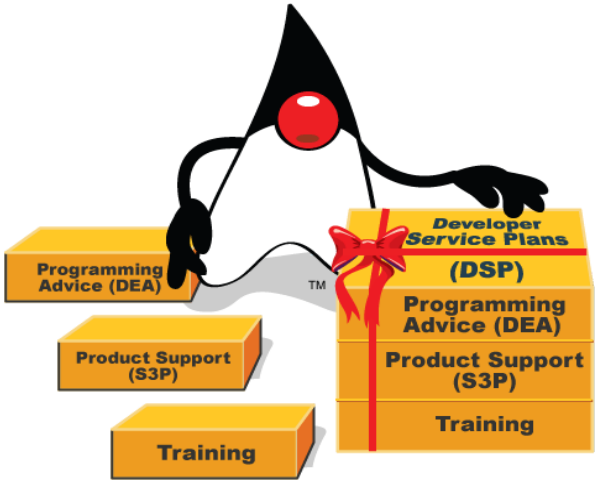
Chap 6 接口和包管理



6.1 接口

6.6.1 接口的定义

接口(interface)定义了在类的体系中可以被任意一个类实现的行为协议（行为规范），但接口本身并不提供规范的实现。

接口的规范定义和规范实现相互分离的原则，使得类(对象)可以实现面向接口的松耦合。从而可以帮助系统实现良好的扩展性（可插入性），满足开闭法则。

举例：标准电源插座和电器的标准插头

6.1.2 接口的语法结构

在语法构成上，接口是若干没有被具体实现的方法的集合，在接口中也可以定义常数。

接口和抽象类之间的不同点：

*接口中不可以有任何方法的具体实现，但抽象类可以
*一个类可以实现多个接口但只能有一个父类
*接口不是类体系的一个部分，不相关的类可以实现同样的接口

语法构成上，接口的由两部份组成：
接口声明和接口体

- 接口声明

public	Makes this interface public.
interface InterfaceName	Class cannot be instantiated.
Extends SuperInterfaces	This interface's superinterfaces.
{ *InterfaceBody* }	

接口声明的组成部分和各自用途.

- 接口体

* 接口内的方法定义后紧跟一个分号(;)

