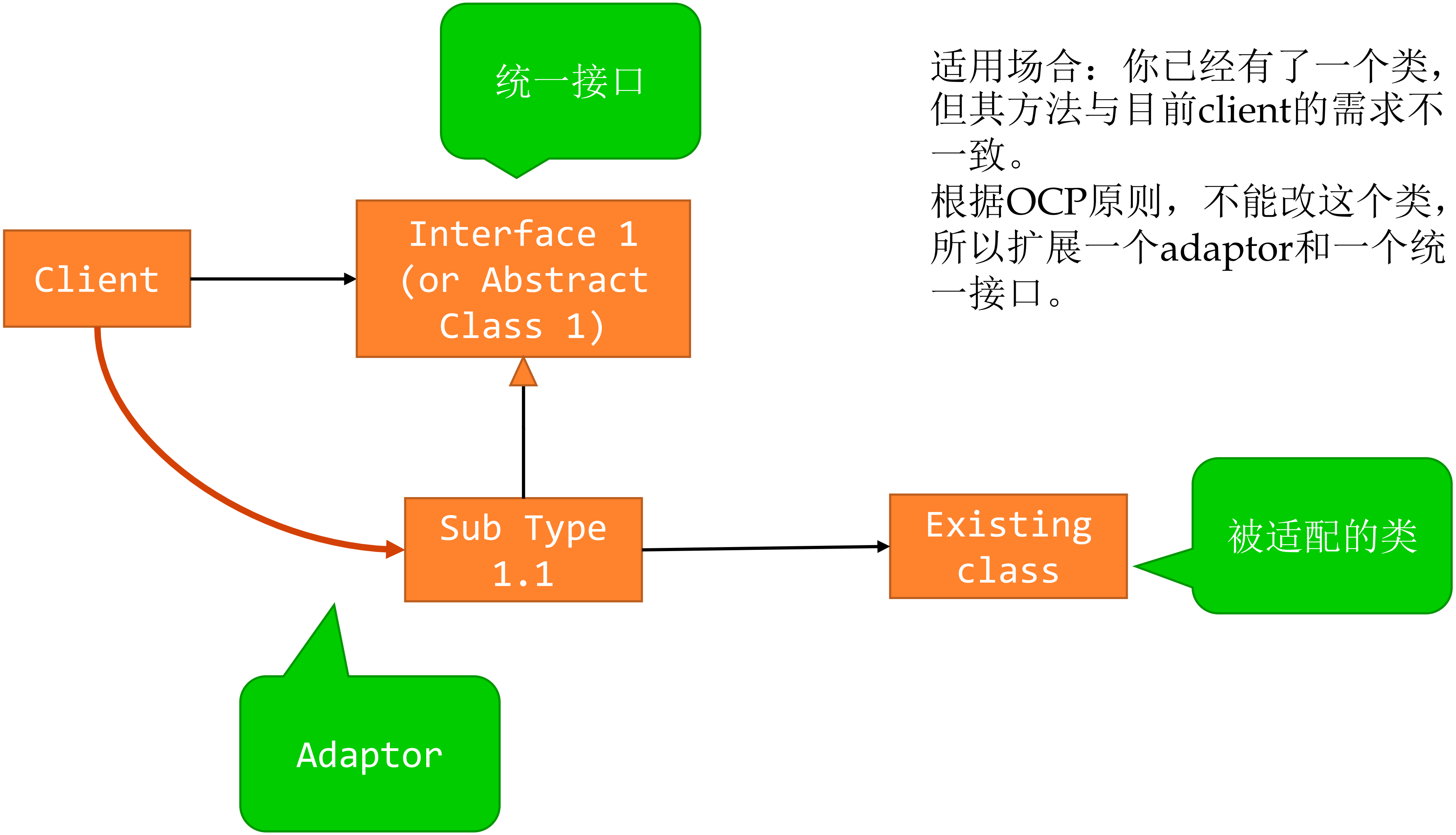
每个子类型，只需要实现上面的各个abstract方法即可。

设计模式的对比：共性样式2

两棵“继承树”，两个层次的“delegation”



