哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 5: Reusability-Oriented Software Construction Approaches

5.3 Design Patterns for Reuse

Wang Zhongjie
rainy@hit.edu.cn

April 10, 2018

Outline

除了Framework，5-2节所讨论的其他复用技术都过于“基础”和“细小”，有没有办法做更大规模的复用设计？
本节：几种典型的“面向复用”的设计模式

- **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 an existing method of an object.
  - Facade provides a simplified interface to a large body of code.
- **Behavioral patterns**
  - Strategy allows one of a family of algorithms to be selected on-the-fly 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.

Reading

- **CMU 15-214：06、07**
- **设计模式：第1、2章；第4.1、4.4、4.5、5.4、5.9、5.10节**

Why reusable design patterns?

A design…

…enables flexibility to change (reusability)

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

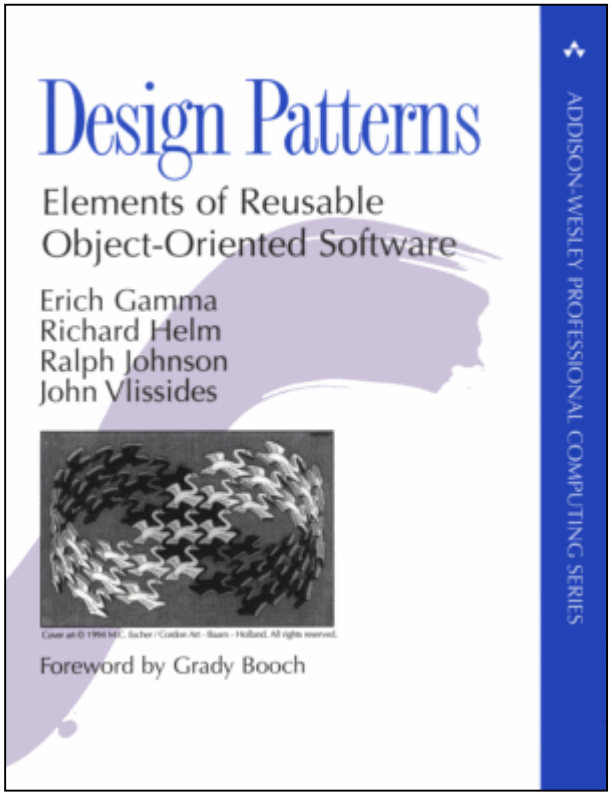
…allows the delivery of more functionality after an initial delivery (extensibility).

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 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.

