Unit 3 Design Patterns

- 3.1 Singleton Pattern
- 3.2 Strategy Pattern
- 3.3 Factory Pattern
- 3.4 Observer Pattern
- 3.5 Adapter Pattern

Unit 3 Design Patterns

- **Book: Design Patterns: Elements of Reusable Object-Oriented Software (设计模式：可复用面向对象软件的基础)**
  - 设计模式提供了用面向对象的思想设计不同系统、不同应用时经常发生的问题的解决方案，向编程人员提供解决特定问题的可依据的方法蓝图。
  - 本书结合设计实例从面向对象的设计中精选出23个设计良好、表达清楚的设计模式，总结了面向对象设计中最有价值的经验，并且用简洁可复用的形式表达出来。

Unit 3 Design Patterns(cont.)

- A design pattern description consists of:
  1. A name that identifies the pattern
  2. A description of the problem that the pattern addresses
  3. A description of the solution that includes the class structures that solve the problem
  4. A discussion of the consequences of using the pattern

Unit 3 Design Patterns

Creational Patterns:	• Abstract Factory • Builder • Factory Method • Prototype • Singleton
Structural Patterns:	• Adapter • Bridge • Composite • Decorator • Façade • Flyweight • Proxy
Behavioral Patterns:	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor • Interpreter • Template Method

Design patterns document good design solutions that can be used for similar problems and form a shared vocabulary for problem-solving discussions. In addition, design patterns are good illustrations of object-oriented concepts.

Unit 3 Design Patterns(cont.)

- 面向对象设计原则：
  - ① 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。
    - 把变化的部分“封装”起来，好让其他部分不会受到影响；代码变化引起的不经意后果变少，系统变得更有弹性。
  - ② 针对接口编程，针对超类型编程

Unit 3 Design Patterns(cont.)

- 面向对象设计原则：
  - ③ 多用组合，少用继承
  - ④ 类应该对修改关闭，对扩展开放，允许系统在不修改代码的情况下，进行功能扩展

Unit 3 Design Patterns

- ✓ 3.1 Singleton Pattern
- ☐ 3.2 Strategy Pattern
- ☐ 3.3 Factory Pattern
- ☐ 3.4 Observer Pattern
- ☐ 3.5 Adapter Pattern

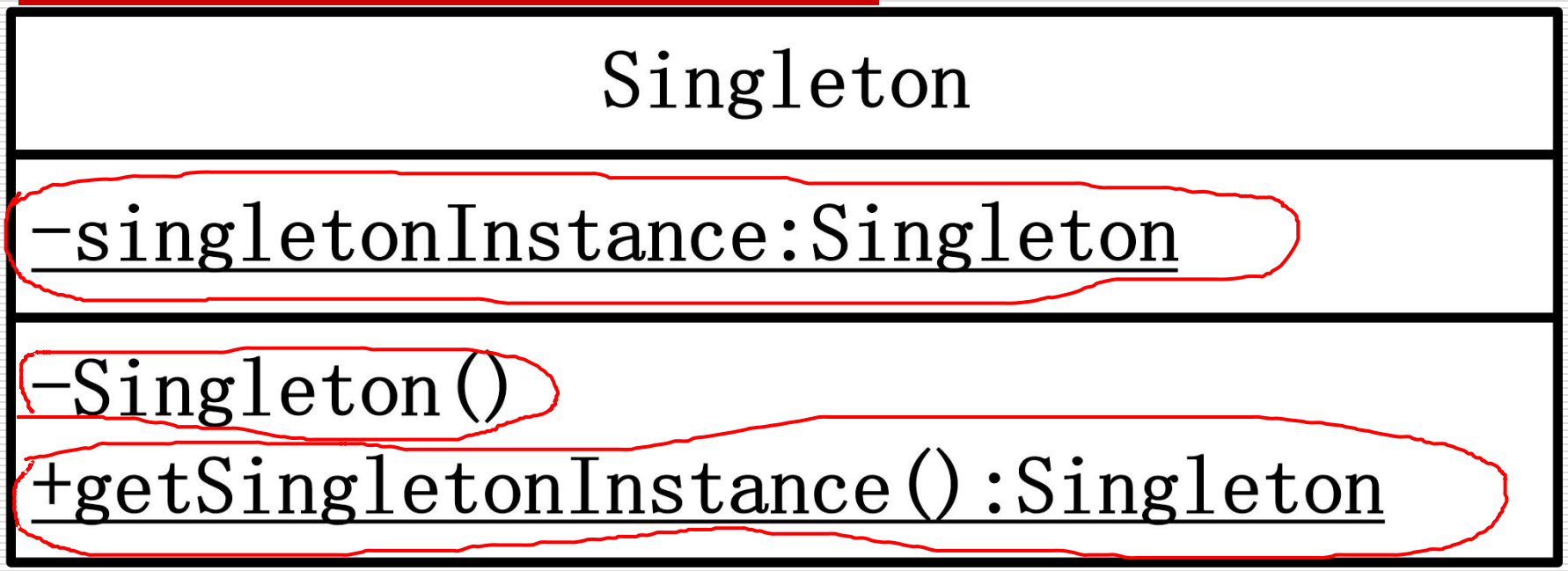
3.1 Singleton Pattern

- 很多时候都需要保证一个类仅有一个实例：
  - 比如希望整个应用程序中只有一个连接数据库的Connection实例；
  - 操作系统只需要一个时钟；
  - 对于建模固定不变信息的类，仅需访问一个实例
- 怎样才能保证一个类只有一个易于被使用的实例呢？

1. Description

- The *singleton pattern* ensures that only one instance of a class is created and provides a method to access that one instance.

2. Structure




```
public  static Singleton  getSingletonInstance() {
        return new Singleton();
}
```

2. Structure(cont.)

- **In this pattern:**
  1) **The static attribute instance contains the single instance of the class.**
  2) **The constructor is defined as private so that other classes cannot create instances.**
  3) **The static method getSingletonInstance() returns the single instance of the class. (The first time this method is called, it creates the single instance.)**
- 通过1）2)、3)三点就可以完全控制类的创建，无论有多少地方需要用到这个类，它们访问的都是类唯一生成的那个实例。

3. Example

- Class ICarnegieInfo contains the contact information for *iCarnegie*. Only one instance of class ICarnegieInfo can be created.
- ICarnegieInfo.java
- ICarnegieInfoDemo.java

4. Consequences

- The singleton pattern has the following benefits:
  - A singleton class can control how client code accesses the single instance.
  - A singleton class can be easily modified if requirements change and the application needs to limit the number of instances to a number other than one.
- ICarnegieInfo.java

3.1Singleton Pattern

- 总结：什么情况下使用单一实例模式(Singleton Pattern)？
  - 需求描述：希望整个应用程序中只有一个连接数据库的Connection实例；
  - 一个类有属性信息，但是其所有的属性信息的值是唯一的，而且只能访问，不能修改，可以考虑用单一实例模式。
  - 一个类没有属性信息，只有方法，可以考虑用单一实例模式。

Unit 3 Design Patterns

- ☐ 3.1 Singleton Pattern
- ✓ 3.2 Strategy Pattern
- ☐ 3.3 Factory Pattern
- ☐ 3.4 Observer Pattern
- ☐ 3.5 Adapter Pattern

3.5 Strategy Pattern

- Consider the Milk system that display the information in the order in one of three ways: plain text, HTML, and XML:
- Solution: MilkSystemSolution.java

3.2 Strategy Pattern(cont.)

- **Above solution has some drawbacks:**
  1) **If the rules to display the information are complex, the method formatOrders will be long and complex.**
  2) **增加新的算法或改变现有算法将十分困难。**

1. Introduction(cont.)

- A solution that uses the strategy pattern doesn't have these drawbacks.
- In a strategy-pattern solution,
  - The code to display the information in plain text is encapsulated in its own class, as is the code in HTML, and XML.
  - Each class implements its own version of the `formatOrders` method.

