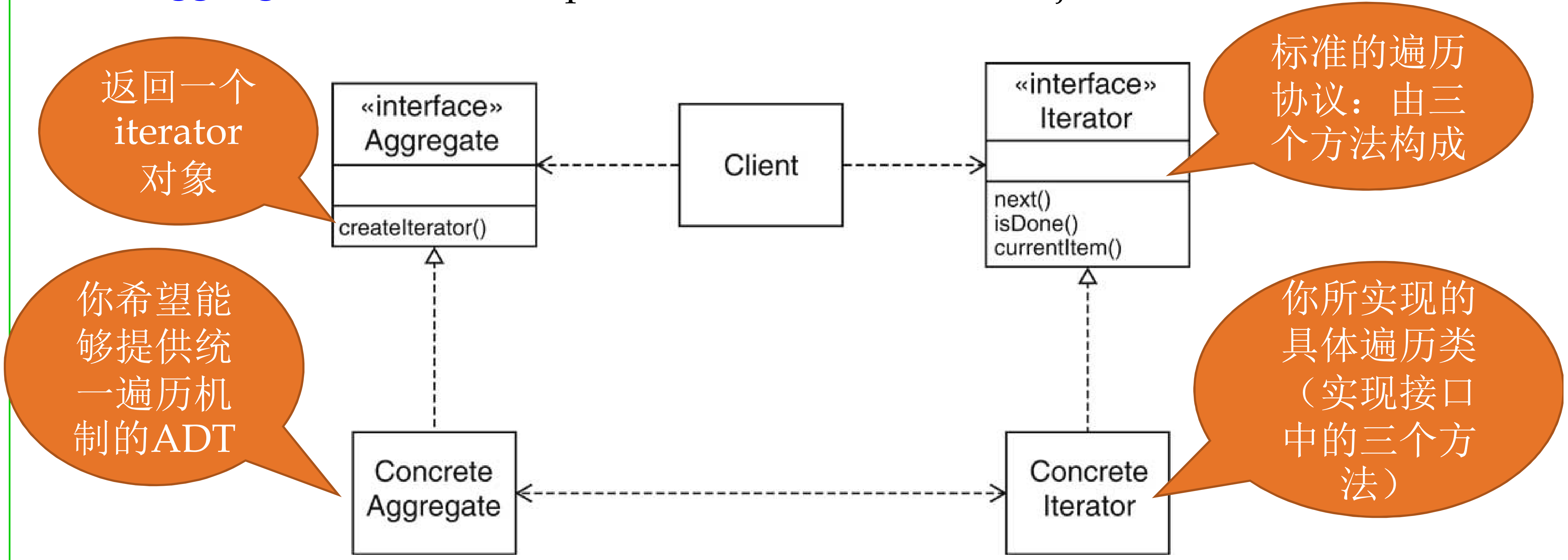
Iterator Pattern

- **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



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);
   …
}
```

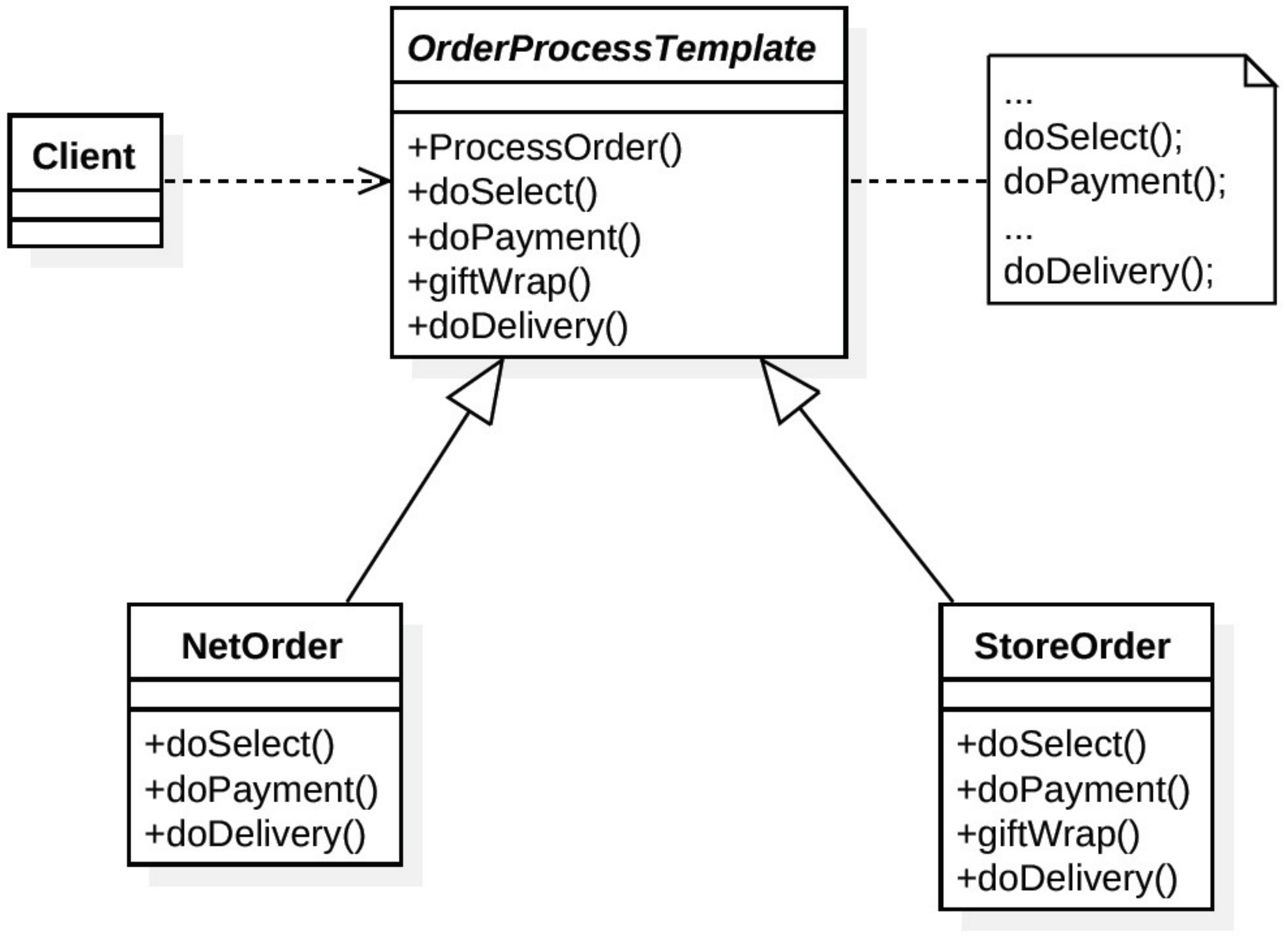
$\longleftarrow$ Defines an interface for creating an Iterator, but allows `Collection` implementation