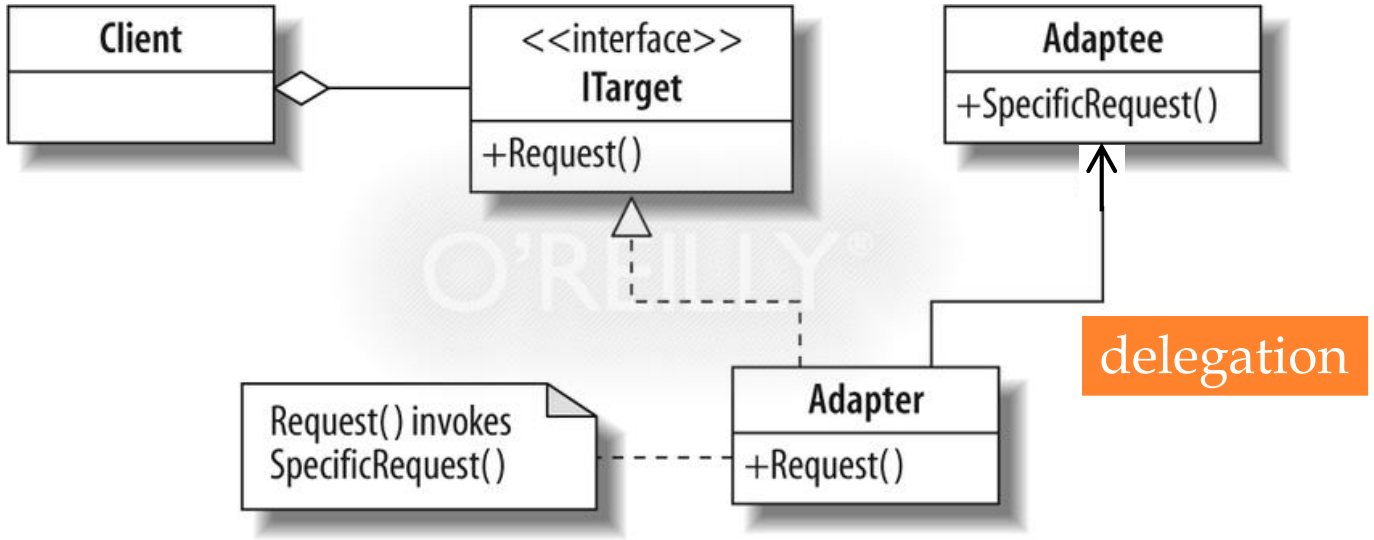
1 Structural patterns

(1) Adapter

适配器模式

Adapter Pattern

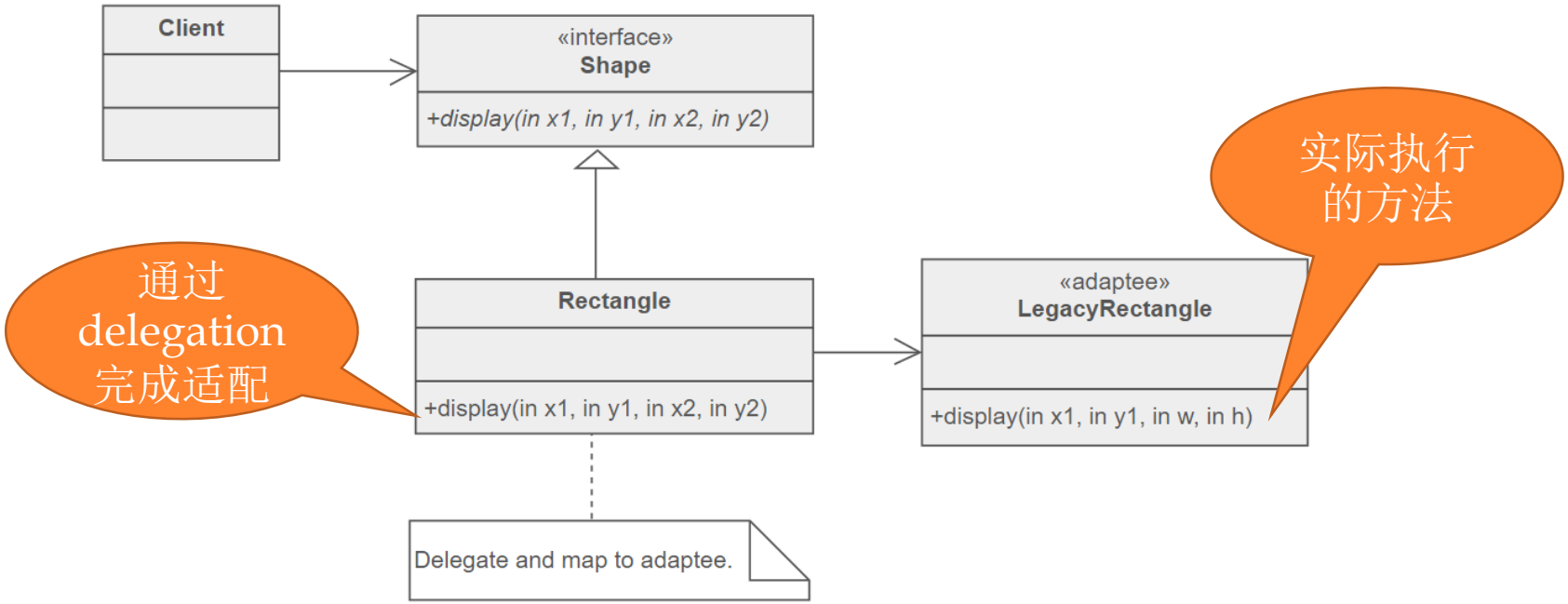
- **Intent: Convert the interface of a class into another interface that clients expect to get. 将某个类/接口转换为client期望的其他形式**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface. 通过增加一个接口，将已存在的子类封装起来，client面向接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



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);
  }
}
```

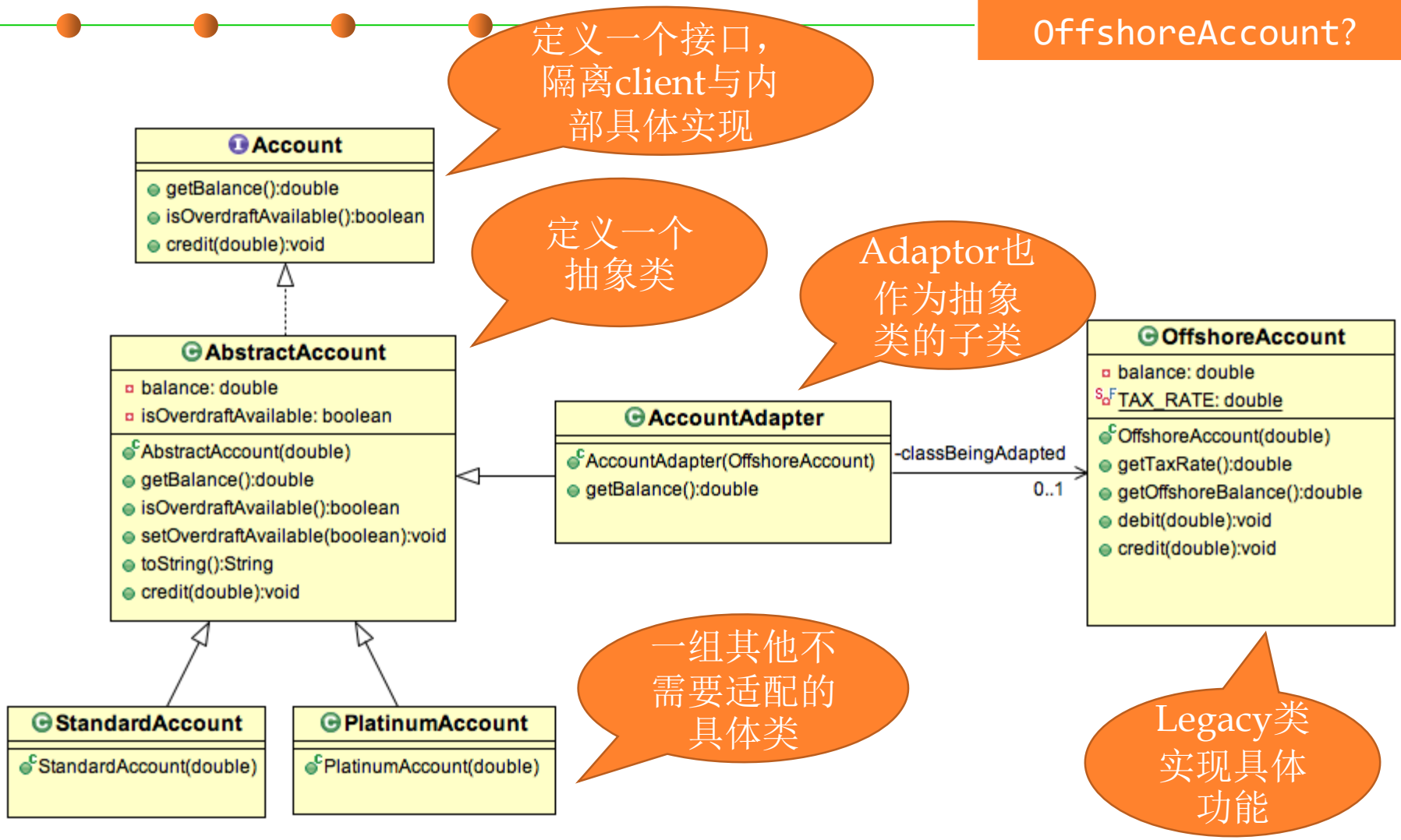
Adaptor类实现抽象接口

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`?



(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

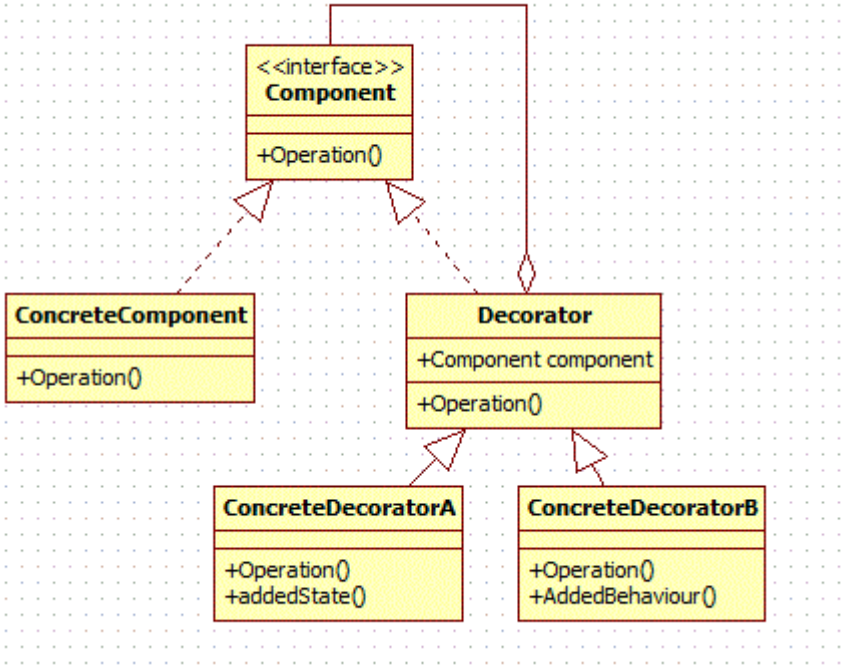
Inheritance

- **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

- **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**



The AbstractStackDecorator Class

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

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

The concrete decorator classes