1. Introduction(cont.)

- The complexity of displaying the information code is reduced by separating it into three classes.
- strategy-pattern-formatOrders-part.png

1. Introduction(cont.)

- 牛奶系统要求： **display the information in the borrower database in one of three ways: plain text, HTML, and XML.**
- 总结： **A program which requires a particular service or function and which has several ways of carrying out that function.**
- 所以可以提供一个声明方法`formatOrders()`的接口**OrdersFormat**。

1. Introduction(cont.)

- strategy-pattern-formatOrders.png
  - 所写的程序代码的大部分都只操作接口OrdersFormat类型的变量。
  - 可以轻易扩增新类(大部分程序代码都不会被影响)，使设计便于阅读和维护。
    - **If we display the information in a new way, a new class that encapsulates the information rules is created.**
      - **The new class implements the interface OrdersFormat and defines its own version of the method `formatBorrowers`.**

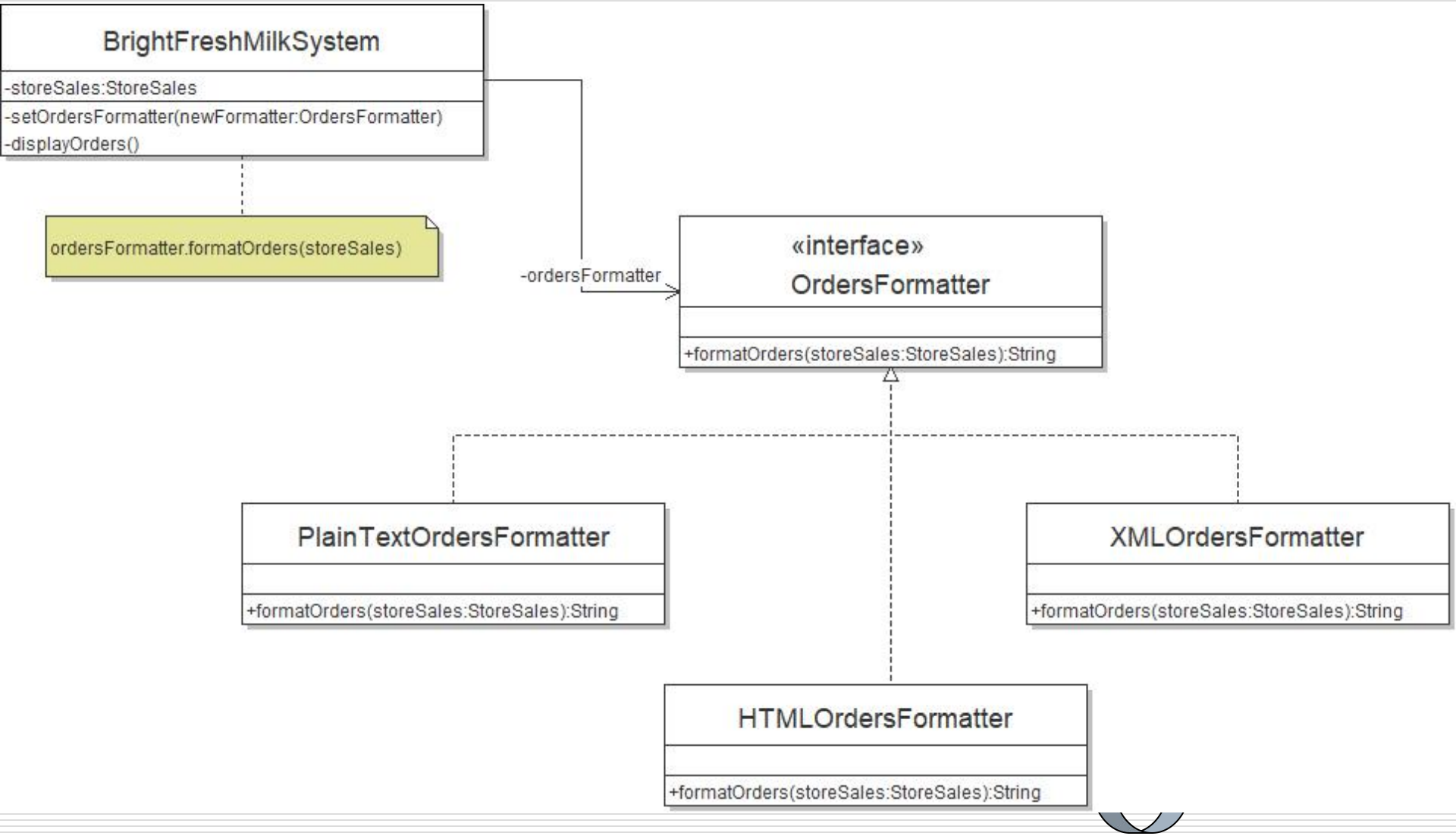
1. Introduction(cont.)

- Interface OrdersFormatter
- Class PlainTextOrdersFormatter
- Class HTMLOrdersFormatter
- Class XMLOrdersFormatter
- Class MilkSystem

1. Introduction(cont.)

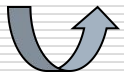
- The idea behind Strategy is to encapsulate the various strategies in a single module and provide a simple interface to allow choice between these strategies.
- 整个Strategy的核心部分就是接口的使用，使用Strategy模式可以在用户需要变化时，修改量很少，而且快速。

3. UML Class Diagram



3. UML Class Diagram(cont.)

- The class Context maintains a reference, called strategy, to an object of type Strategy.
  - The class Context contains the method setStrategy, which allows client code to specify the desired algorithm.
  - The class Context contains the method invokeStrategy, which the client code calls when it needs to execute an algorithm.

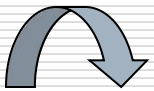


4. Example

- strategy-pattern-example.png
- Interface OrdersFormatter
- Class PlainTextOrdersFormatter
- Class HTMLOrdersFormatter
- Class XMLOrdersFormatter
- Class BrightFreshMilkSystem

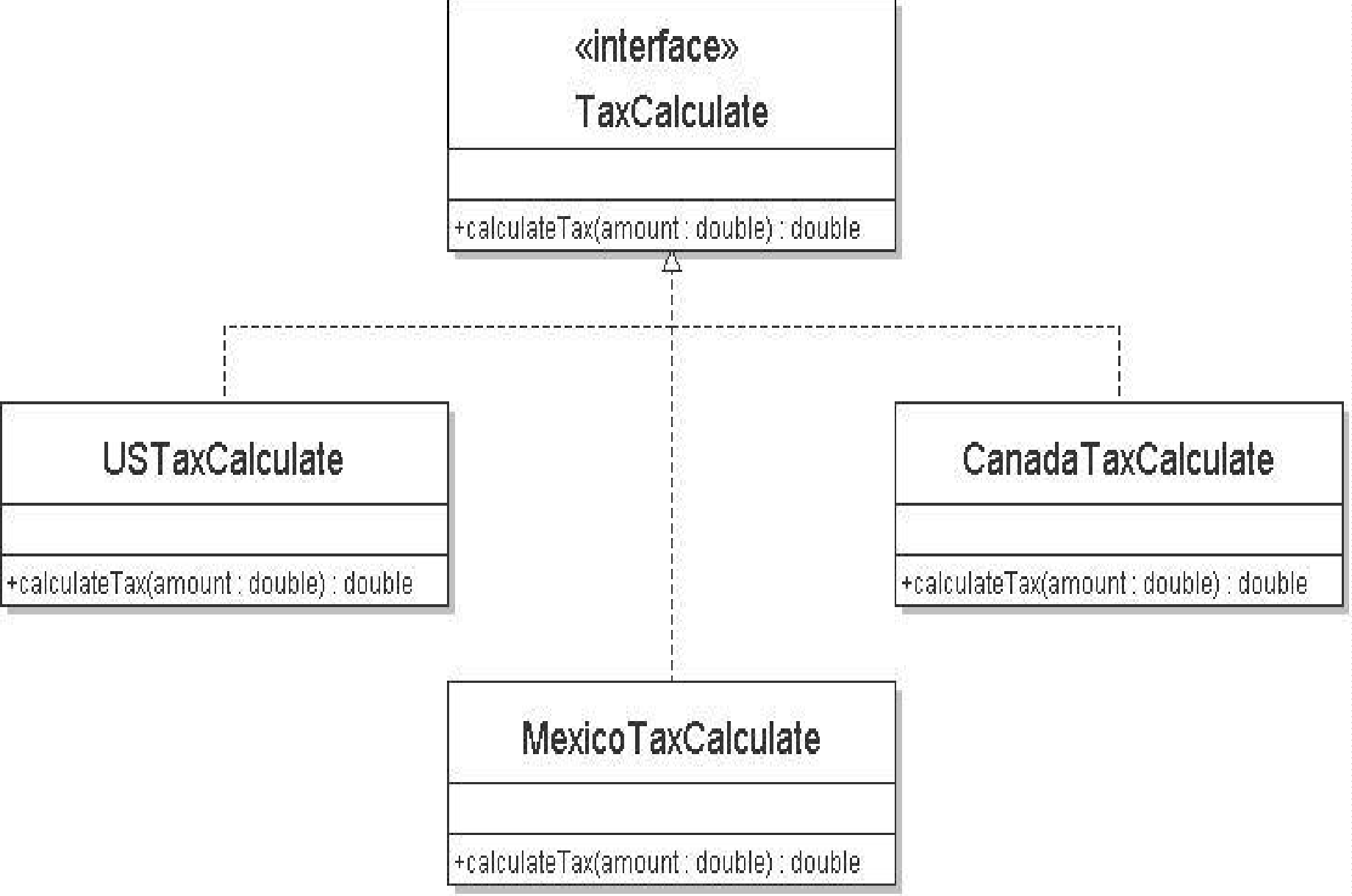
2. Description

- There are a number of cases in programs where we'd like to do the same thing in several different ways.