to decide which `Iterator` to create.

An example of Iterator pattern

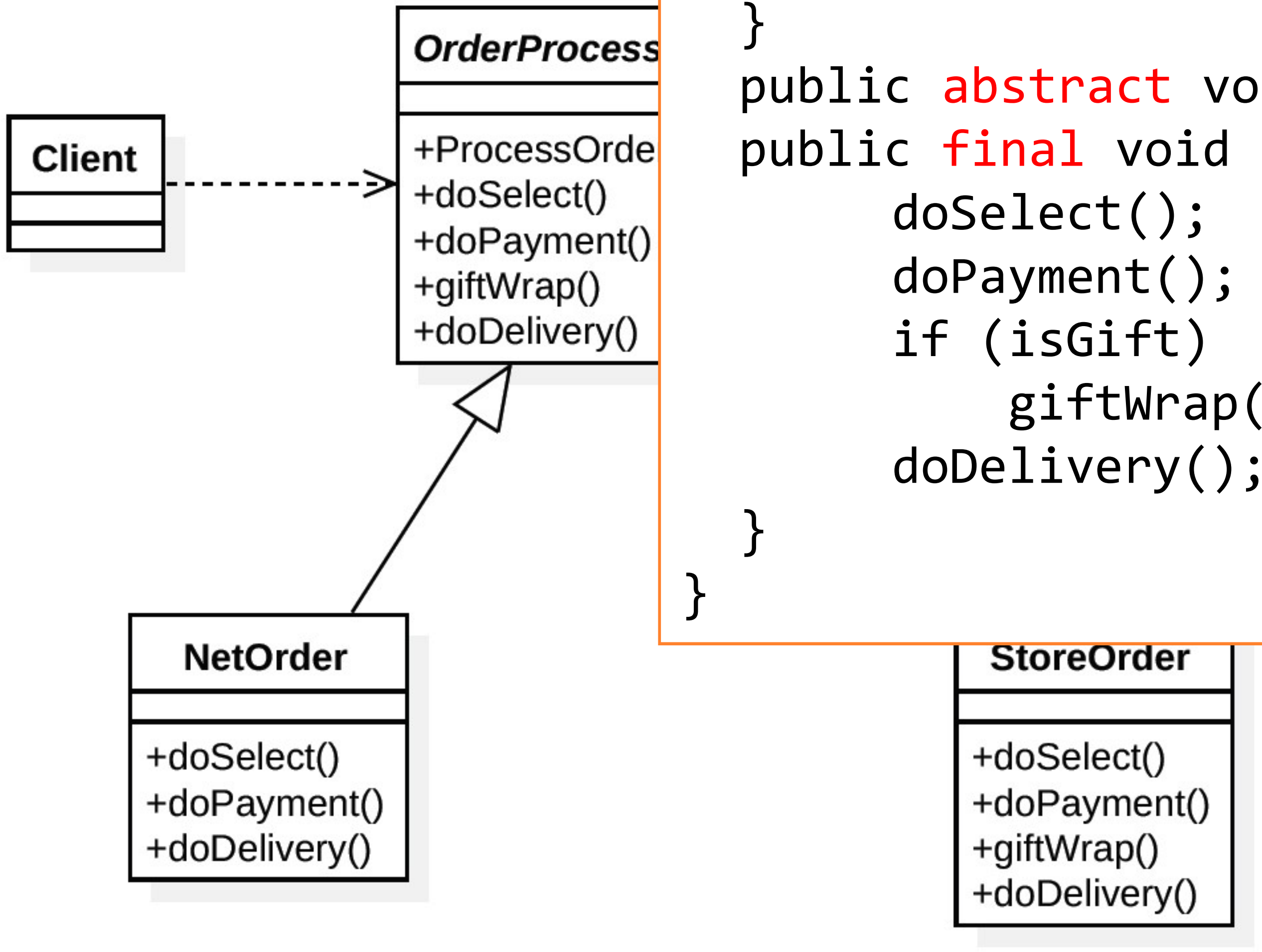
```
public class Pair<E> implements Iterable<E> {
   private final E first, second;
   public Pair(E f, E s) { first = f; second = s; }
   public Iterator<E> iterator() {
      return new PairIterator();