```
public class UndoStack
        extends AbstractStackDecorator
        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:
  - `UndoStack s = new UndoStack(new ArrayStack());`
- To construct a secure synchronized undo stack:
  -
```
SecureStack s = new SecureStack(
                    new SynchronizedStack(
                        new UndoStack(s))
```

- **Flexibly Composable!**

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

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);`
- **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 3-1

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) Facade

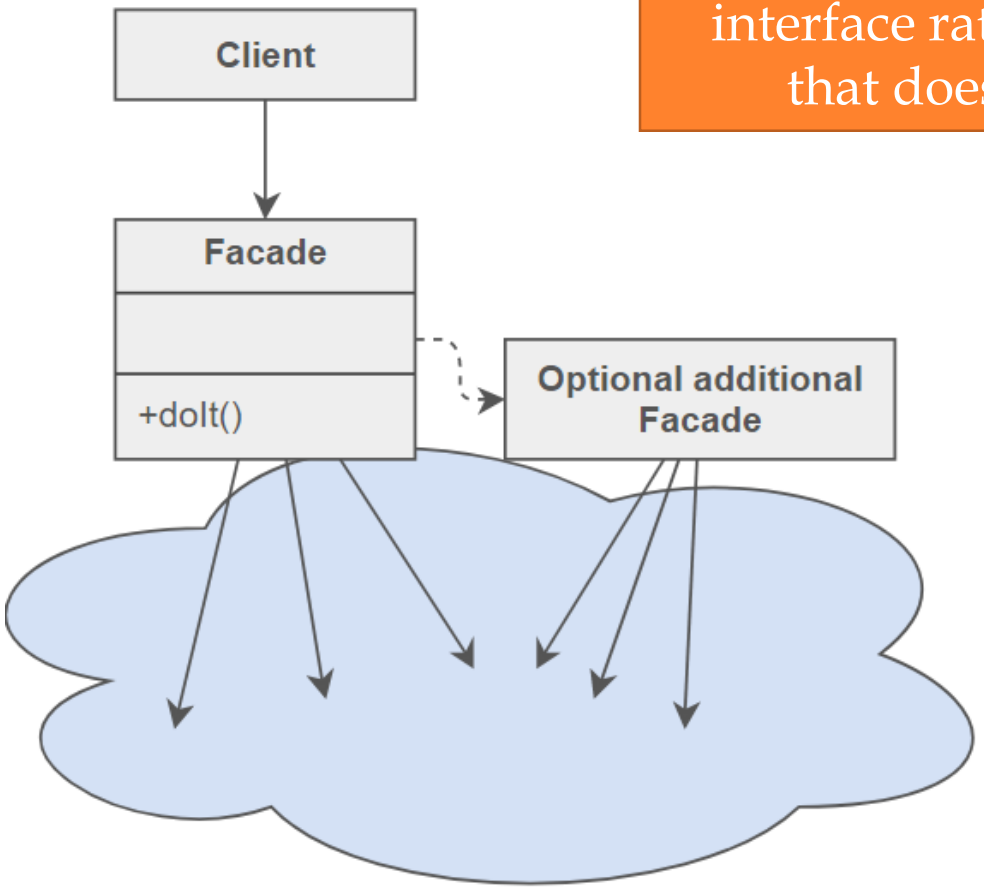
外观模式

Facade

- **Problem: a segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.** 客户端需要通过一个简化的接口来访问复杂系统内的功能
- **Intent** 提供一个统一的接口来取代一系列小接口调用，相当于对复杂系统做了一个封装，简化客户端使用
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.
- **This reduces the learning curve necessary to successfully leverage the subsystem.** 便于客户端学习使用，解耦
- **It also promotes decoupling the subsystem from its potentially many clients.**

Facade

Facade pattern is applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.