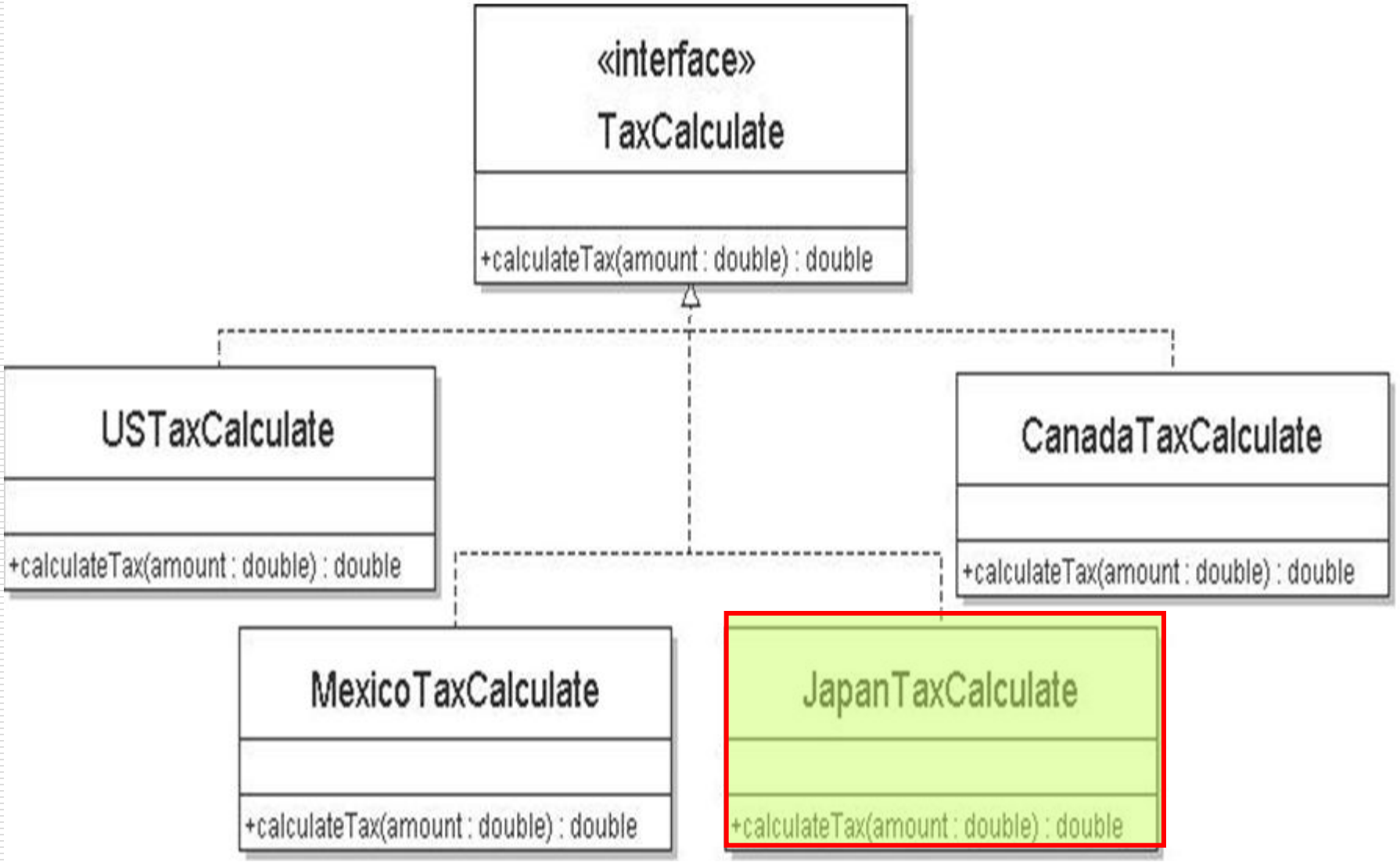
税费计算

- 某一公司在美国、加拿大和墨西哥都有分公司，都销售相同的产品，但在这三个国家的消费税的计算方法是不一样的；
- 但需要设计一个灵活的税费计算方法，以便可以适应当该公司的业务扩展到其它国家，比方说南美、日本等，消费税的计算可以方便加入到该系统中。


«interface»
TaxCalculate
+calculateTax(amount : double) : double
USTaxCalculate
+calculateTax(amount : double) : double
CanadaTaxCalculate
+calculateTax(amount : double) : double
MexicoTaxCalculate
+calculateTax(amount : double) : double

```
public class TestTaxCaculate {

    public static void main(String[] args){

        TaxCalculate taxCalculate;

        taxCalculate = new USTaxCalculate();
        taxCalculate.calculateTax(1000000);

        taxCalculate = new MexicoTaxCalculate();
        taxCalculate.calculateTax(1000000);

        taxculate = new CanadaTaxCalculate();
        taxCalculate.calculateTax(10000000);
    }
}
```

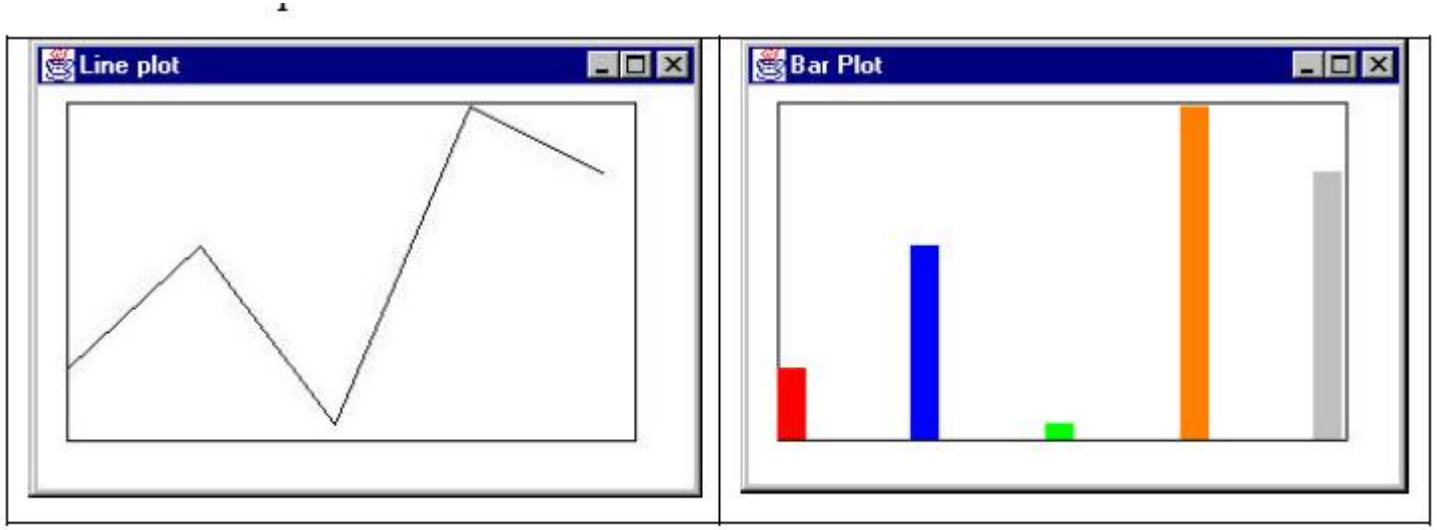
税费计算图


«interface»
TaxCalculate
+calculateTax(amount : double) : double
USTaxCalculate
+calculateTax(amount : double) : double
CanadaTaxCalculate
+calculateTax(amount : double) : double
MexicoTaxCalculate
+calculateTax(amount : double) : double
JapanTaxCalculate
+calculateTax(amount : double) : double

总结：什么情况下使用策略模式(Strategy Pattern)？

- Save files in different formats.
- Compress files using different algorithms
- Capture video data using different compression schemes
- Use different line-breaking strategies to display text data.
- Plot the same data in different formats: line graph, bar chart or pie chart.

3.2 Strategy Pattern



Unit 3 Design Patterns

- ☐ 3.1 Singleton Pattern
- ☐ 3.2 Strategy Pattern
- ✓ 3.3 Factory Pattern
- ☐ 3.4 Observer Pattern
- ☐ 3.5 Adapter Pattern

3.3 Factory Pattern

- 举例：
  - 手机生产商： Manufacturer.java
  - 生产各种类型的手机：随着时间的过去，生产的品种发生变化。
  - 将类Manufacturer中改变的东西进行封装

3.3 Factory Pattern（续）

```
public class Manufacturer {
    Mobile orderMobile(String type){
        Mobile mobile;
        mobile.assembleMainboard();
        mobile.assembleSUBboard();
        mobile.assembleKeypad();
        mobile.assembleWholewidget();
        mobile.BurnSD();
        mobile.testing();
    }
}
```