   }

   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) { … }
```

Example


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

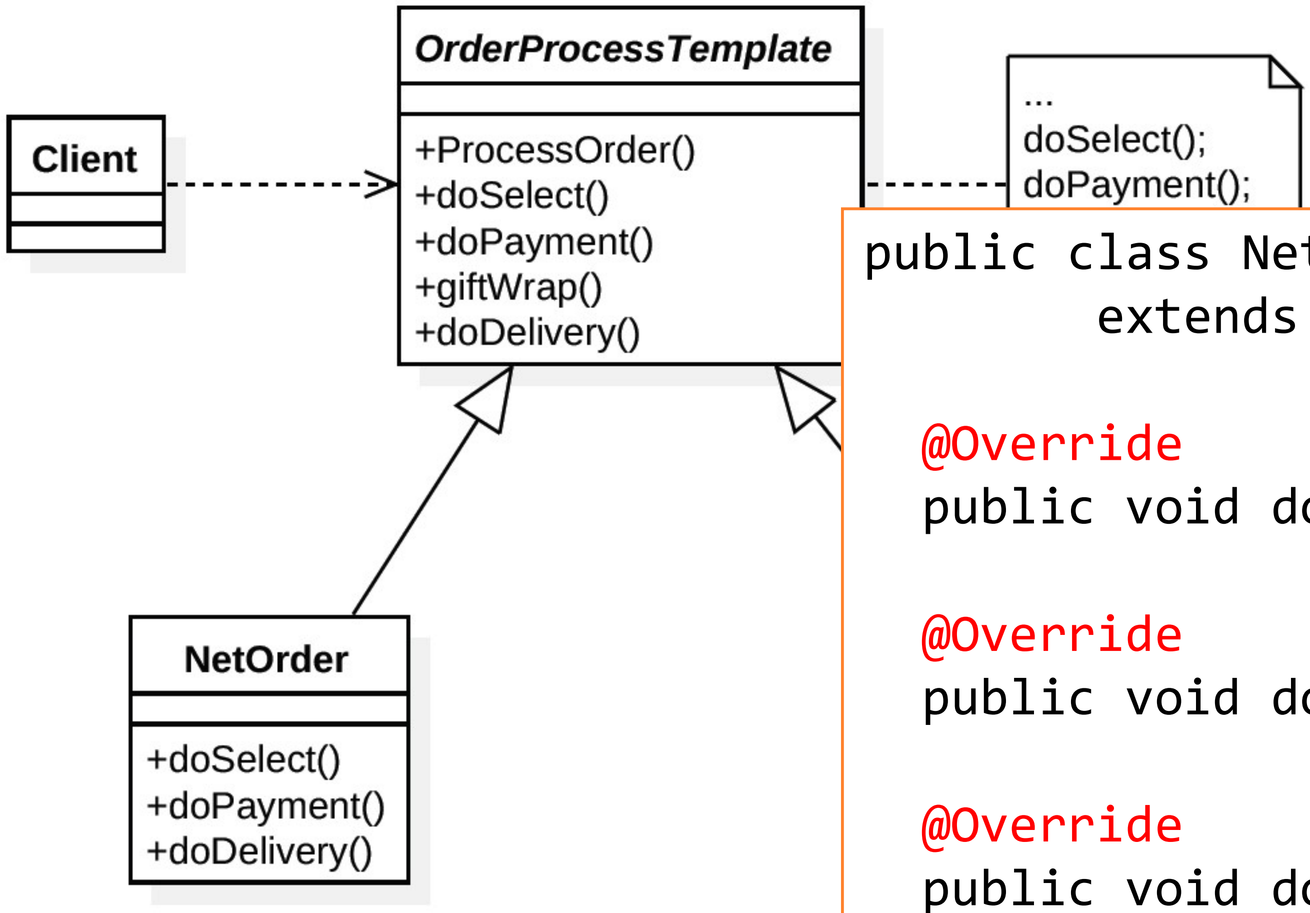
Example



```
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

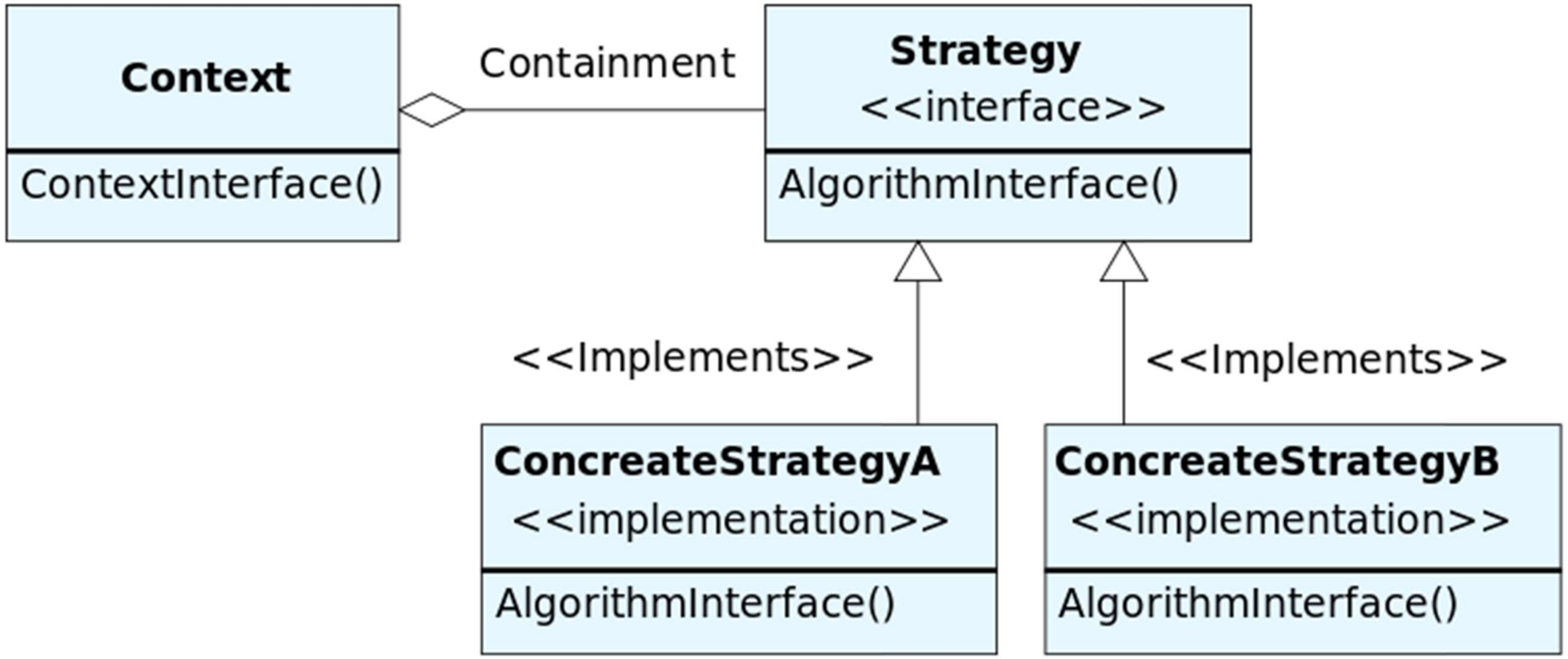
```