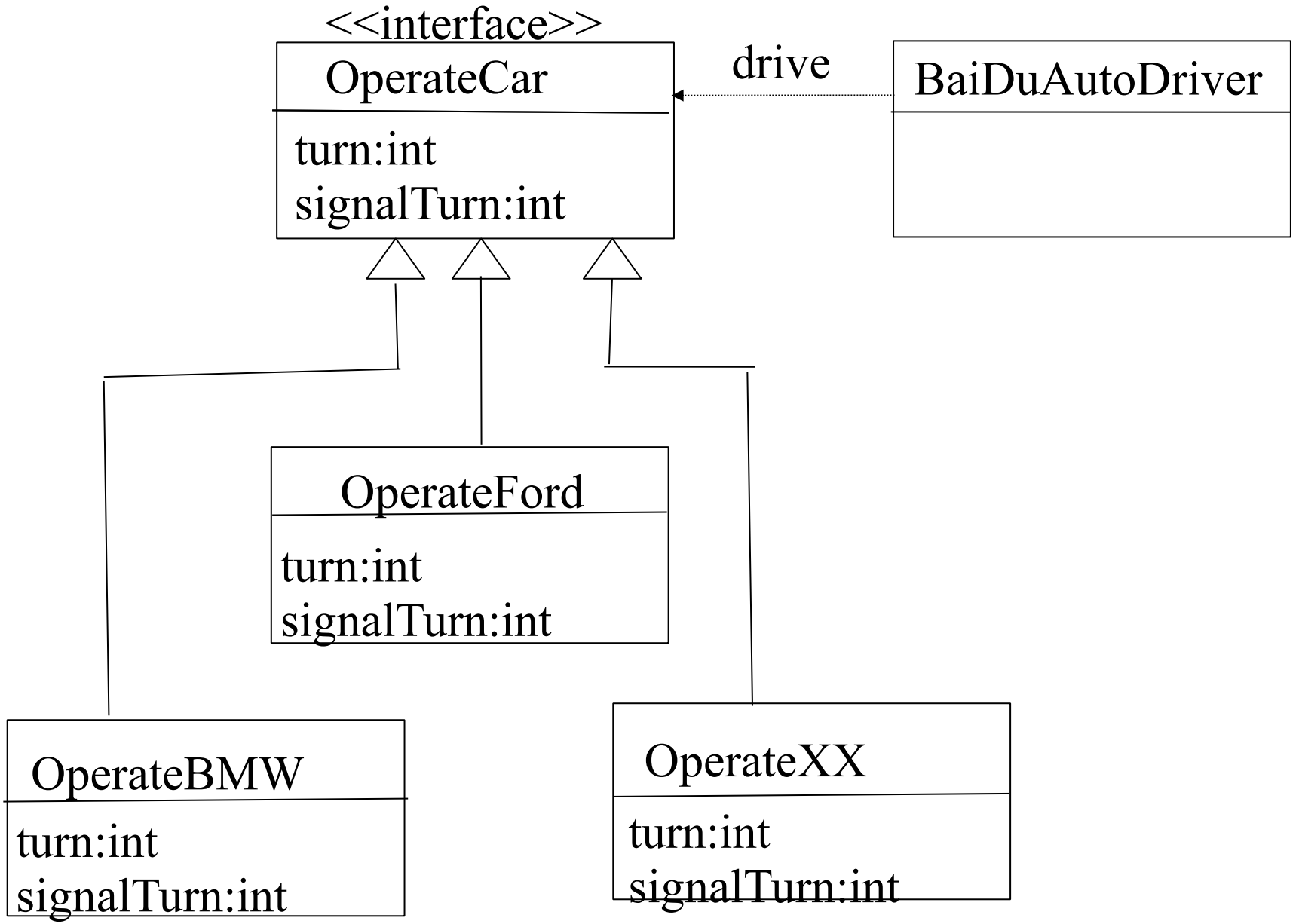
* 接口内的所有方法隐含为public和abstract （公有，抽象方法）

*接口内定义的所有常数隐含为public,static和final

* 接口中定义的方法不能使用transient,volatile,synchronized这些修饰字

举例：

问题：自动汽车导航驾驶系统。如何设计汽车制造厂商，自动导航软件厂商的各个类之间的关系？


<<interface>>
OperateCar
turn:int
signalTurn:int
drive
BaiDuAutoDriver
OperateFord
turn:int
signalTurn:int
OperateBMW
turn:int
signalTurn:int
OperateXX
turn:int
signalTurn:int

OperateCar 接口定义:

```
public interface OperateCar{

  //method
 int turn(Direction direction, double radius….);
 int signalTurn(Direction direction, boolean signalOn);

}
```

某个汽车制造商对OperateCar接口的实现；

```
public class OperateBMW implements OperateCar{

    int turn(Direction direction, double radius….){

      ……
   }
   int signalTurn(Direction direction, boolean signalOn){

      ……
   }

}
```

需要对接口中定义的所有方法都进行实现。

导航软件对象使用汽车对象，进行导航操作

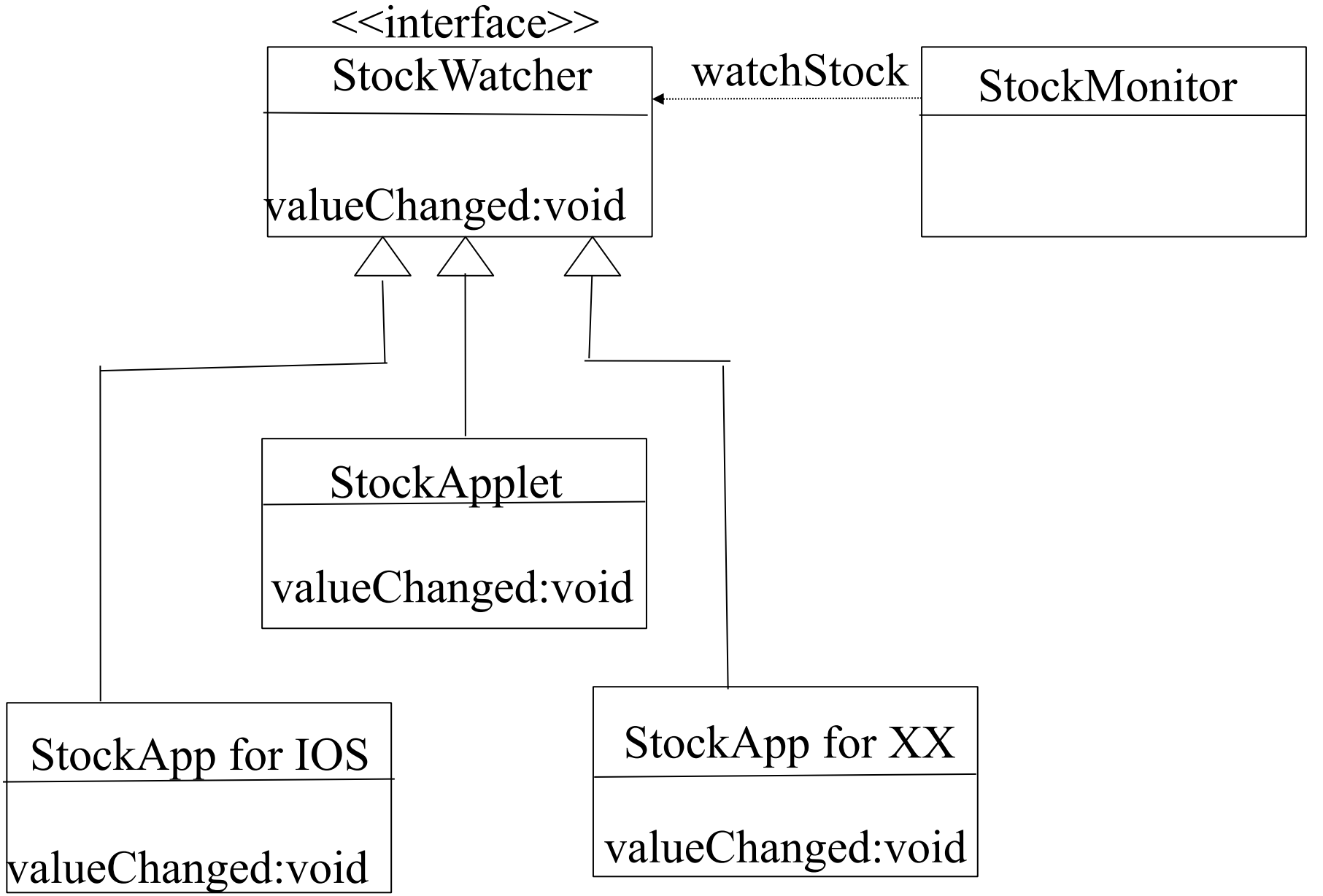
接口名也可以作为一个类型使用。但一个对象只有在它的类实现了接口时才能使用一个对应接口类型的变量对它进行引用。与此同时实现了可插入性。

```
public class BaiduAutoDriver {

    public void drive(OperateCar oc) {
    ……

      oc.turn(……);
    }

}
```