把创建对象的代码抽离

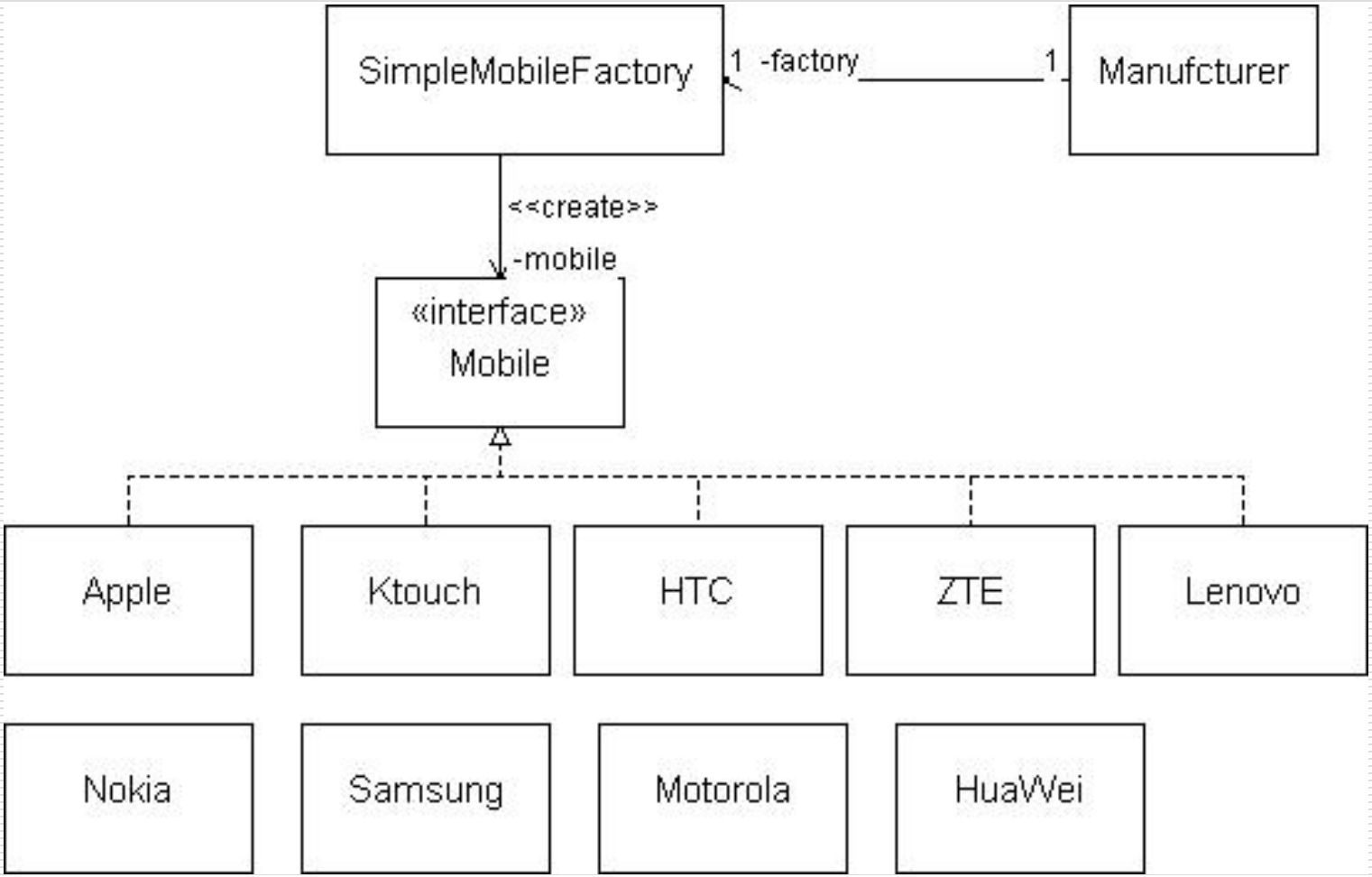
3.3 Factory Pattern（续）

将变化的代码封装在一个新类中，该类仅为客户创建各种手机对象：

SimpleMobileFactory.java

```
if(type.equals("apple")){
    mobile = new Apple();
}else if (type.equals("SAMSUNG")){
    mobile = new Samsung();
}else if (type.equals("HTC")){
    mobile = new HTC();
}else if (type.equals("Motorola")){
    mobile = new Motorola();
}else if (type.equals("NOKIA")){
    mobile = new Nokia();
}else if (type.equals("ZTE")){
    mobile = new ZTE();
}else if (type.equals("HuaWei")){
    mobile = new HuaWei();
}else if (type.equals("K-touch")){
    mobile = new Ktouch();
}else if (type.equals("Lenovo")){
    mobile = new Lenovo();
}
```