public class Calculator extends Application {
  protected String getApplicationTitle() { return "My Great Calculator"; }
  protected String getButtonText() { return "calculate"; }
  protected String getInititalText() { return "(10 – 3) * 6"; }
  protected void buttonClicked() {
    JOptionPane.showMessageDialog(this, "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

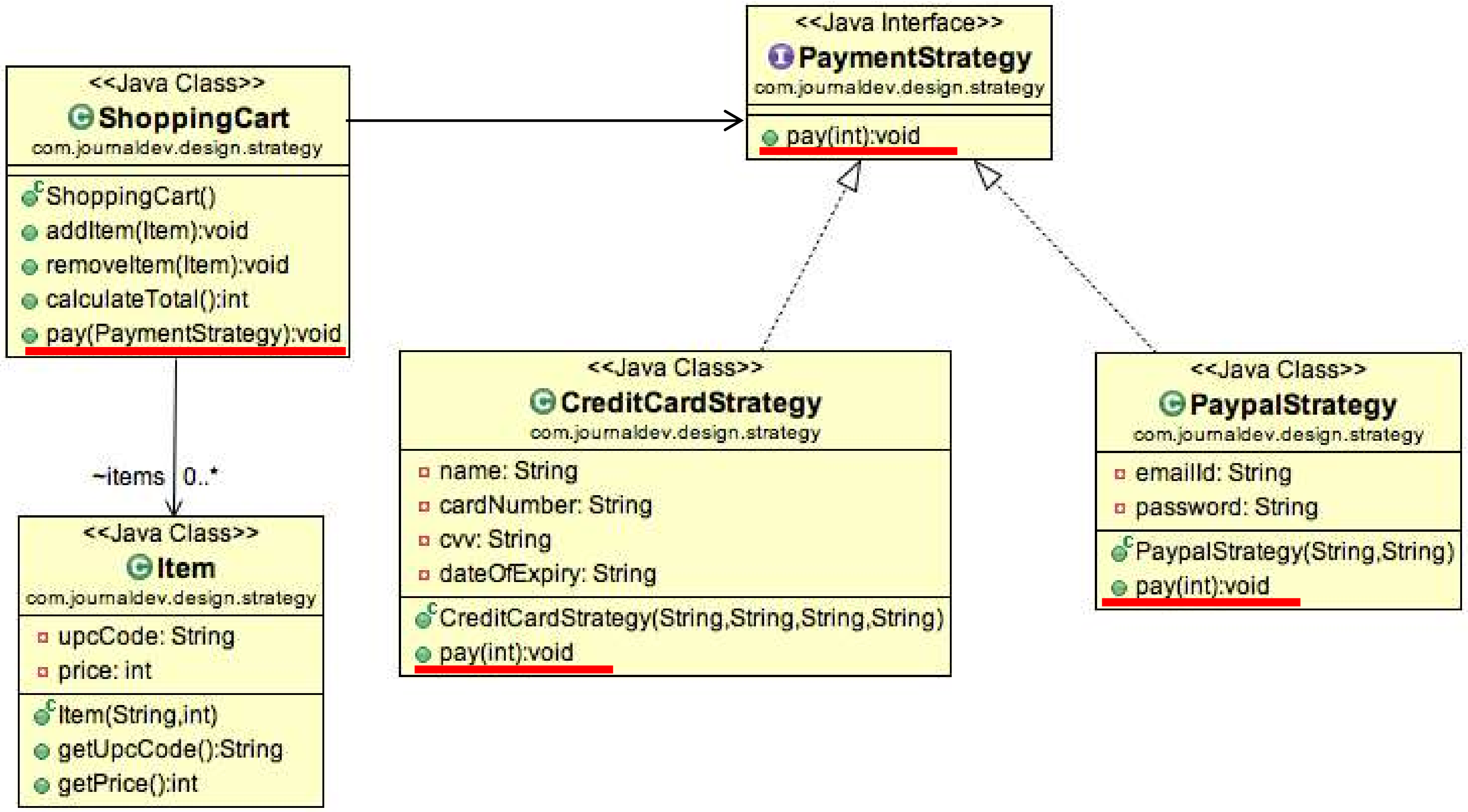
```
public class Ping extends Application {
  protected String getApplicationTitle() { return "Ping"; }
  protected String getButtonText() { return "ping"; }
  protected String getInititalText() { return "127.0.0.1"; }
  protected void buttonClicked() { ... }