Façade Example

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

Two Helper Classes for MySQL and Oracle

分别封装了客户端所需的功能

```
public class MySqlHelper {

  public static Connection getMySqlDBConnection() {…}
  public void generateMySqlPDFReport
              (String tableName, Connection con){…}
  public void generateMySqlHTMLReport
              (String tableName, Connection con){…}
}

public class OracleHelper {

  public static Connection getOracleDBConnection() {…}
  public void generateOraclePDFReport
              (String tableName, Connection con){…}
  public void generateOracleHTMLReport
              (String tableName, Connection con){…}
}
```

A façade class

```
public class HelperFacade {
  public static void generateReport
     (DBTypes dbType, ReportTypes reportType, String tableName){
        Connection con = null;
        switch (dbType){
        case MYSQL:
          con = MySqlHelper.getMySqlDBConnection();
          MySqlHelper mySqlHelper = new MySqlHelper();
          switch(reportType){
                case HTML:
                  mySqlHelper.generateMySqlHTMLReport(tableName, con);
                  break;
                case PDF:
                  mySqlHelper.generateMySqlPDFReport(tableName, con);
                  break;
          }
          break;
          case ORACLE: …
        }
        public static enum DBTypes      { MYSQL,ORACLE; }
        public static enum ReportTypes  { HTML,PDF;}
}
```

Client code

Without facade

```
String tableName="Employee";

Connection con = MySqlHelper.getMySqlDBConnection();
MySqlHelper mySqlHelper = new MySqlHelper();
mySqlHelper.generateMySqlHTMLReport(tableName, con);

Connection con1 = OracleHelper.getOracleDBConnection();
OracleHelper oracleHelper = new OracleHelper();
oracleHelper.generateOraclePDFReport(tableName,
```

With facade

```
HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
HelperFacade.ReportTypes.HTML, tableName);
HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
HelperFacade.ReportTypes.PDF, tableName);
```