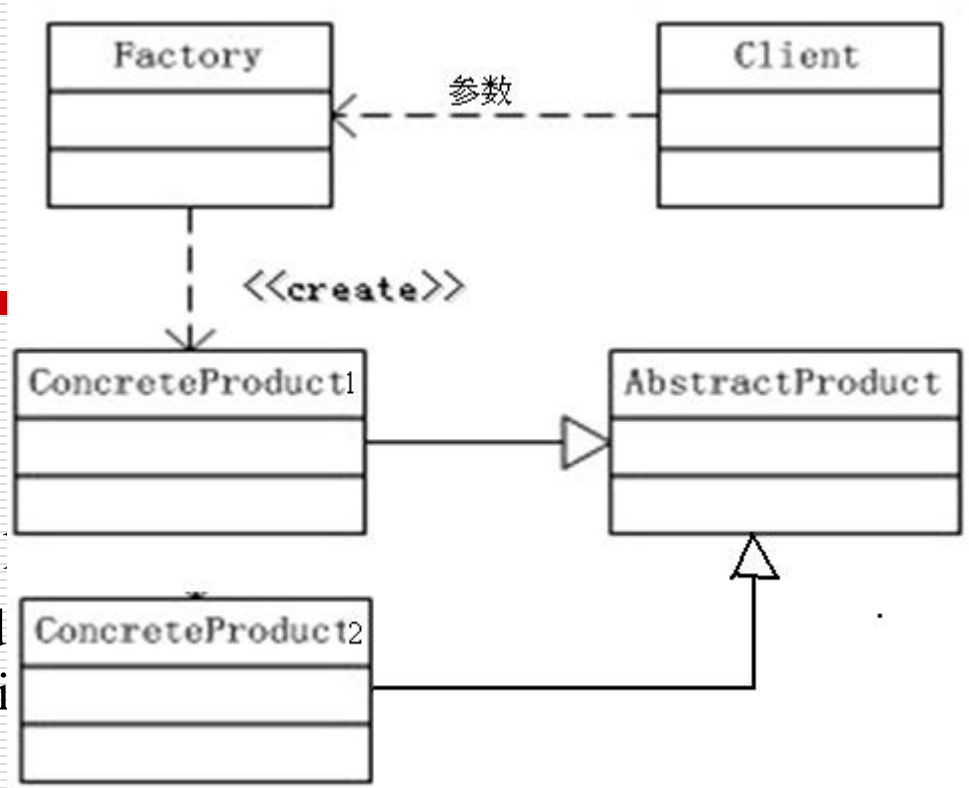
3.3 Factory Pattern（续）



3.3 Factory Pattern(续)

- A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently.
- 1）简单工厂
- 2）工厂模式

1）简单工厂



AbstractProduct is a base c

The Factory is a class that d
subclasses to return dependi

有关手机举例：

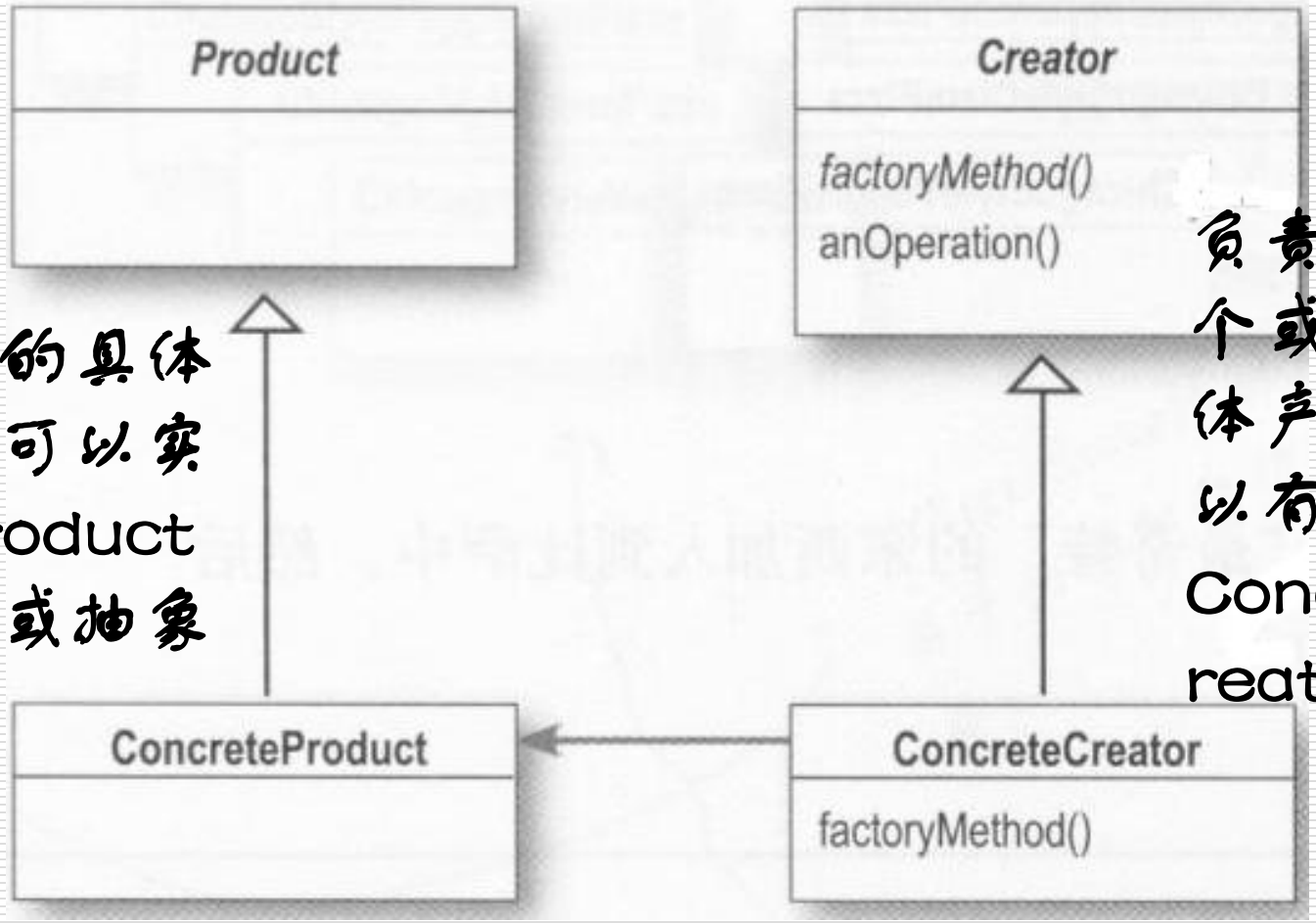
1）简单工厂（续）

- 简单工厂：
  - 工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；
  - 把改变的内容进行封装，使他们不会干扰应用的其他部分；
  - 该工厂类可能会被除了Manufacturer类的其他类使用；

2）工厂模式

- 现实中不同品牌的手机可能由不同的手机厂家生产。
- 手机案例

2）工厂模式（续）


Product
Creator
factoryMethod()
anOperation()
不同的具体产品可以实现Product接口或抽象类
负责创建一个或多个具体产品，可以有很多ConcreteCreator
ConcreteProduct
ConcreteCreator
factoryMethod()

2）工厂模式（续）

- 使用继承机制，把对象的创建委托给子类，子类实现工厂方法来创建对象；
- 与简单工厂相比，实现了可扩展，应用于产品结构更复杂的场合，有一个工厂类负责具体类的实例化，变成由一群子类来负责实例化，易于扩展加盟店。

Unit 3 Design Patterns

- ☐ 3.1 Singleton Pattern
- ☐ 3.2 Strategy Pattern
- ☐ 3.3 Factory Pattern
- ✓ 3.4 Observer Pattern
- ☐ 3.5 Adapter Pattern

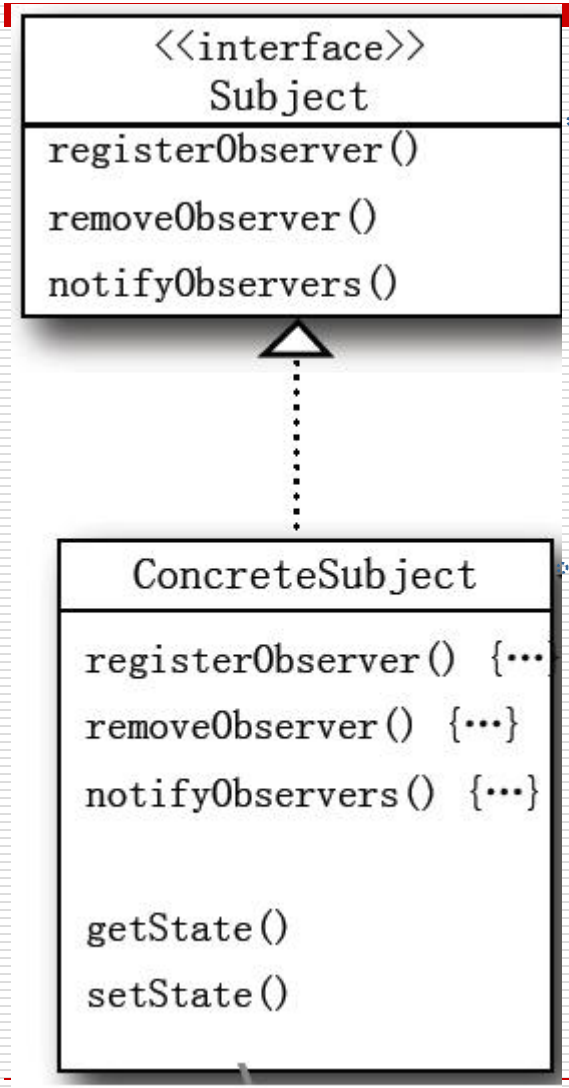
3.4 Observer Pattern

- 观察者模式定义了**主题**对象和**观察者**对象之间的一对多依赖，当主题对象的状态发生变化时，它的所有依赖者对象都会收到通知并自动更新。
  - **观察者**对**主题**的数据改变感兴趣，希望**主题**一有变化，会通知它。

3.4 Observer Pattern（续）

- 当一个投资者在华夏股票公司注册账号后，可以收到股票信息中心的股市信息；
- 当投资者不炒股，去注销账号，就不会再收到股票行情了；
- 对于股票信息中心，它不知道每个投资者的具体情况的，它也无需关心这些，它仅仅通知观察者，观察者关注的主题发生了变化。