举例：股票交易信息系统

设计目标：某只股票价格变动时候，能告知各种类型的客户端股票价格的变动。

举例：股票交易信息系统


<<interface>>
StockWatcher
valueChanged:void
watchStock
StockMonitor
StockApplet
valueChanged:void
StockApp for IOS
valueChanged:void
StockApp for XX
valueChanged:void

举例：股票交易信息系统

```
public interface StockWatcher {
   final String  sunTicker = "SUNW"; //constant
   final String oracleTicker = "ORCL"; //constant
   final String ciscoTicker = "CSCO"; //constant
   void valueChanged(String tickerSymbol, double newValue); //method
}
```

6.1.3 实现接口

类使用implements语句来声明对接口的实现。实现接口的类必须实现定义在这个接口中的所有方法。

举例:

```
public class StockApplet extends Applet implements StockWatcher {
    ...
    public void valueChanged(String tickerSymbol, double newValue) {
        if (tickerSymbol.equals(sunTicker)) {
            ...
        } else if (tickerSymbol.equals(oracleTicker)) {
            ...
        } else if (tickerSymbol.equals(ciscoTicker)) {
            ...
        }
    }
} //接口常数可以使用简单命名，比如 sunTicker
```

6.1.4 使用接口类型

接口名也可以作为一个类型使用。但一个对象只有在它的类实现了接口时才能使用一个对应接口类型的变量对它进行引用。与此同时实现了可插入性。

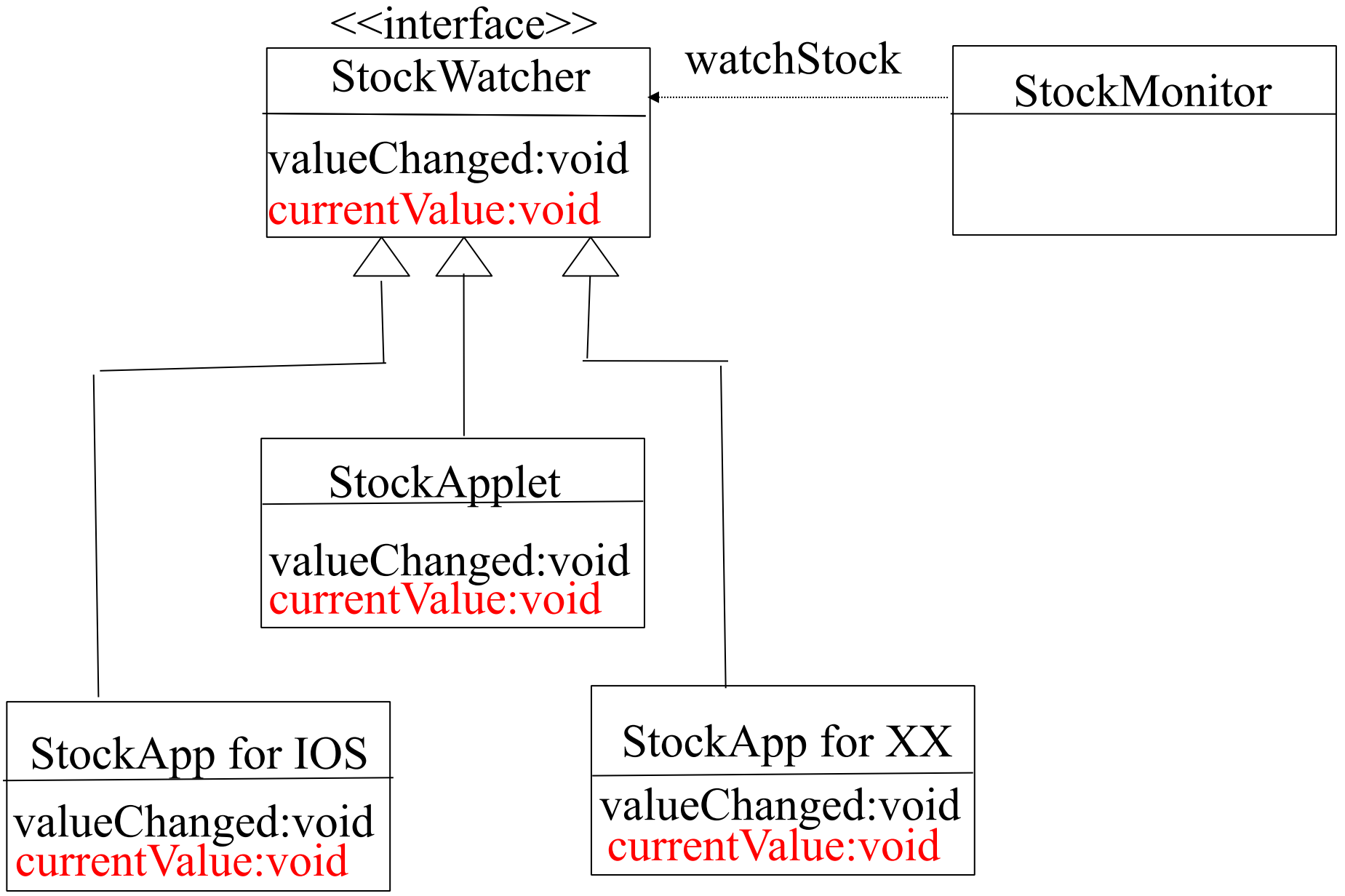
```
public class StockMonitor {
    public void watchStock(StockWatcher watcher,
                           String tickerSymbol, double delta) {
    ...
    }
}
```