Adaptor



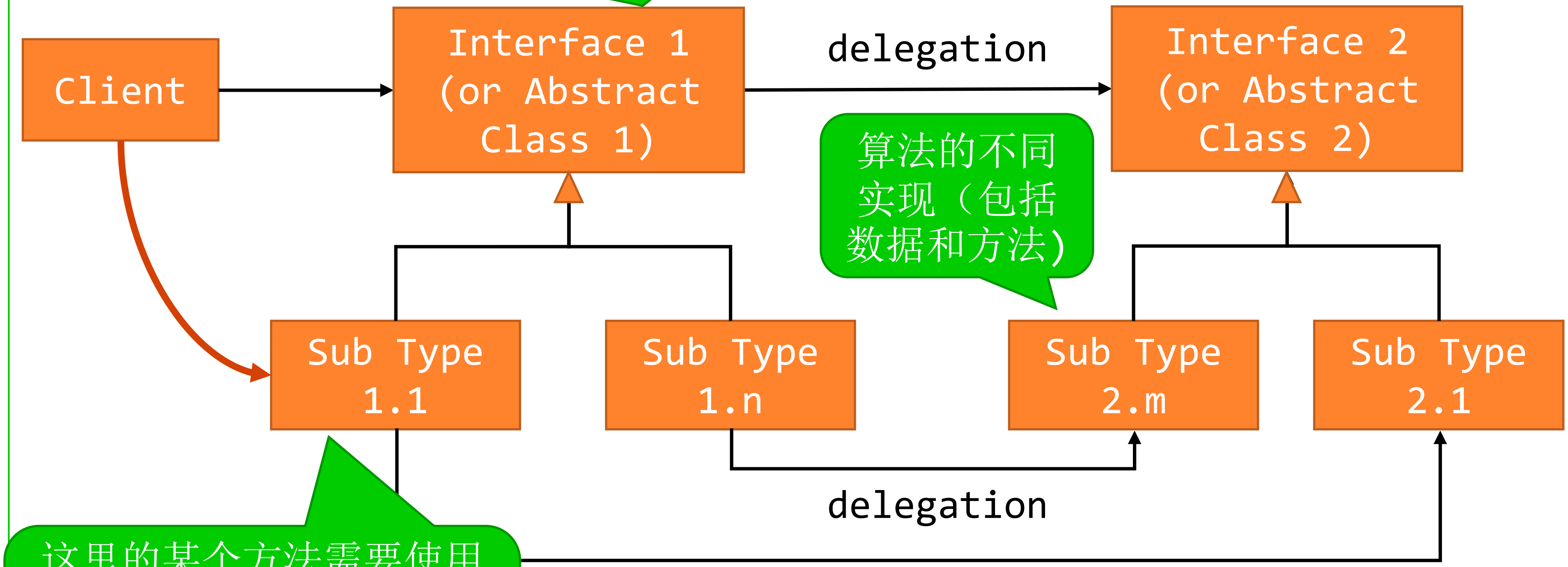
适用场合：你已经有了一个类，但其方法与目前client的需求不一致。

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。

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



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

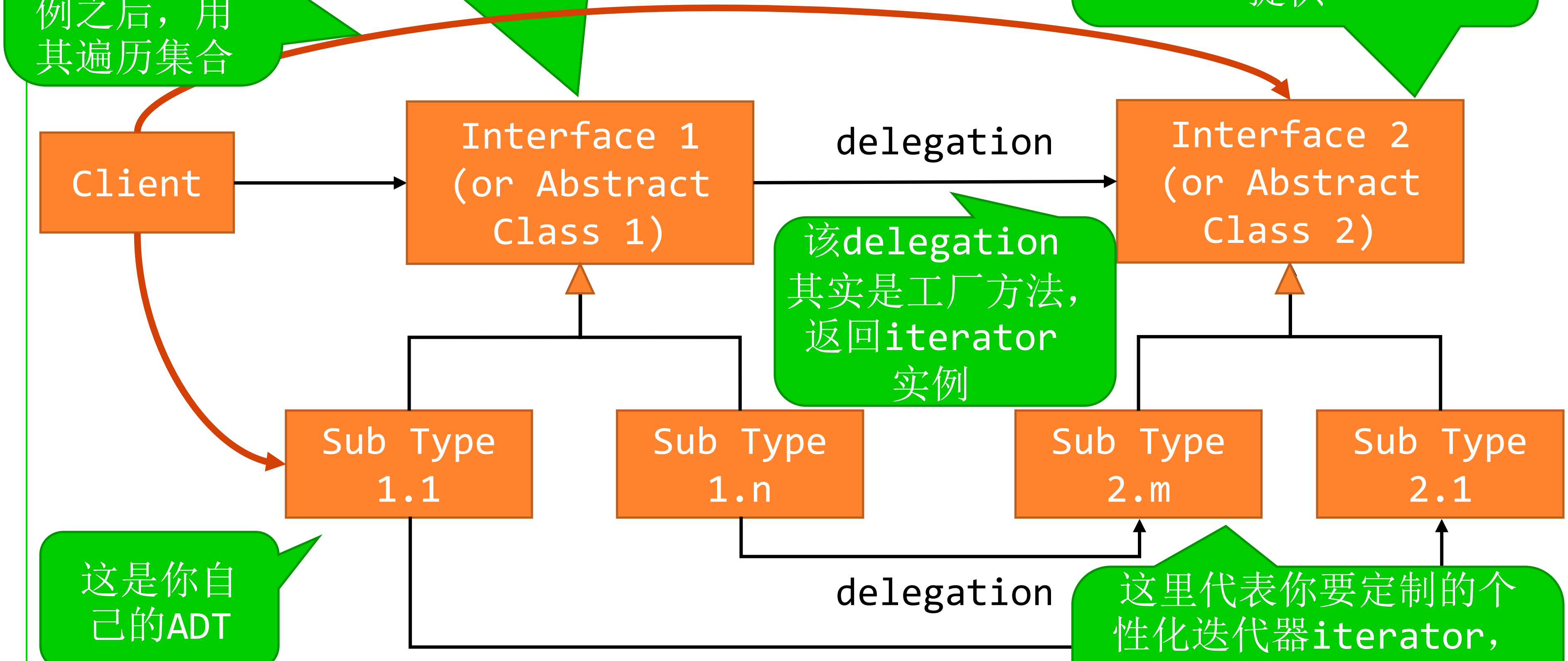
左右两侧的两棵树的子类型，不需要一一对应

Iterator

该关系是指：client拿到Iterator实例之后，用其遍历集合

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

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)