Observer Class Diagram

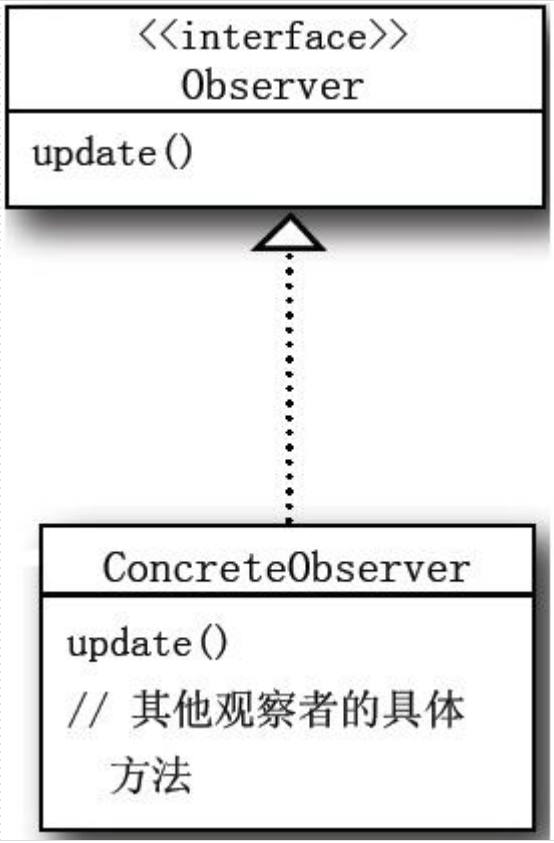


主题接口

一个具体主题

- 注册观察者
- 删除观察者
- 通知（更新）所有当前观察者

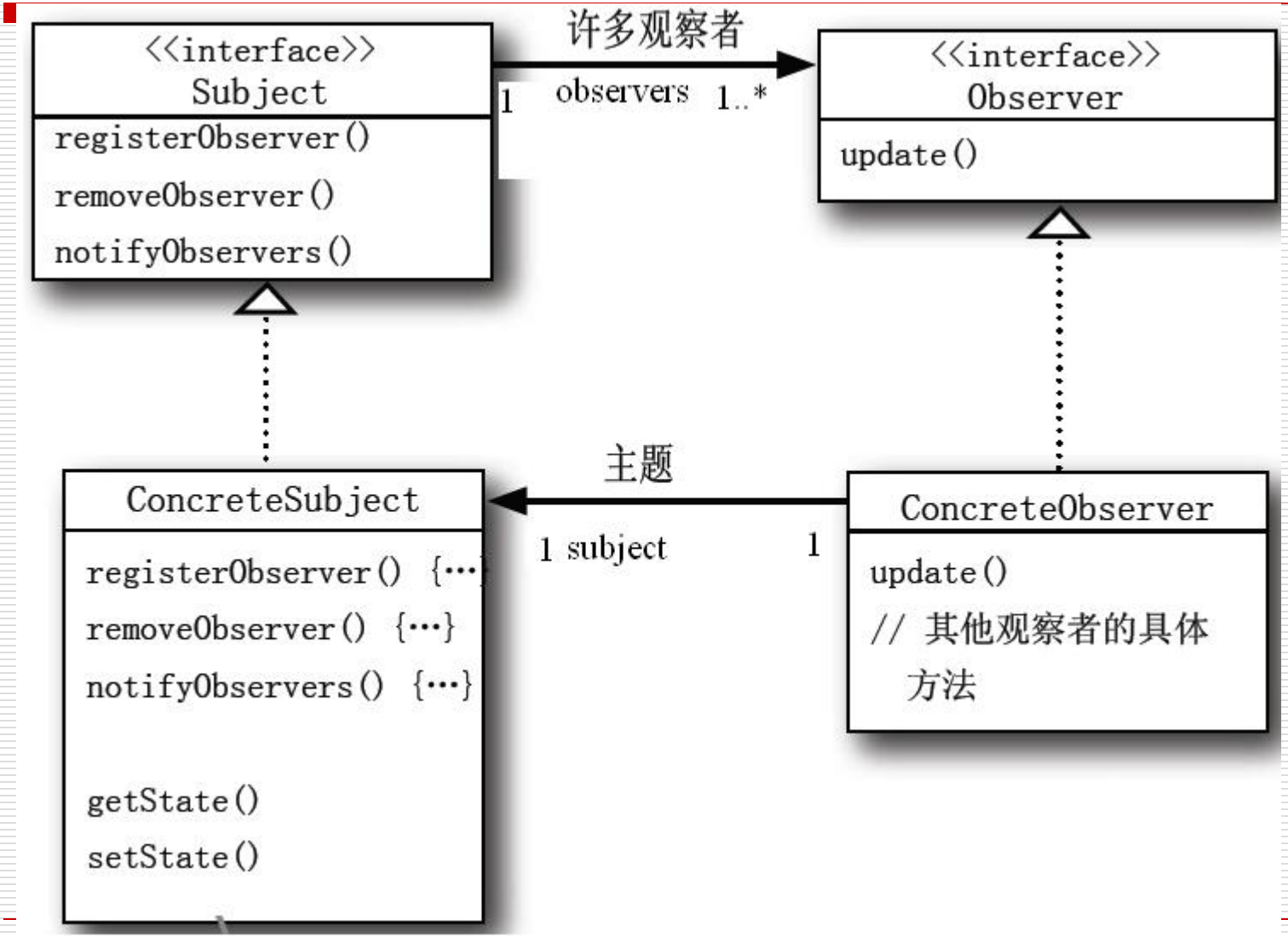
Observer Class Diagram（续）



观察者接口

具体观察者

Observer Class Diagram（续）

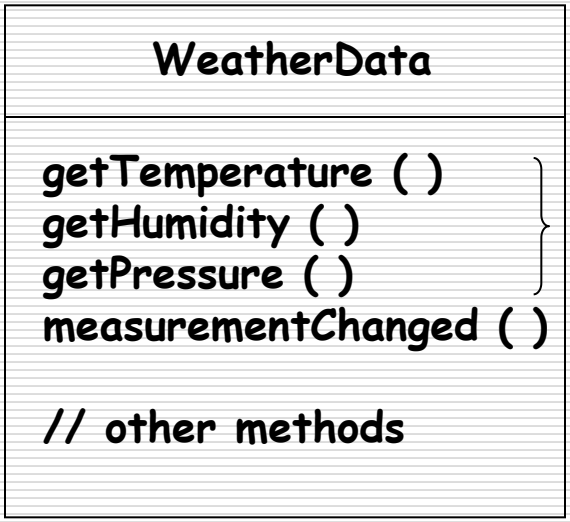


3.4 Observer Pattern（续）

- 上面介绍的就是一个典型的Observer设计模式。

3.4 Observer Pattern（续）

- 案例：目前公司有一个WeatherData类，其对象负责追踪目前的天气状况（温度、湿度和气压），希望在该类之上，建立一个应用软件，该应用软件可以发布三种布告板，分别显示目前的状况、气压统计和预报，当WeatherData 对象获得最新的测量数据时，三种布告板必须实时更新。



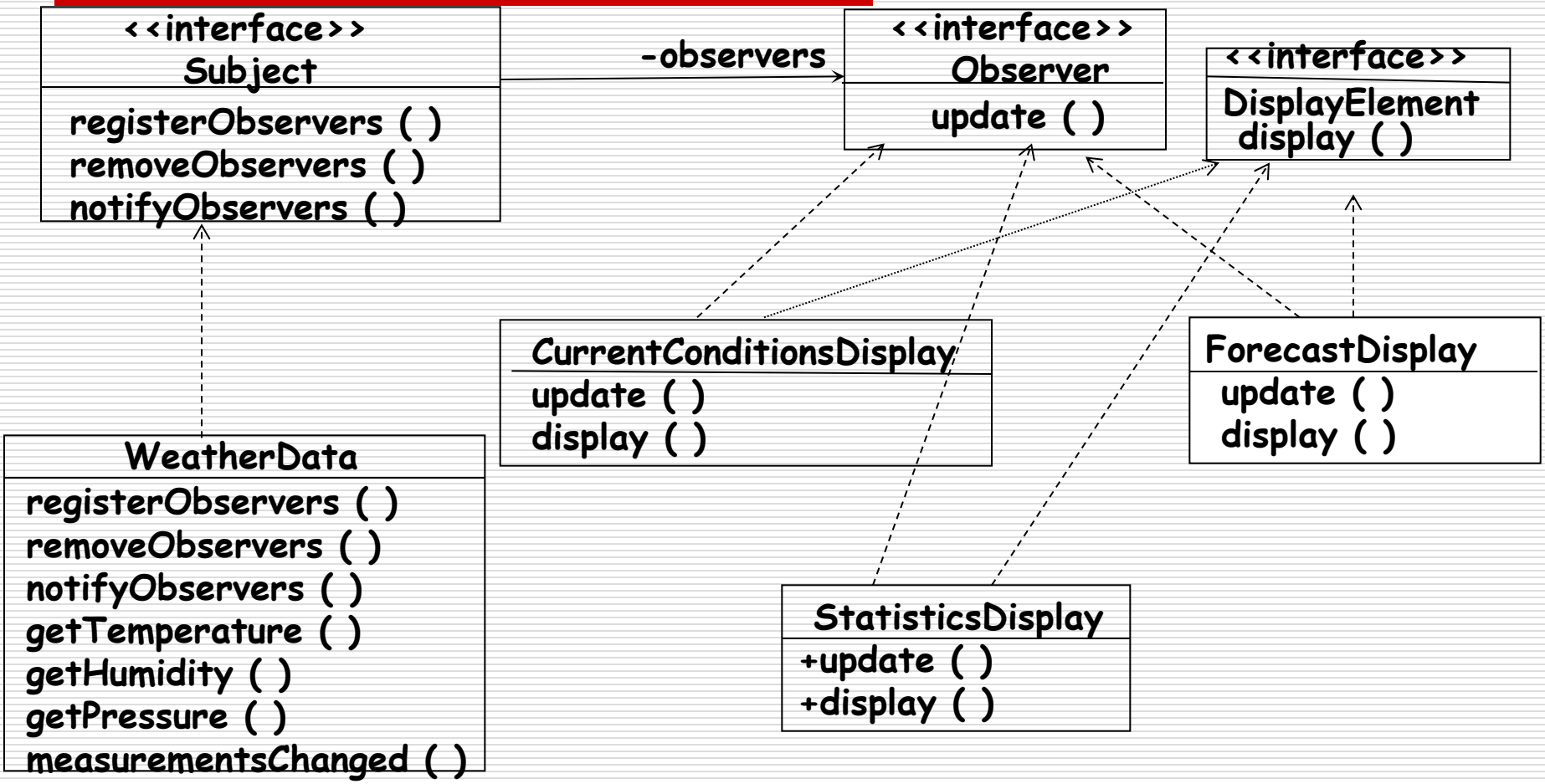
3.4 Observer Pattern（续）

第一号布告板

第二号布告板

第三号布告板

3.4 Observer Pattern（续）



Implementing the Weather Station

```
public interface Subject {
    public void registerObserver (Observer o);
    public void removeObserver (Observer o);
    public void notifyObservers ( );
}

public interface Observer {
    public void update (float temp, float humidity, float pressure);
}

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

Both of these methods take an Observer as an argument, that is the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

The Observer interface is implemented by all observers, so they all have to implement the update ( ) method.

The DisplayElement interface just includes one method, display ( ), that we will call when the display element needs to be displayed.

These are the state values the Observers get from the Subject when a weather measurement changes.

Implementing the Subject Interface in WeatherData

```
public class WeatherData implements Subject {
        private ArrayList observers;
        private float temperature;
        private float humidity;
        private float pressure;
        public WeatherData ( ){
            observers = new ArrayList ( );   }
       public void registerObserver (Observer o) {
            observers.add(o);  }
      public void removeObserver (Observer o) {
            int j = observer.indexOf(o);
            if (j >= 0) {
                              observers.remove(j);       } }
       public void notifyObservers ( ) {
            for (int j = 0; j < observers.size(); j++) {
              Observer observer = (Observer)observers.get(j);
              observer.update(temperature, humidity, pressure);
            }}
       public void measurementsChanged ( ) {
             notifyObservers ( ); }
      }
```

Added an ArrayList to hold the Observers, and we create it in the constructor

Here we implement the Subject Interface

Notify the observers when measurements change

The Display Elements

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperatue;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay (Subject weatherDataS) {
        this.weatherData = weatherDataS;
        weatherData.registerObserver (this);
    }
    public void update (float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display ( );
    }
    public void display ( ){
        System.out.println(" Current conditions : " + temperature + " F degrees
    and " + humidity + " % humidity" );
    }
}
```

Implements the Observer and DisplayElement interfaces

The constructors passed the weatherData object (the subject) and we use it to register the display as an observer.