}
```

Overriding

Strategy Pattern

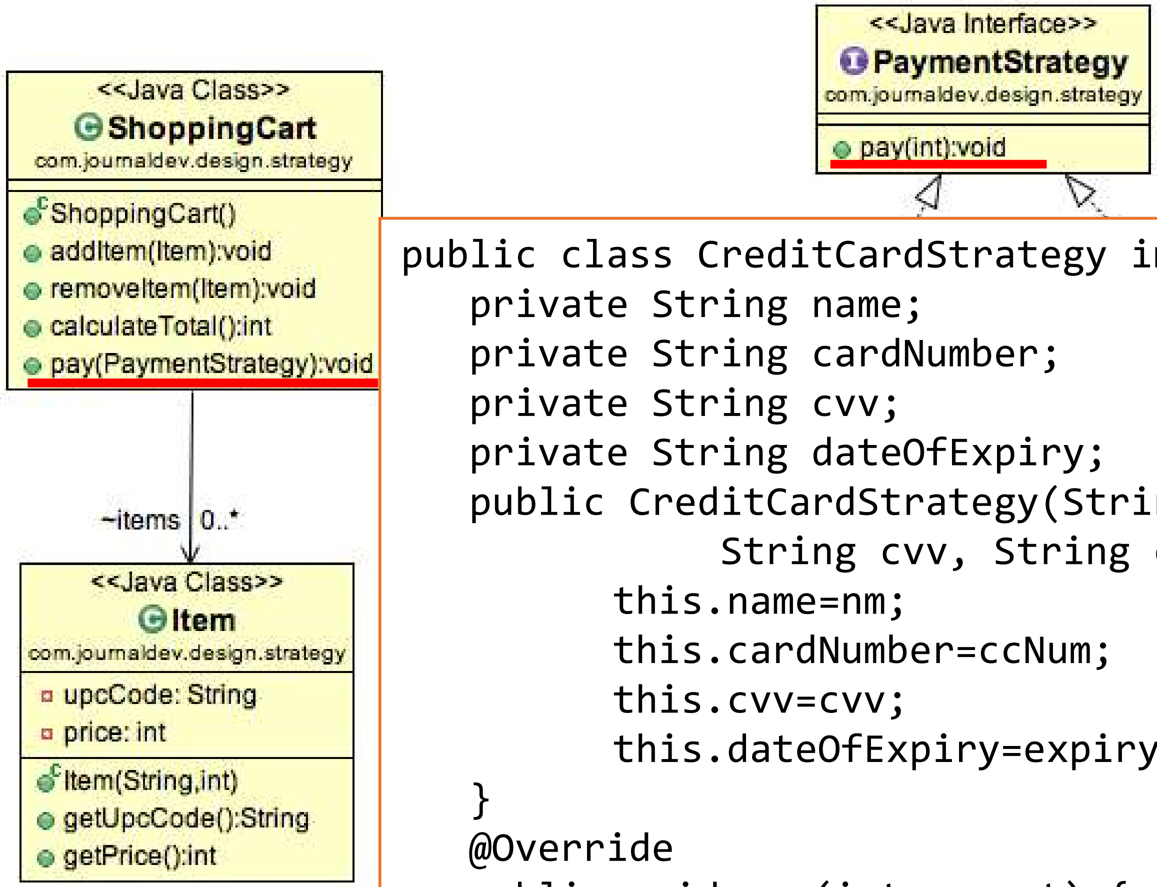


Code example



Code example

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

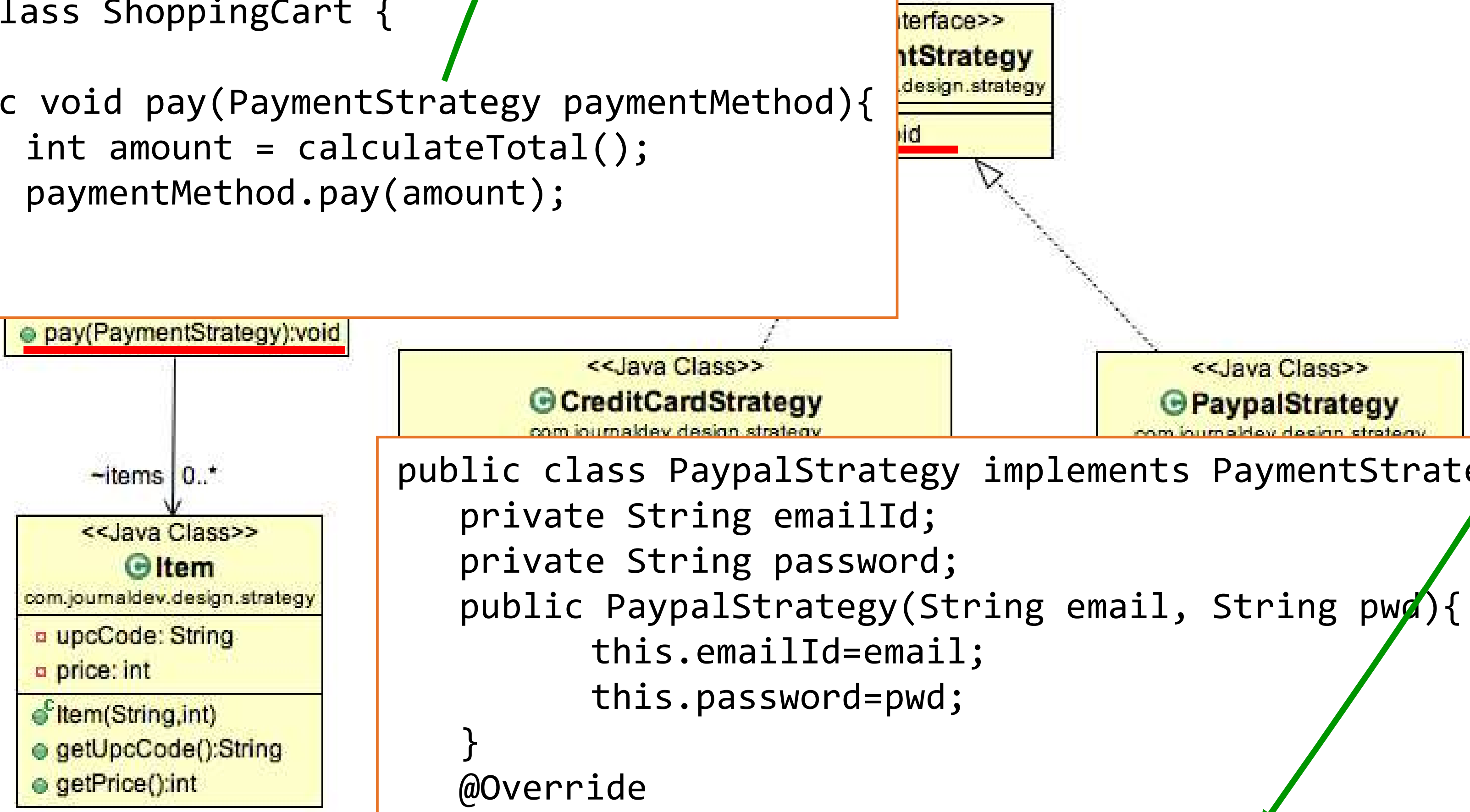


```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
              String cvv, String expiryDate){