2 Behavioral patterns

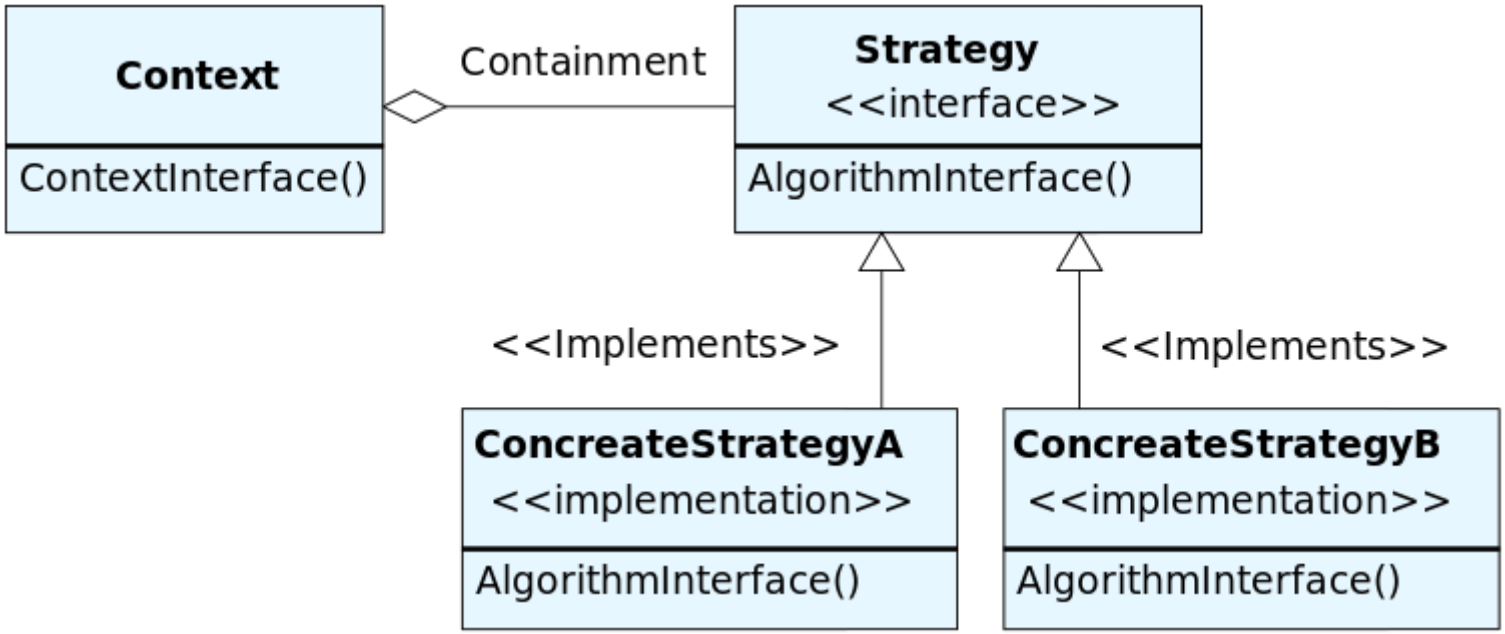
(1) Strategy

策略模式

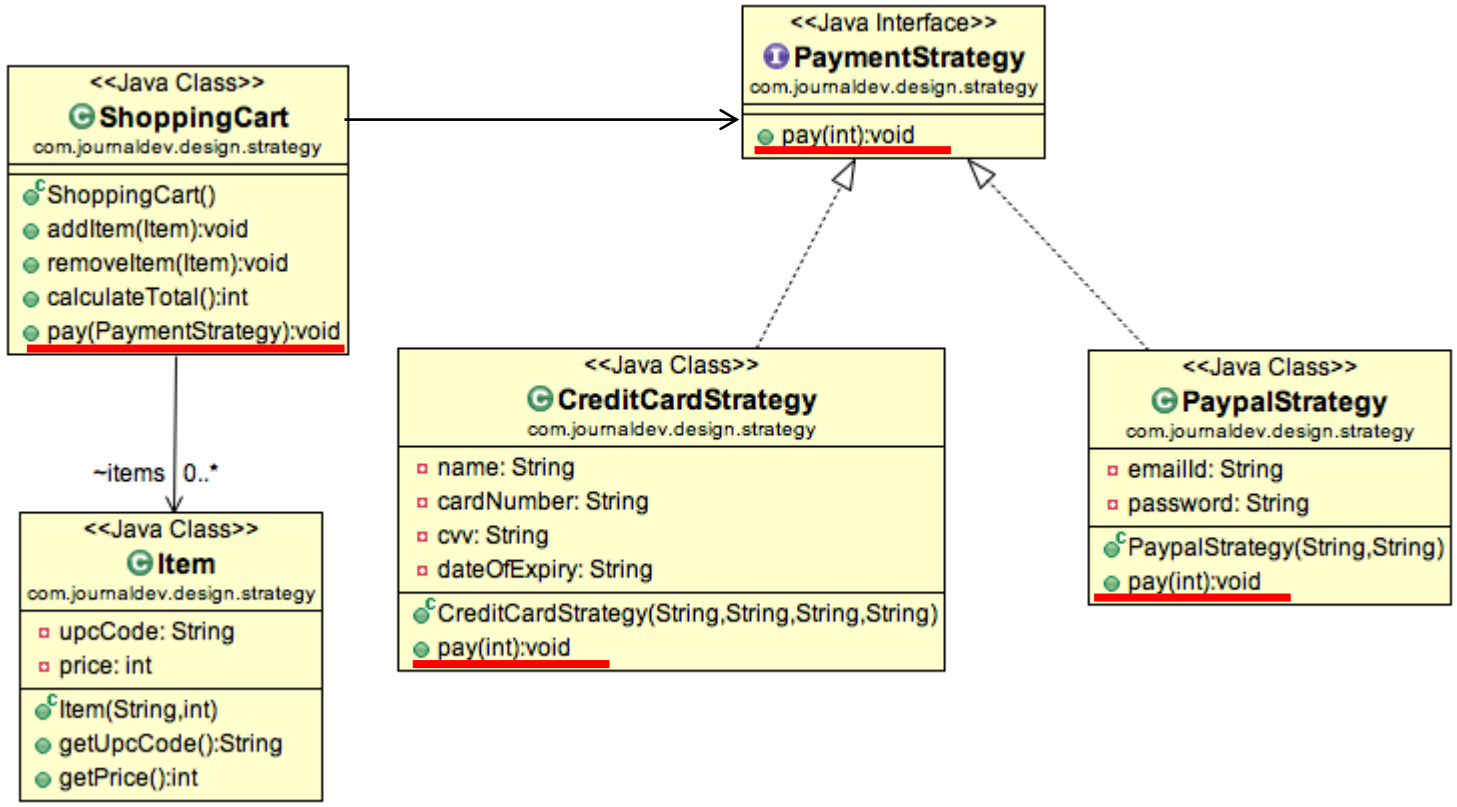
Strategy Pattern

- **Problem:** Different algorithms exists for a specific task, but client can switch between the algorithms at run time in terms of dynamic context.
- **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.
- **Advantage:**
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context