When update ( ) is called, we save the temp and humidity and call display ( )

The display ( ) method just prints out the most recent temp and humidity.

Unit 3 Design Patterns

- 3.1 Singleton Pattern
- 3.2 Strategy Pattern
- 3.3 Factory Pattern
- 3.4 Observer Pattern
- ✓ 3.5 Adapter Pattern

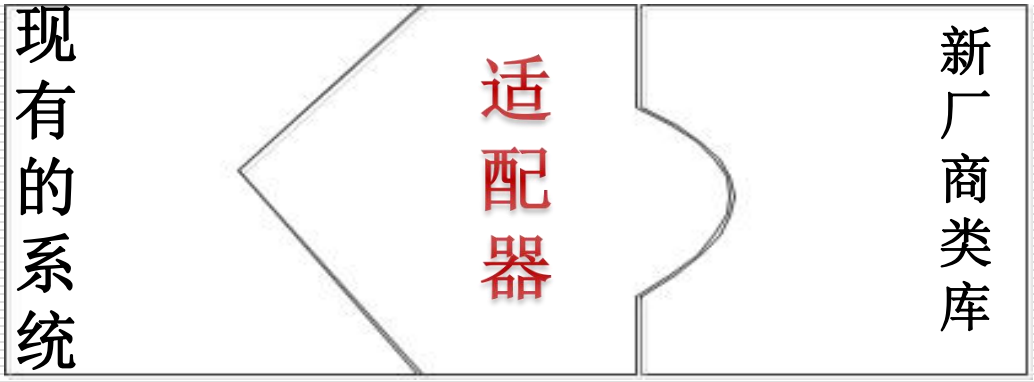
3.5 Adapter Pattern

Adapter

适配器改变了插座口的以符合本的笔电需求

3.5 Adapter Pattern(续)

- 面向对象适配器



3.5 Adapter Pattern(续)

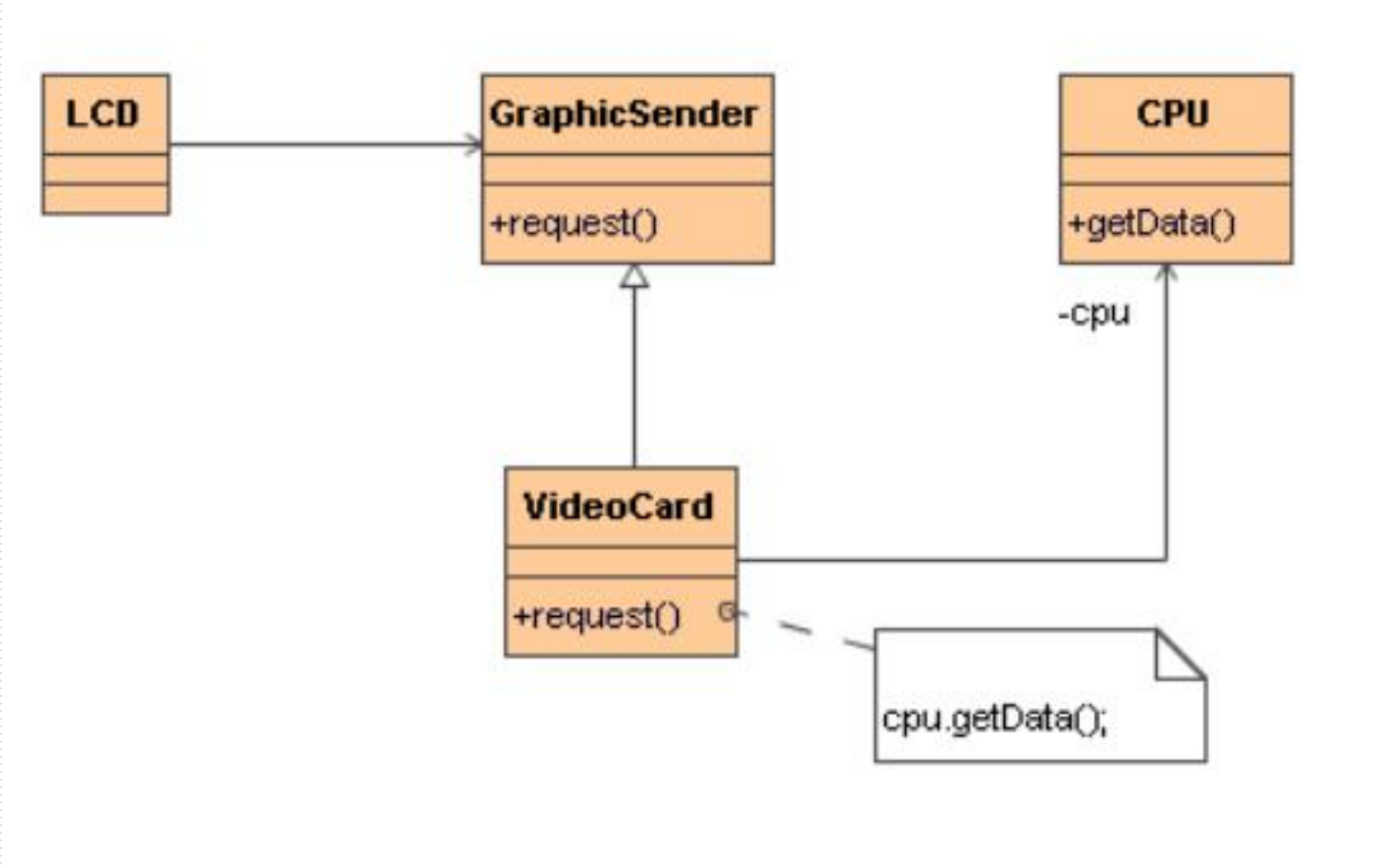
- 客户使用适配器的过程如下：
  ① 客户通过目标接口调用适配器的方法对适配器发出请求；
  ② 适配器使用被适配者接口把请求转换成被适配者的一个或多个调用接口；
  ③ 客户接受到调用的结果，但并未察觉这一切是适配器在起转换作用。
- 适配器将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

3.5 Adapter Pattern(续)

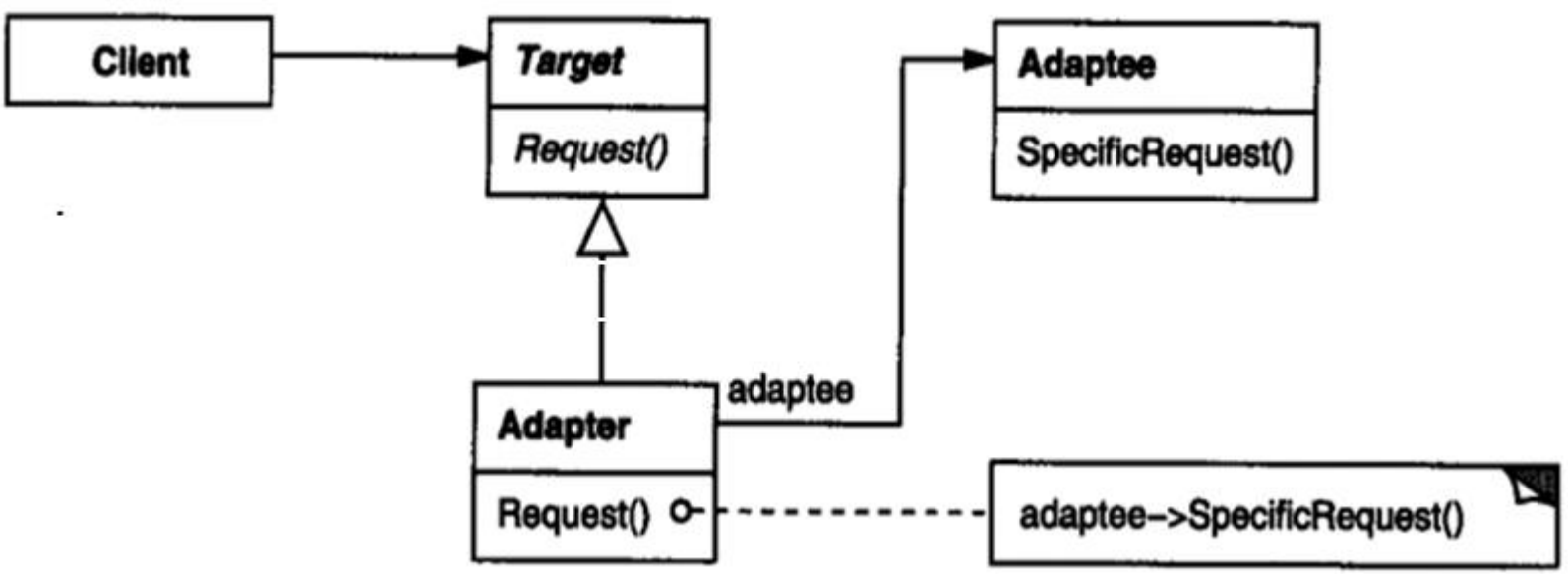
- 举例：
  - 显示器(**Client**)是用来显示图形的，它是不能显示数据，它只能够接受来自图形发送设备(**Target**)的信号。
  - 可是只有CPU (Adaptee)这个产生各种描述图形的数据的数据发送器。如何将这些数据让显示器进行显示？这两个部件是不兼容的。
  - 于是需要一个中间设备，它能够将CPU“适配”于显示器，这便是显卡——图形适配器(**Adapter**)。