         this.name=nm;
   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



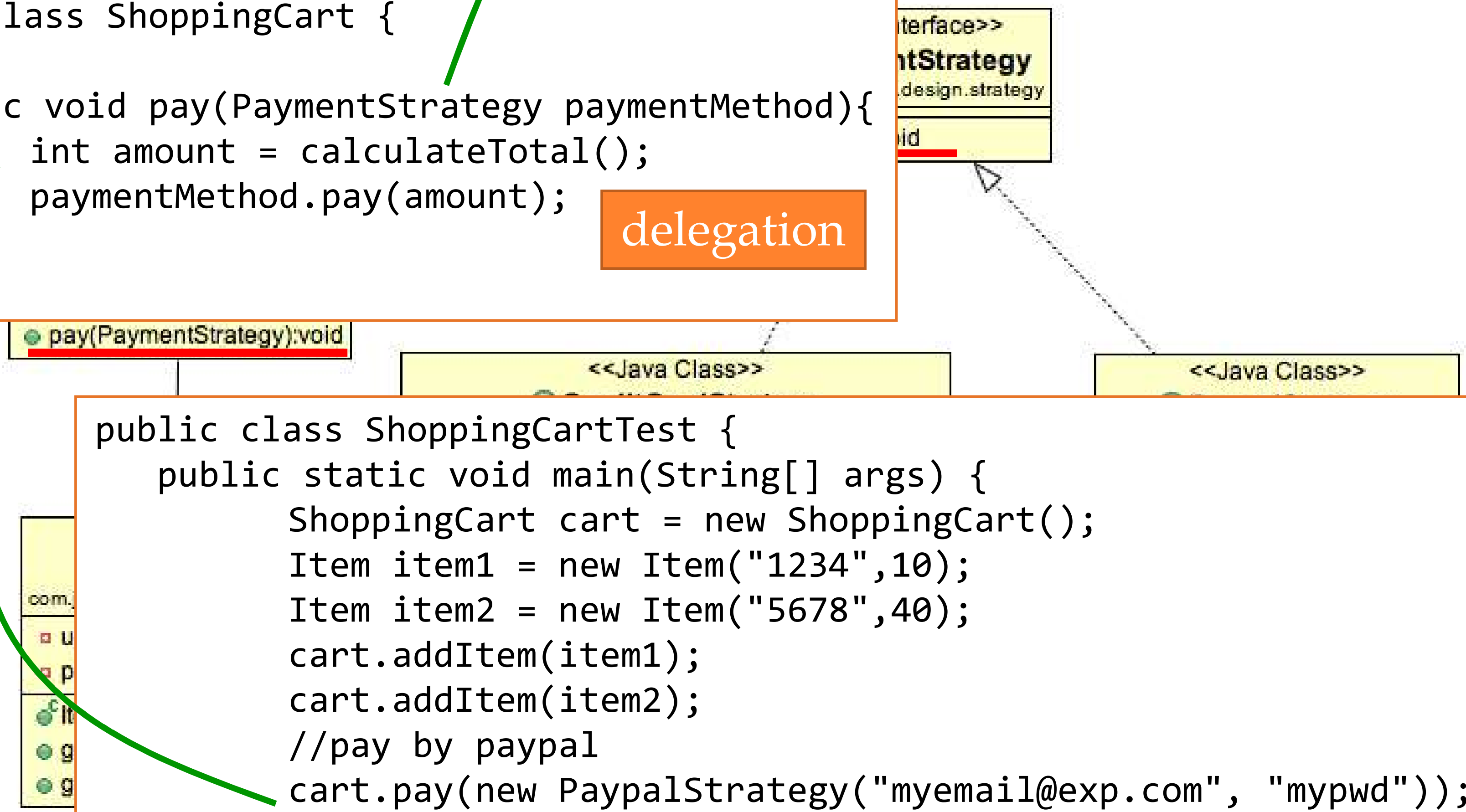
```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
         this.emailId=email;
         this.password=pwd;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount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