Sub Type 1.1

Sub Type 1.n

Sub Type 2.m

Sub Type 2.1

delegation

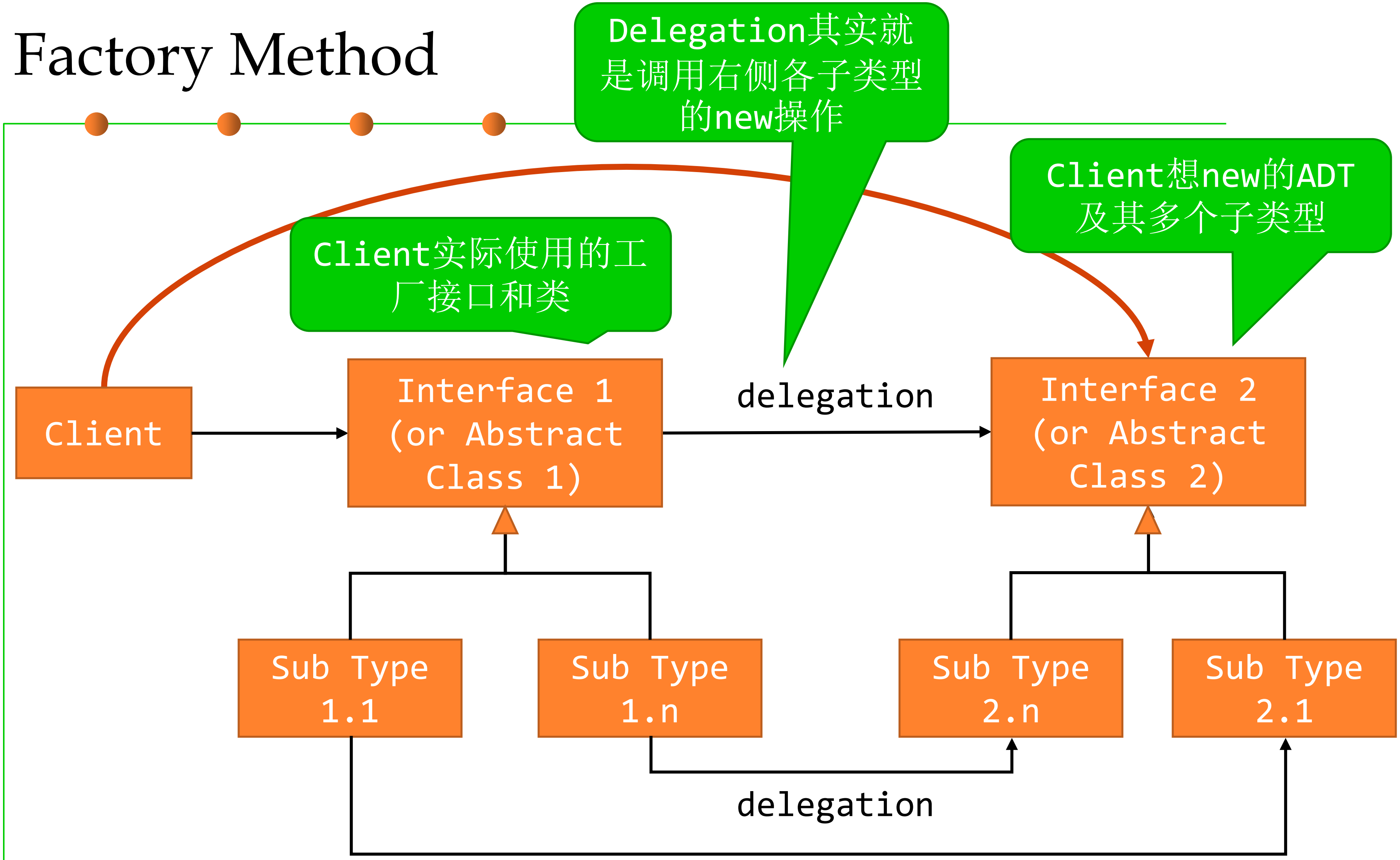
该delegation其实是工厂方法，返回iterator实例

这是你自己的ADT

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

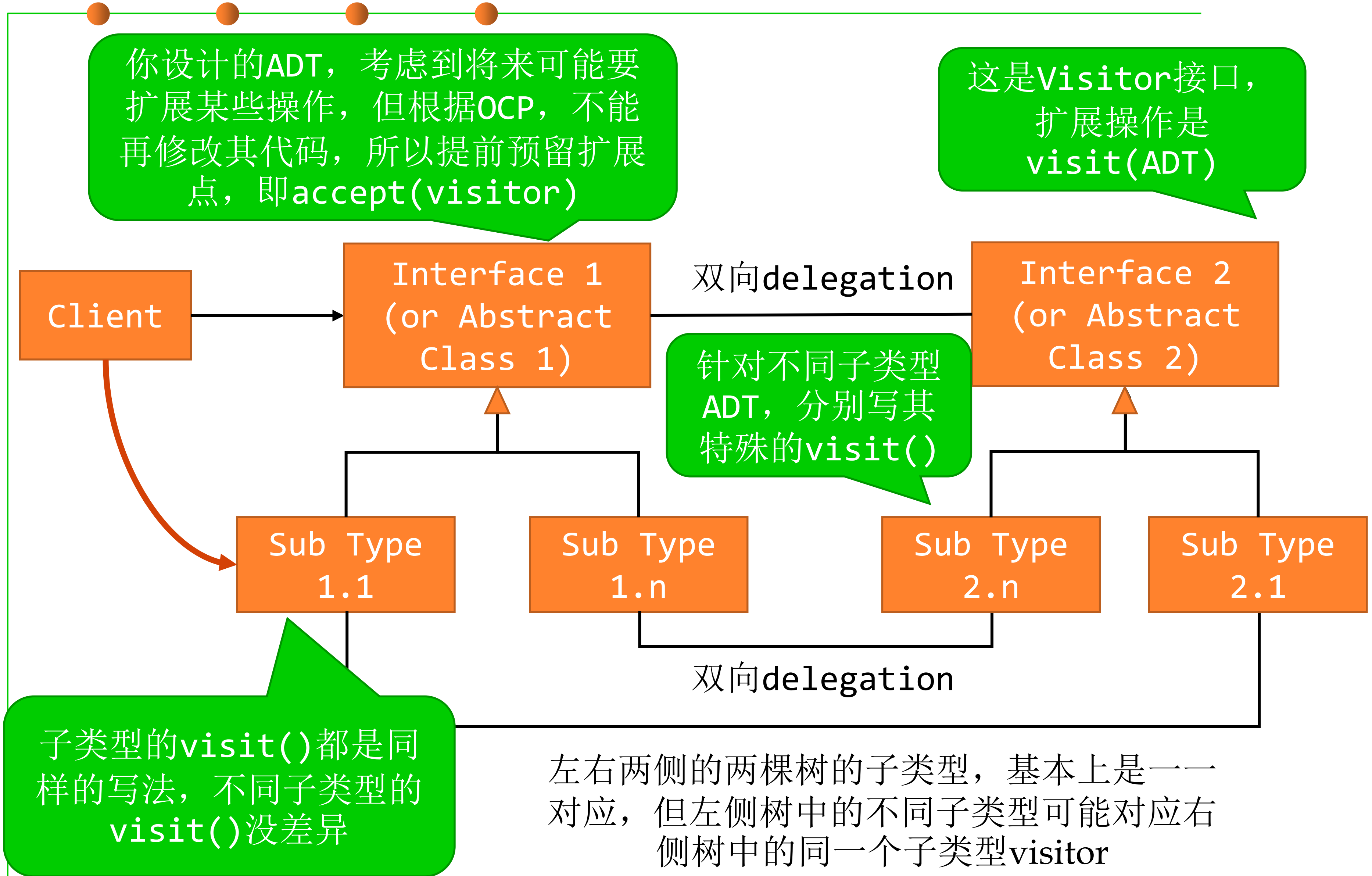