Strategy Pattern

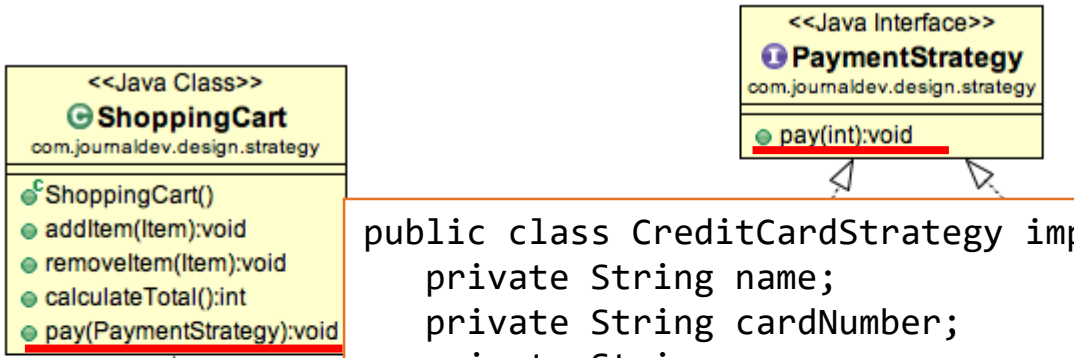


Code example



Code example

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

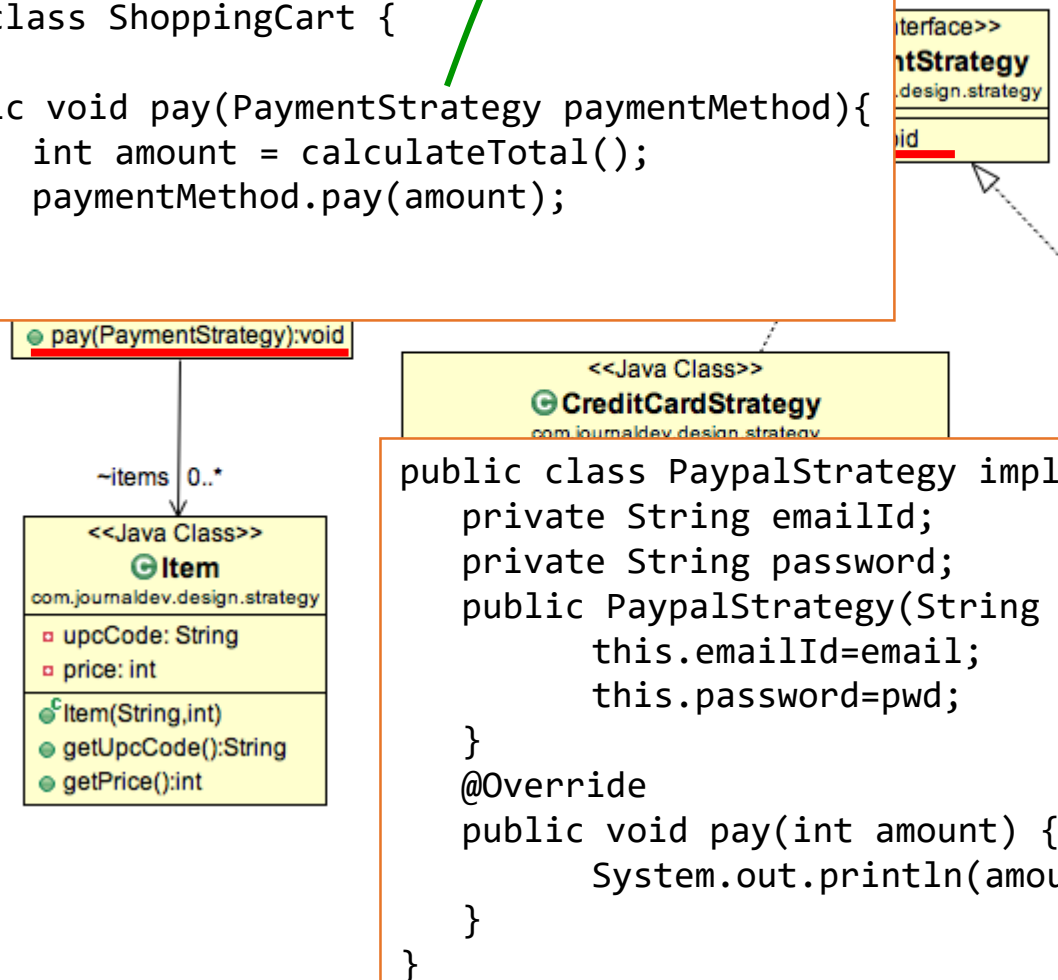



```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
               String cvv, String expiryDate){
         this.name=nm;
  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



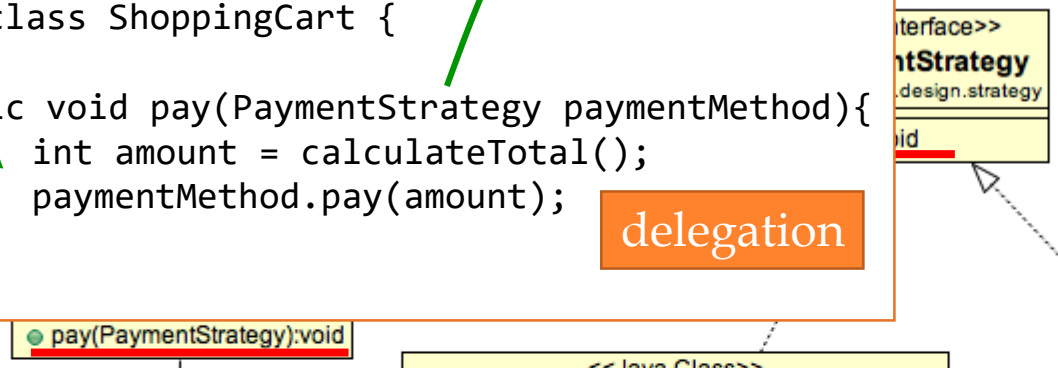

```
public class PaypalStrategy implements PaymentStrategy {
    private String emailId;
    private String password;
    public PaypalStrategy(String email, String pwd){
          this.emailId=email;
          this.password=pwd;
    }
    @Override
    public void pay(int amount) {
          System.out.println(amount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