6.1.5 注意! 接口不能生长

接口中的任何变化将会影响到实现它的类

举例;

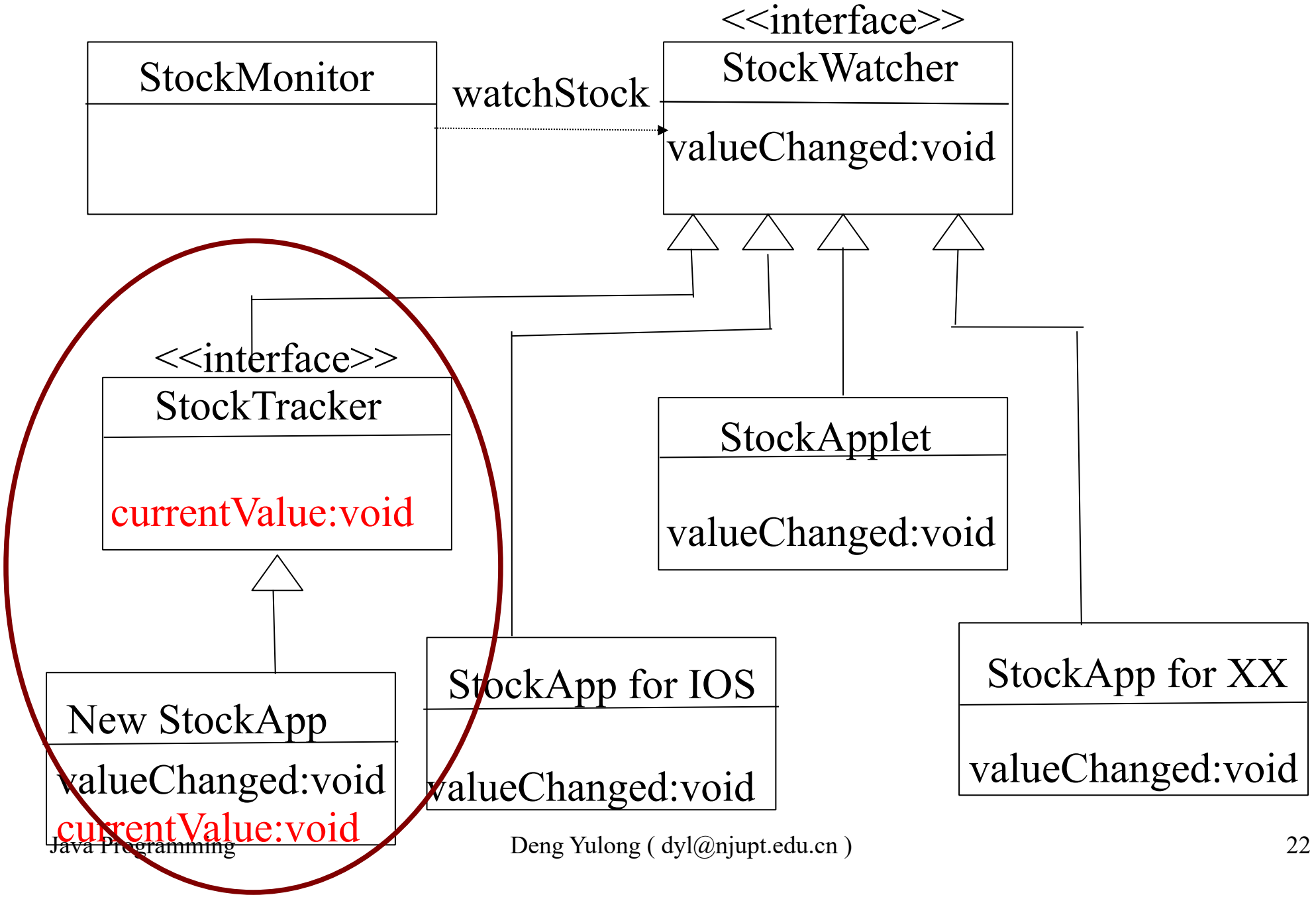
```
public interface StockWatcher {
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";

    void valueChanged(String tickerSymbol, double newValue);
    void currentValue(String tickerSymbol, double newValue); //新增加
}
```