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

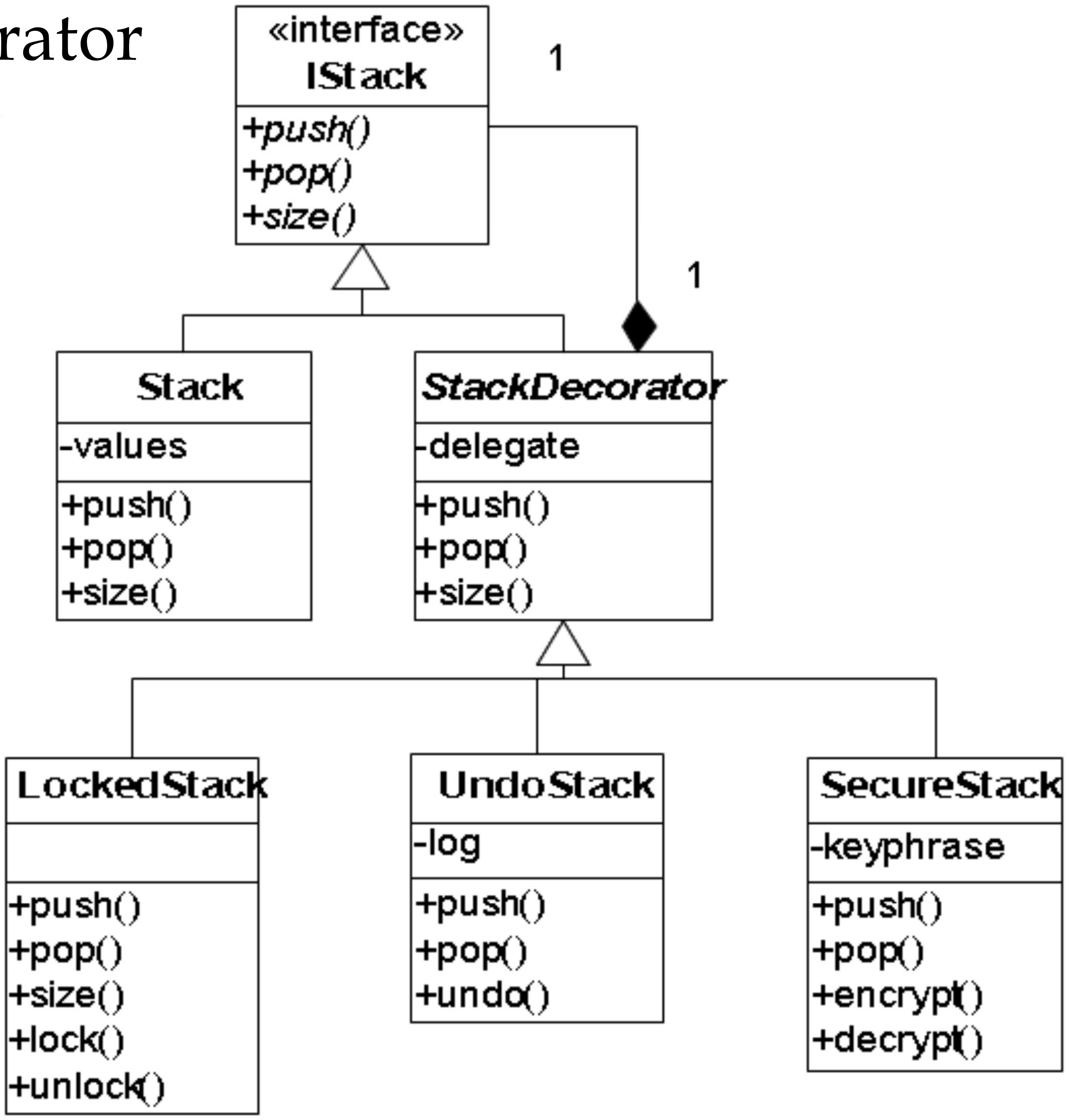
Without facade

```
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);
```

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…

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