步骤不变

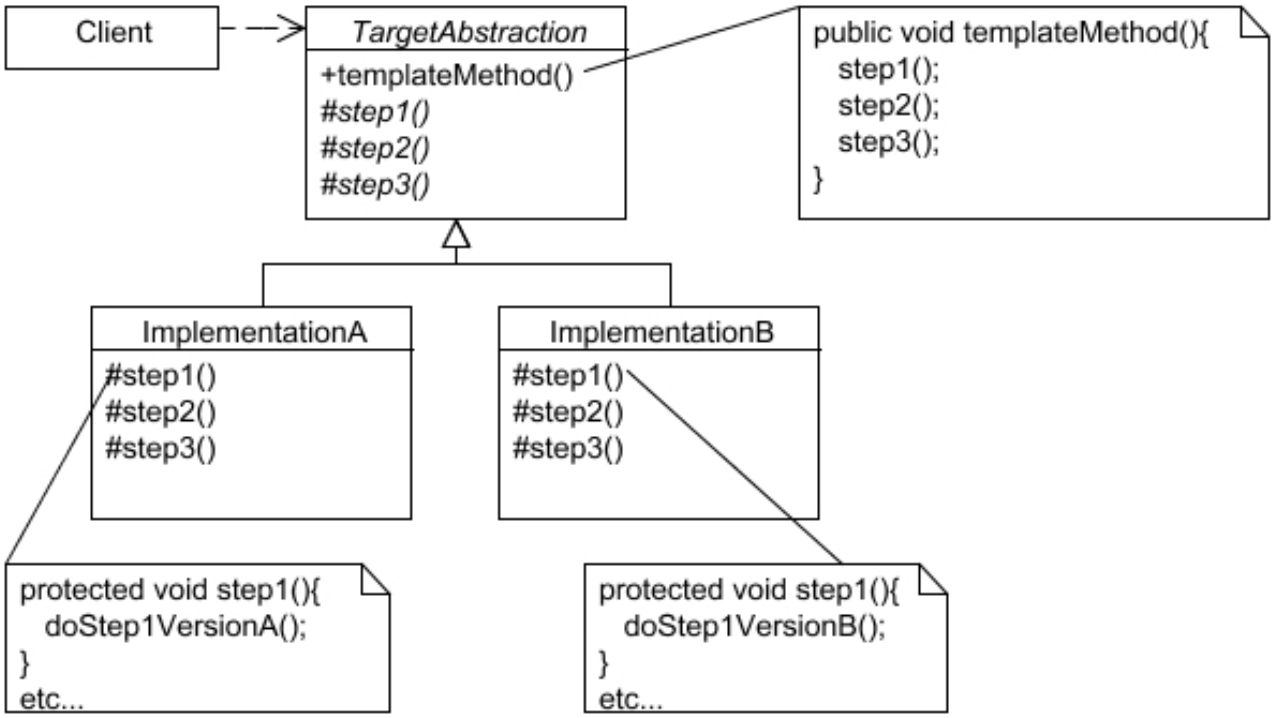
```
step1();
…
step2();
…
step3();
```

Template Method Pattern

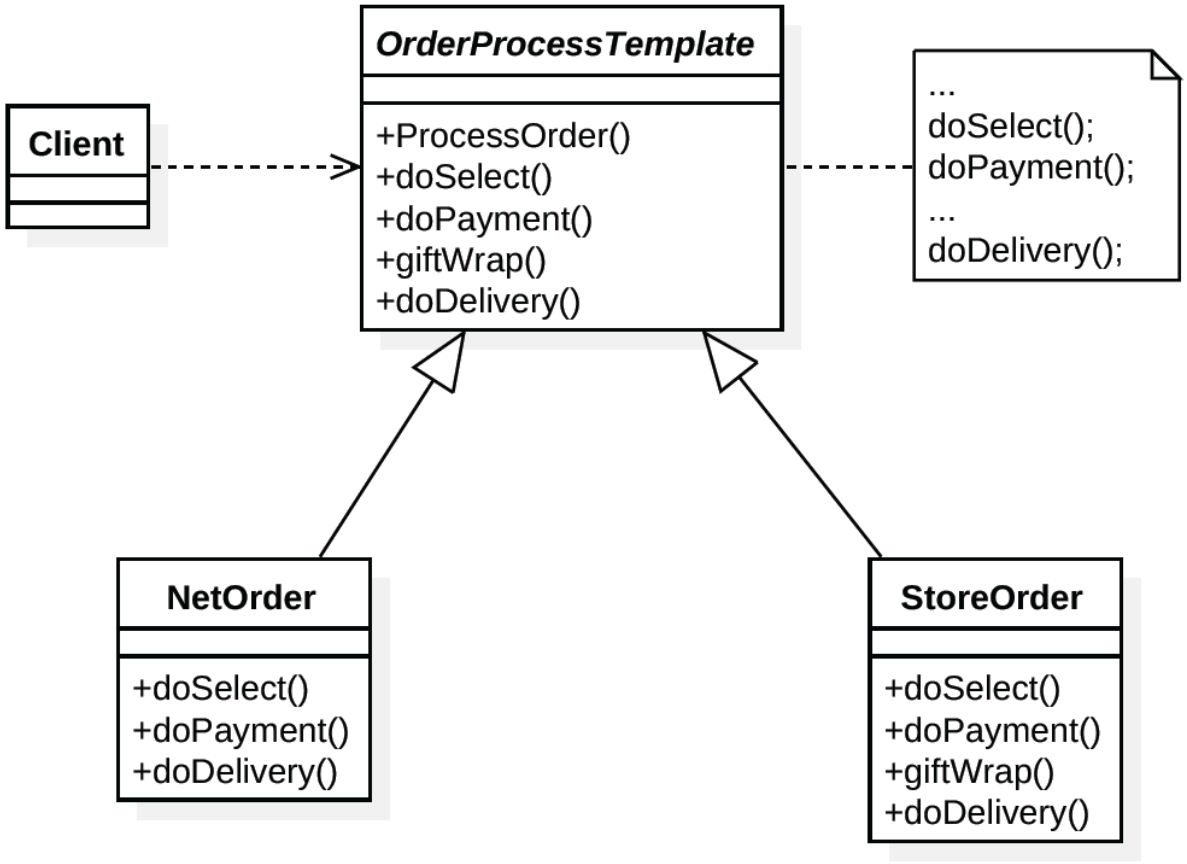
- **Template method** pattern uses inheritance + overridable methods **to vary part of an algorithm** 使用继承和重写实现模板模式
  - While strategy pattern uses delegation to vary the entire algorithm (interface and ad-hoc polymorphism).