<<interface>>
StockWatcher
valueChanged:void
currentValue:void
watchStock
StockMonitor
StockApplet
valueChanged:void
currentValue:void
StockApp for IOS
valueChanged:void
currentValue:void
StockApp for XX
valueChanged:void
currentValue:void

```
public class StockApplet extends Applet implements StockWatcher {
    ...
    public void valueChanged(String tickerSymbol, double newValue) {
        if (tickerSymbol.equals(sunTicker)) {
            ...
        } else if (tickerSymbol.equals(oracleTicker)) {
            ...
        } else if (tickerSymbol.equals(ciscoTicker)) {
            ...
        }
    }
    //需要增加对新增加接口方法currentValue的实现
    void currentValue(String tickerSymbol, double newValue){
     ……
    }
}
```

在这种情况下，应该创建一个子接口来反应接口的变化。

```
public interface StockTracker extends StockWatcher {
   void currentValue(String tickerSymbol, double newValue);
}
```

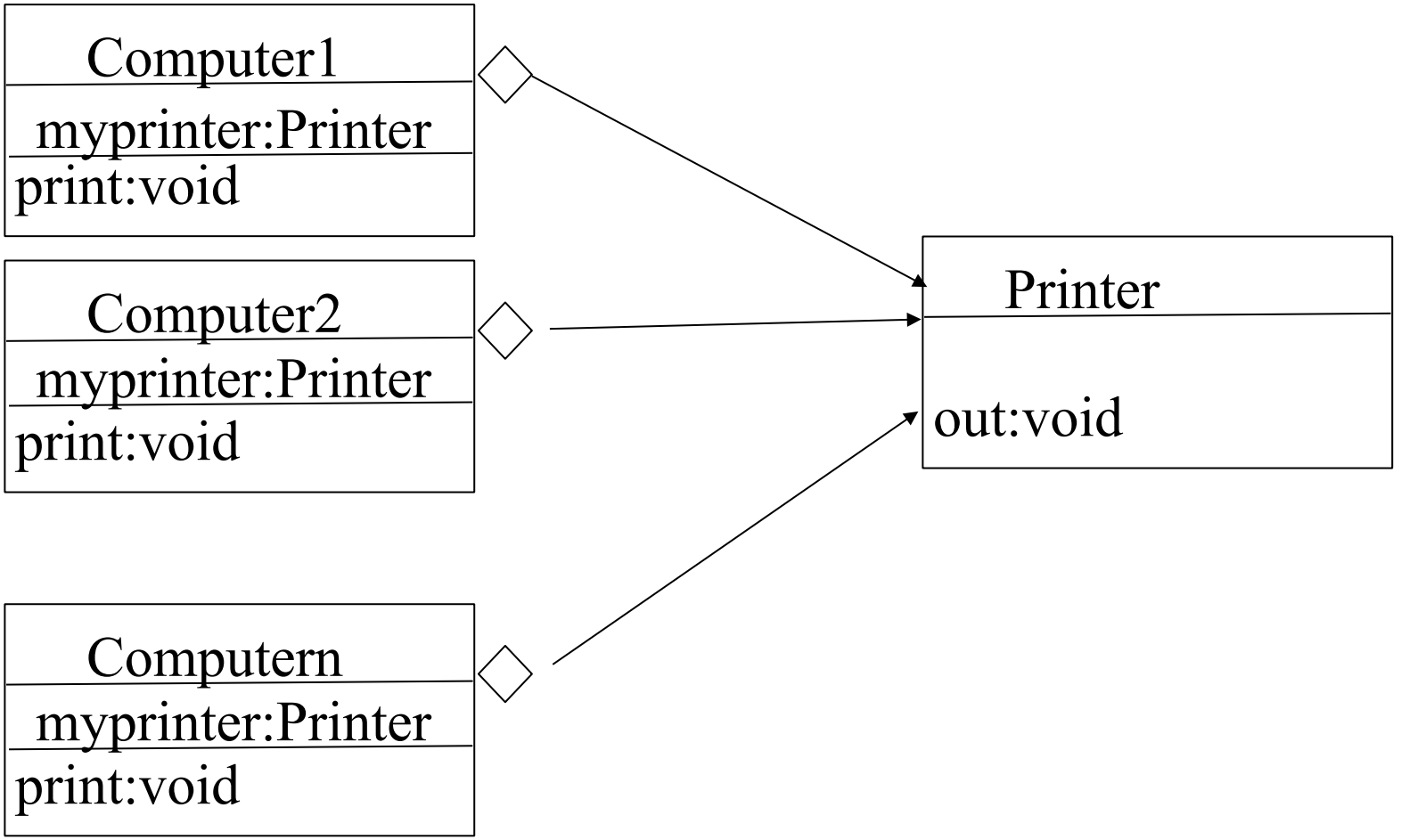

<<interface>>
StockWatcher
valueChanged:void
StockMonitor
watchStock
<<interface>>
StockTracker
currentValue:void
StockApplet
valueChanged:void
New StockApp
valueChanged:void
currentValue:void
StockApp for IOS
valueChanged:void
StockApp for XX
valueChanged:void

6.1.6 系统设计中利用接口实现可插入性。

接口体现了规范和实现相互分离的设计原则，从而降低了模块之间的耦合度，是实现可插入性的基础。

举例：

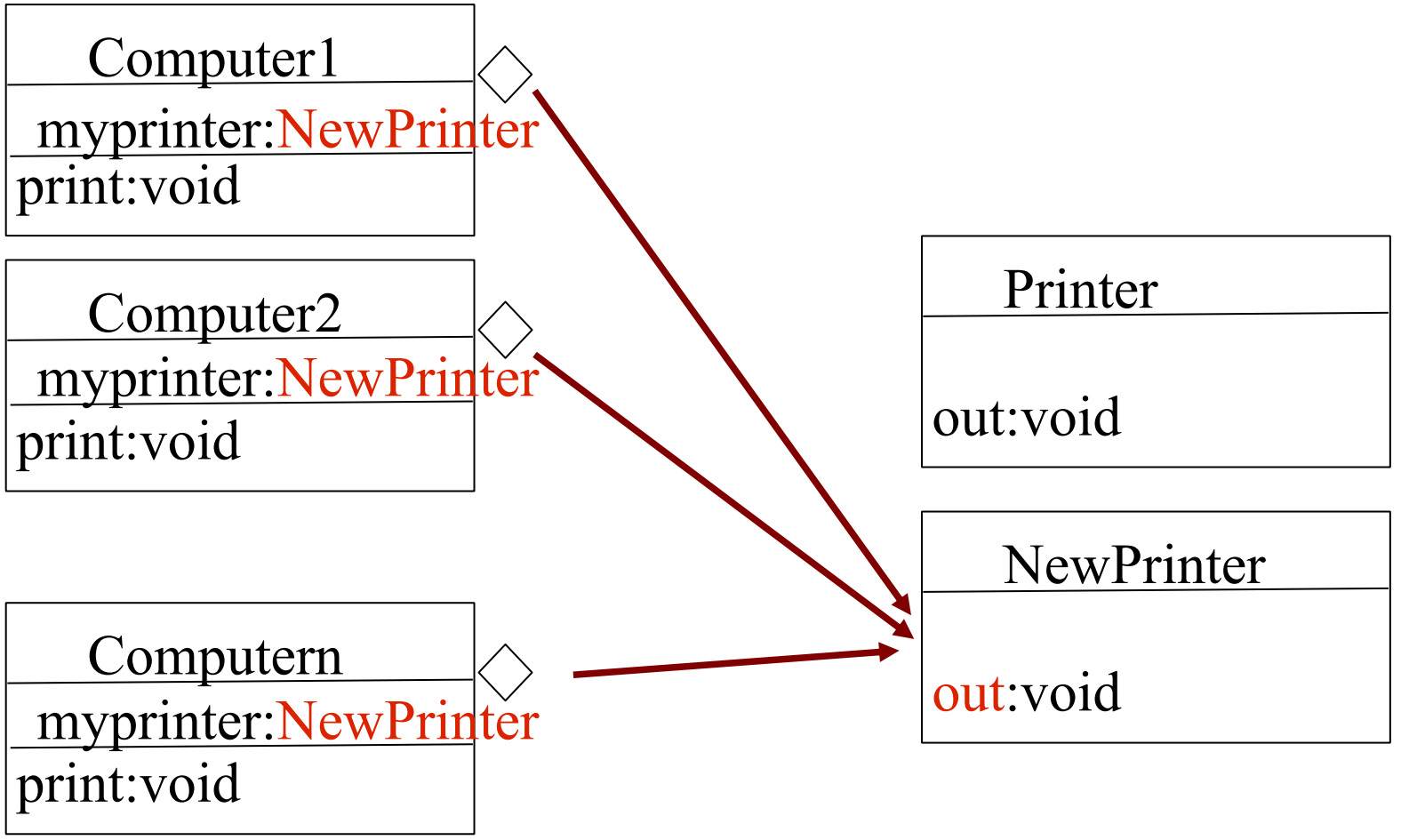
问题：在系统中有n个对象 Computer (computer1~n)需要使用输出设备,比如printer。如何设计类之间的关系？