3.5 Adapter Pattern(续)

解决方案



对象的适配器设计模式



- Client： 客户，有请求
- Adaptee ： 被适配者，所有的请求都委托给被适配者完成
- Target： 目标抽象类，定义客户要用的请求接口
- Adapter： 适配器，调用被适配者的接口，实现目标中用户请求的接口

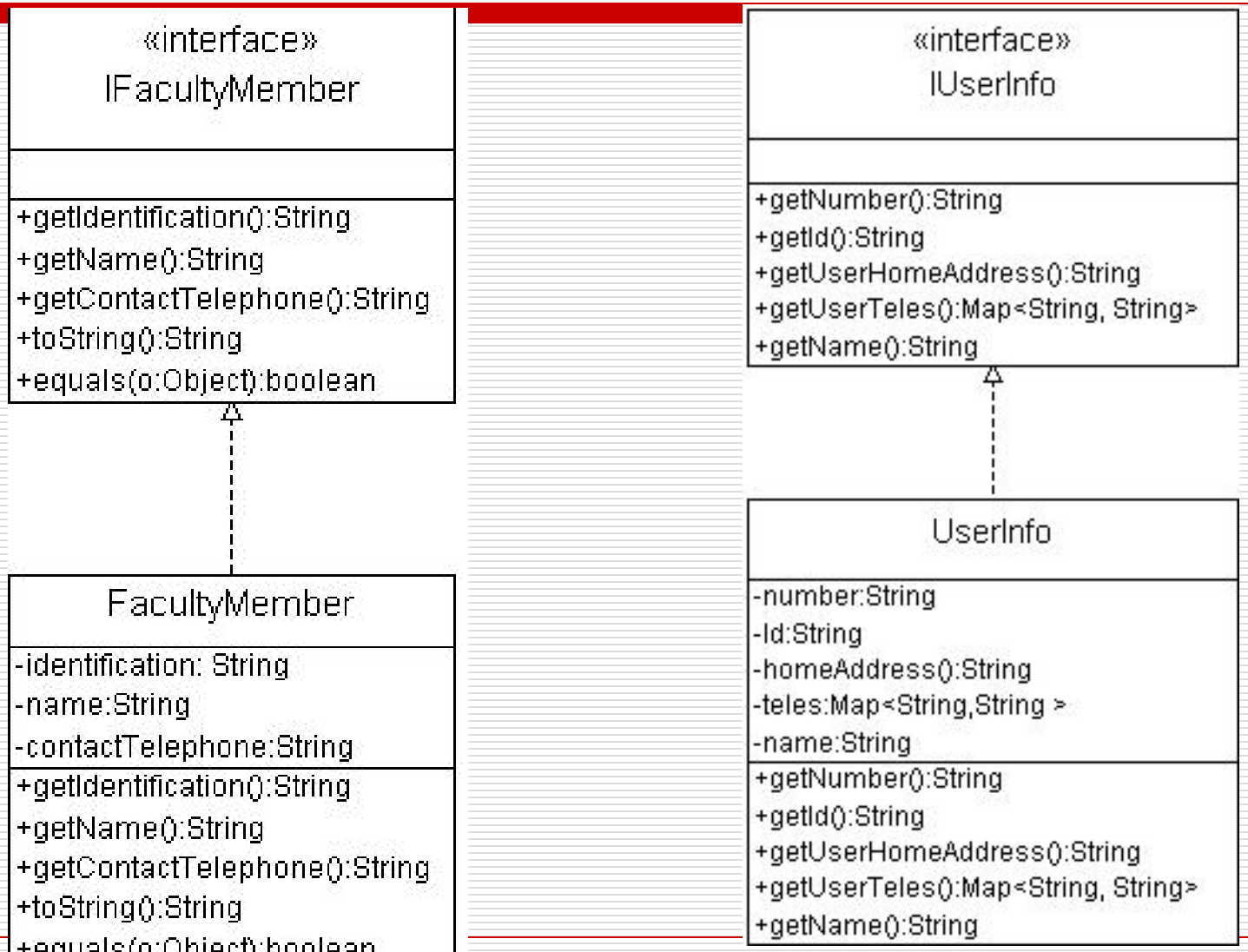
3.5 Adapter Pattern(续)

- 适配器模式Java程序示例：
  - LCD.java, CPU.java, GgraphicSender.java, VideoCard.java

3.5 Adapter Pattern(续)

- 软件学院职工信息系统
  - 校聘职工信息接口 IFacultyMember，员工信息类FacultyMember
- 院聘职工通过劳动服务公司进行聘用，劳动服务公司的人员信息系统提供的信息接口为IUserInfo，员工信息类为UserInfo
- 学校人事处要求软件学院职工信息系统需要与劳动服务公司人员信息系统进行远程联机交互，同步劳动服务公司的信息，怎么办？

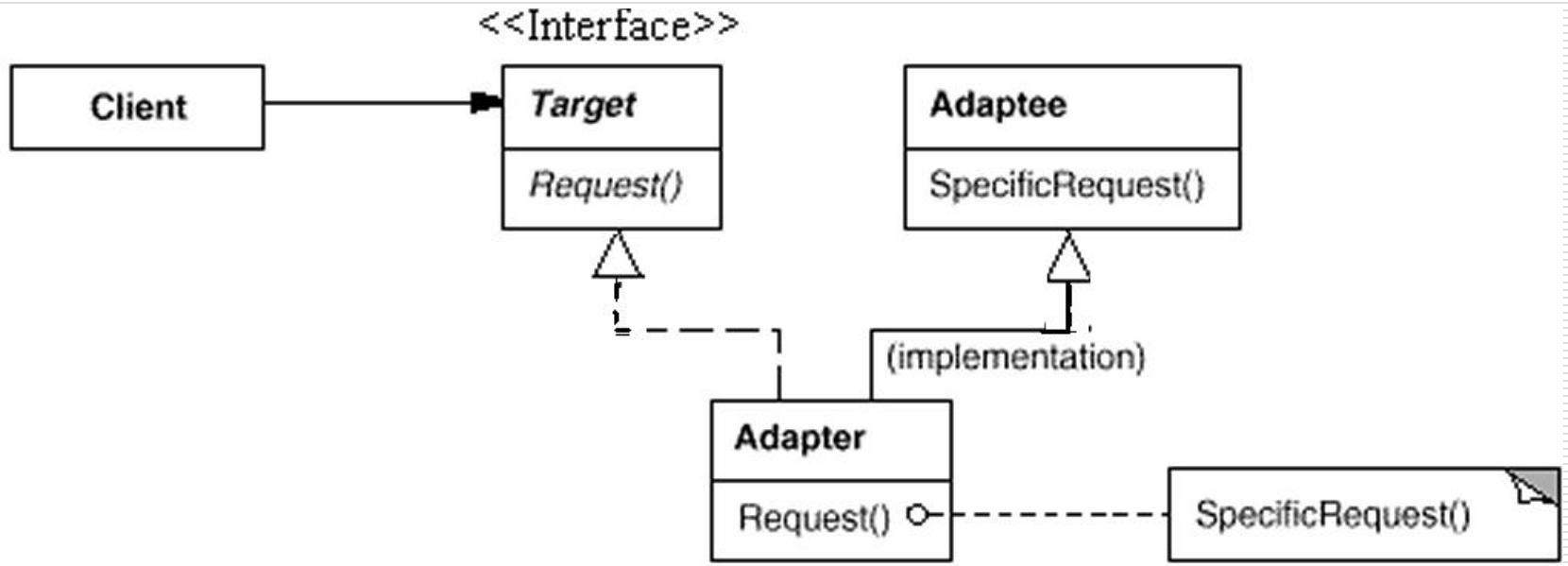
3.5 Adapter Pattern(续)



对象适配器设计模式

- 对象适配器设计模式中的Adaper和Adaptee直接的关系可否是继承关系？
- 如果是继承关系，构成类适配器设计模式
- 对象适配器使用关联关系，更灵活，可以适配某个类以及该类的子类，将工作委托给适配者进行，让工作更具备弹性。

类适配器设计模式



- 类适配器不是使用关联（即组合）来适配被适配者，而是继承被适配者和目标类。

在什么情况下使用适配器模式

- 在以下各种情况下使用适配器模式：
  - 1、使用第三方组件，而这个组件的接口与目前系统接口不兼容（如方法与系统方法不一致等），可以使用适配器模式解决接口不兼容问题。
  - 2、使用早前项目一些有用的类，可以用适配器模式解决现有接口与原有对象接口不兼容问题。