在该模式里，左右两个树里，其实分别只有一个子类型

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)

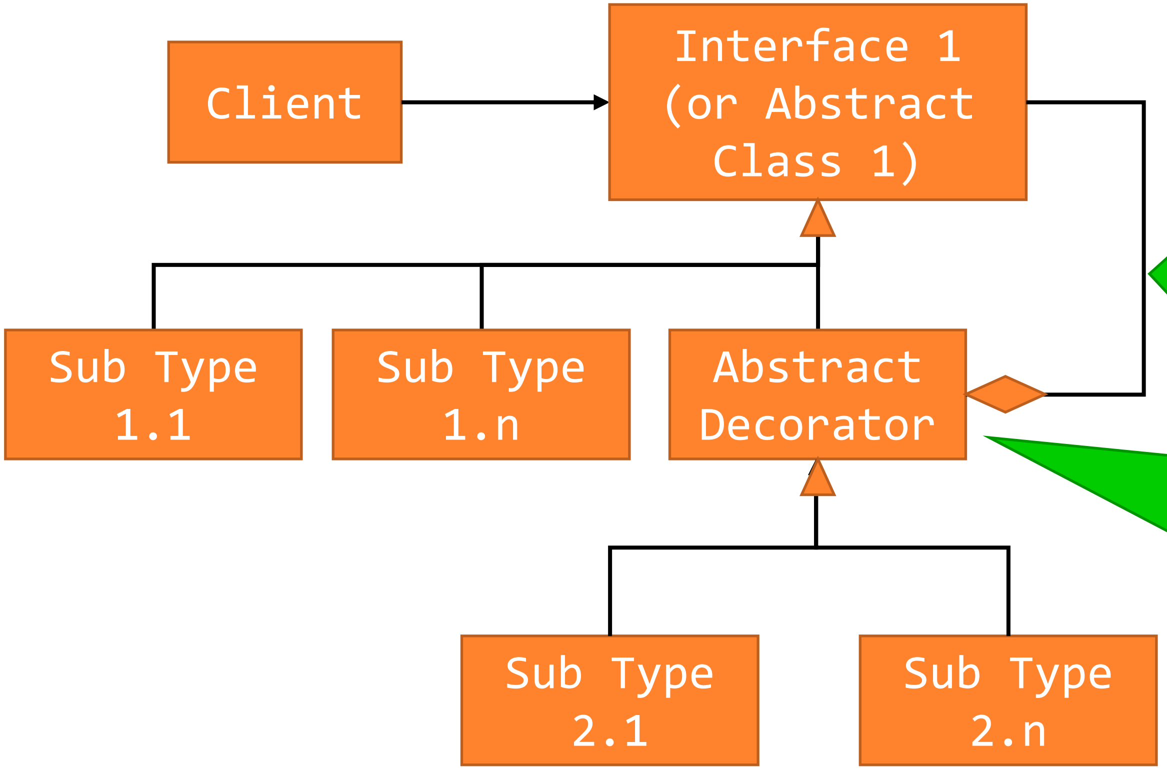
针对不同子类型ADT，分别写其特殊的visit()

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

Decorator

所以这其实是个递归的设计，多个装饰器子类型可以嵌套的调用，实现对原始ADT对象的多次装饰。
但不管怎么装饰，永远是属于原始ADT的子类型



Delegation到ADT接口，所以这里既有继承，又有delegation

继承：具有ADT的所有基本功能，但通过override来扩展功能以实现装饰；
委派：具有一个上面ADT接口类型的变量，通过该变量调用基本功能。

Summary

Summary

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

The end

April 21, 2024