Computer1
myprinter:Printer
print:void
Computer2
myprinter:Printer
print:void
Computern
myprinter:Printer
print:void
Printer
out:void

```
public class ComputerX
  {
   private Printer myprinter;
  public Computer(Printer myprinter)
      {  this.myprinter = myprinter;
       }
  public void print()
      {
     myprinter.out();
      }
   }
```

```
public class Printer
{
 public void out(){ ... }
}
```

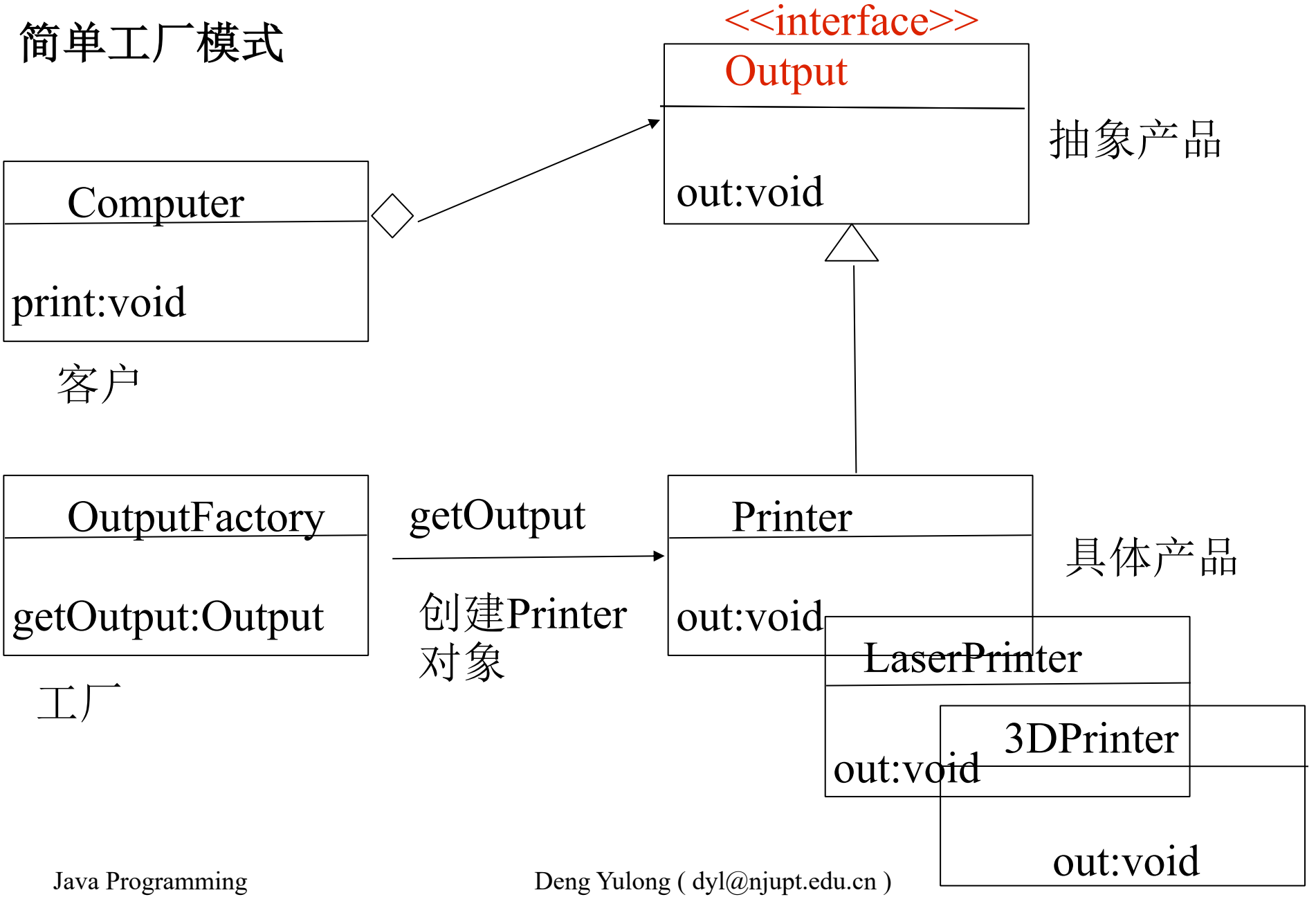
新型打印机 (NewPrinter)出现



```
public class ComputerX
  {
   private NewPrinter myprinter;
  public Computer(NewPrinter myprinter)
      {  this.myprinter = myprinter;
       }
  public void print()
      {
     myprinter.out();
      }
   }
```

```
public class NewPrinter
{
 public void out(){ ... }
}
```