- **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

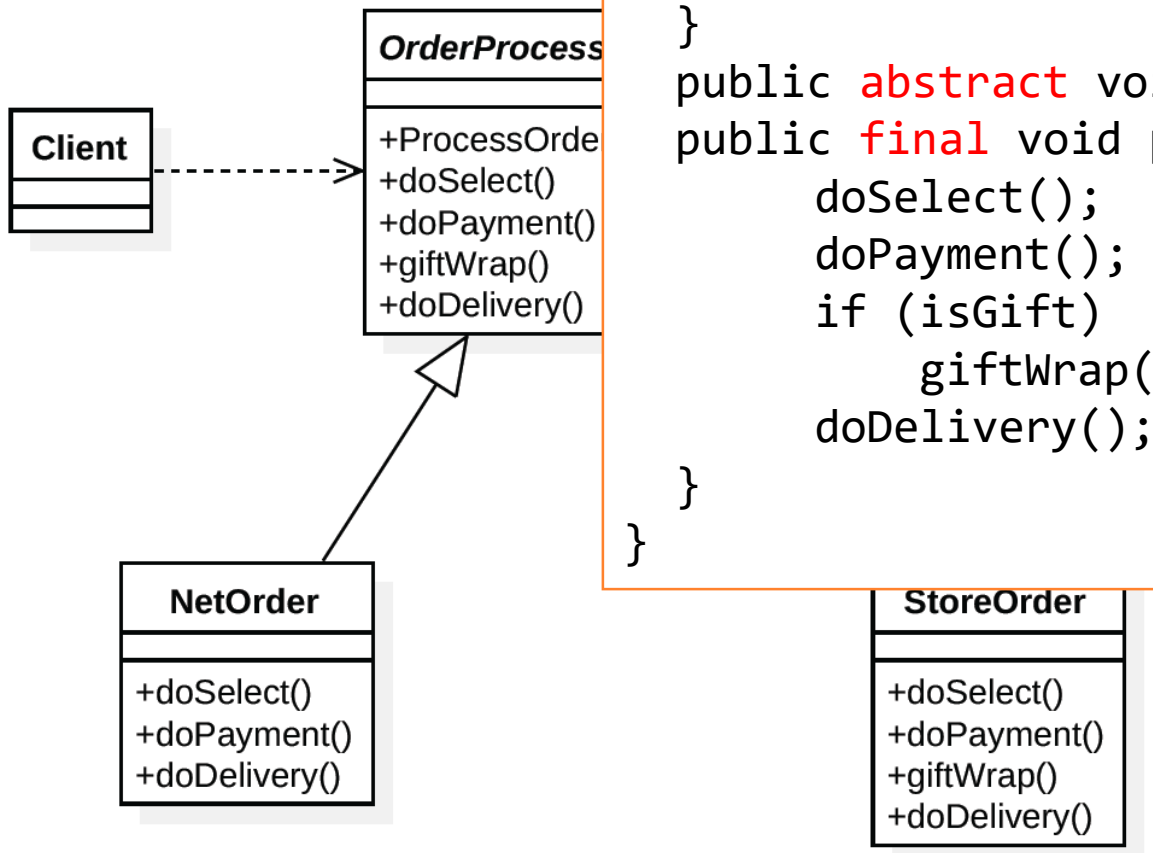


Example


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

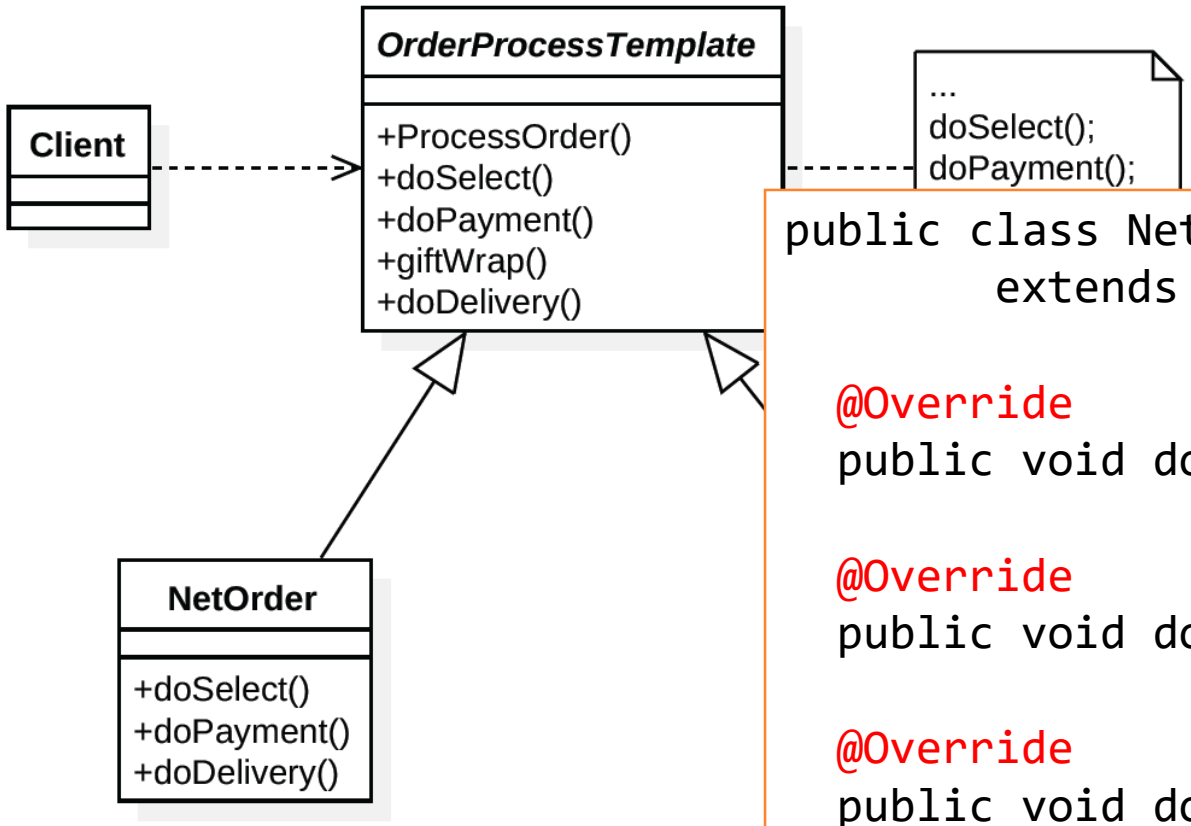
Example

```
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() { … }
}
```

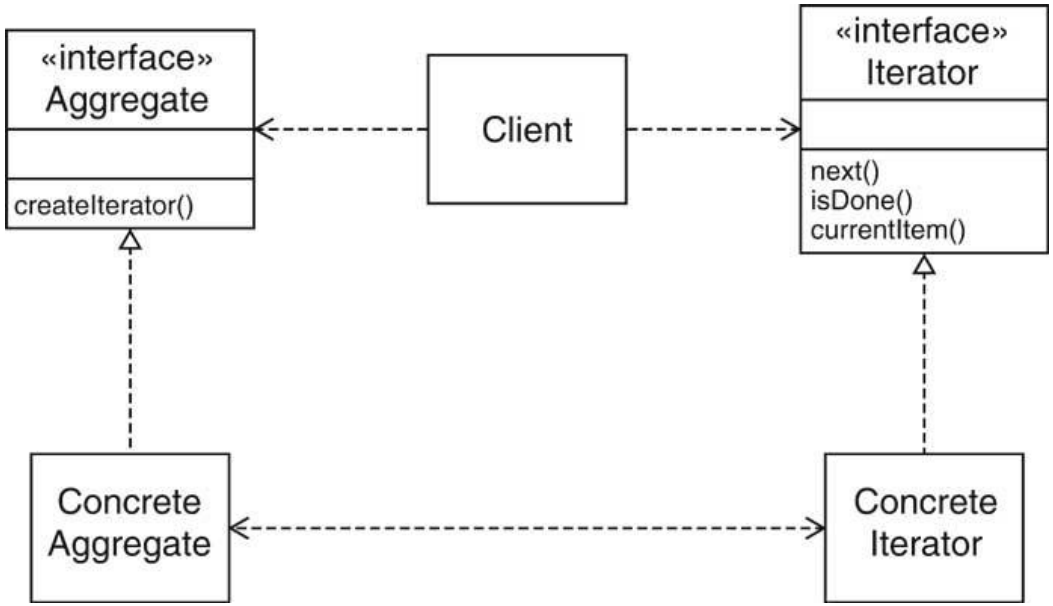
(3) Iterator

Iterator Pattern

- **Problem:** Clients need uniform strategy to access all elements in a container, independent of the container type
- **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



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) { … }
```

Summary

Summary

- **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 an existing method of an object.
  - Facade provides a simplified interface to a large body of code.
- **Behavioral patterns**
  - Strategy allows one of a family of algorithms to be selected on-the-fly 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.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 10, 2018