简单工厂模式


<<interface>>
Output
out:void
抽象产品
Computer
print:void
客户
OutputFactory
getOutput:Output
工厂
getOutput
创建Printer
对象
Printer
out:void
具体产品
LaserPrinter
out:void
3DPrinter
out:void

```
public class Computer
  {
   private Output out;
  public Computer(Output out)
      {  this.out = out;
       }
  public void print()
      {
     out.out();
      }
   }
```

```
public class OutputFactory
 {
  public Output getOutput()
   {  return new Printer();
   }

  public static void main(String args[])
   {
   OutputFactory of = new OutputFactory();
   Computer  c = new Computer(of.getOutput());
    c.print();
   }

 }
```

```
public class Printer implements Output
 {
   ......
  public void out()
   {
   ......
   }
   ......
  }

 interface Output
  {
   void out();
  }
```

6.2 创建和使用包(packages)

包是由相关的类和接口构成的集合，它提供了访问控制机制以及名字空间管理。

举例:

```
//in the Graphics.java file
public abstract class Graphic {  . . .}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {  . . .}

//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {  . . .}

//in the Draggable.java file
public interface Draggable {  . . .}
```

考虑到如下原因，你应该将这些类和接口放入一个包中：

*你和其他程序员能确定这些类和接口是相关的。
*你和其他程序员知道到何处能找到提供图形相关功能的类和接口。
*由于包创建了新的名字空间，你的类名不会和其他包里的类名发生冲突。
*你可以允许在一个包中的类彼此之间自由访问，而与此同时对包以外的类的访问进行控制。

6.2.1 创建一个包

你必须在定义类和接口的每一个Java源文件的开头包含一条package语句，以便能将这些类和接口放入包中。

举例:

```
// in the Circle.java file

package graphics;

public class Circle extends Graphic implements Draggable {
   . . .
}
```

```
// in the Rectangle.java file


package graphics;

public class Rectangle extends Graphic implements Draggable {
    . . .
}
```

注意!

*** 一个源文件中可以包含多个类的定义但其中只能有一个是主类,即声明为public并且其名字与源文件名相同。**

*** 在包中只有声明为public的成员才能够从包外进行访问。**

6.2.2 包的命名方法

同样名字的类如果包含不同的包的名称，将会形成不同的限定命名
举例:
graphics. Rectangle
java.awt.Rectangle

命名约定

软件公司采用反向**Internet**域名作为包名的开头部分
例如： **abc**公司的域名 **abc.com**
定义包名为： **com.abc.graphics**
Rectangle 的限定命名为： **com.abc.graphics.Rectangle**

6.2.3使用包中的成员

在包外使用包中一个声明为**public**的成员，可以通过下面方式：

*通过限定命名方法对成员进行引用，这样名字可能比较长

* 引入(import)包的成员

* 引入(import)含有这个成员的整个包

•通过名字来引用包中的成员

*** 如果你写的代码与需要引用的成员处于同一个包中，或者是含有这个成员的包已经被引入(import)时，可以使用简单命名方式来使用这个成员。**

***如果你需要从一个包中对另一个包中的成员进行访问，或者是含有该成员的包没有被引入(import)时，需要使用限定命名方式来使用这个成员。**

举例:

```
// 包名为 graphics
graphics.Rectangle myRect = new graphics.Rectangle();
```

- 引入包中的成员

可以使用**import**语句来将某个成员引入到当前文件中，**import**语句放在源文件的开头，同时该语句需要置于所有的类或接口定义的前面并位于**package**语句的后面。

举例:

```
// 包名为 graphics


import graphics.Circle;


之后可以使用简单命名来访问 Circle 类:

Circle myCircle = new Circle();
```

•引入整个包

使用**import**语句和*符号引入一个包中所有的类和接口：

举例：

// 包名为 **graphics**

import graphics.*;

之后可以使用简单命名来访问**graphics**包中所有的类

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

Java运行环境会自动引入下面三个包

*** 默认包 (不使用package语句，这个包没有名字)**
*** java.lang 包**
***当前包**

- 同名问题解决

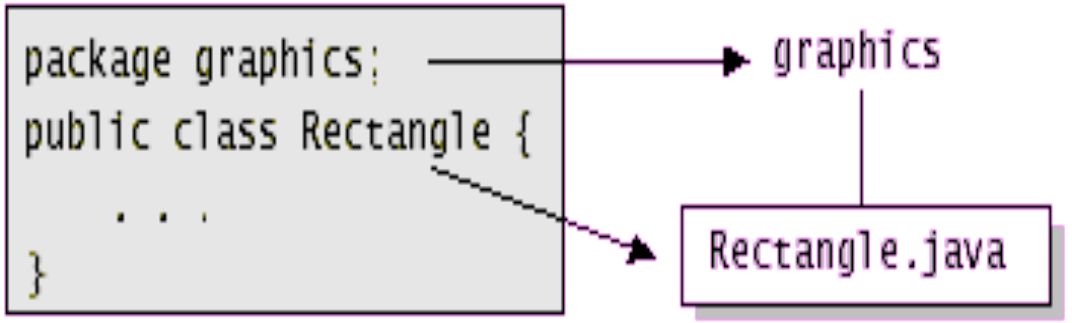
如果在一个包中的成员与另一个包中的成员名字相同，但两个包都被引入了，此时仍然需要使用限定命名方法来访问成员以便加以区分。

举例:

```
graphics.Rectangle rect;
java.awt.Rectangle rect1;
```

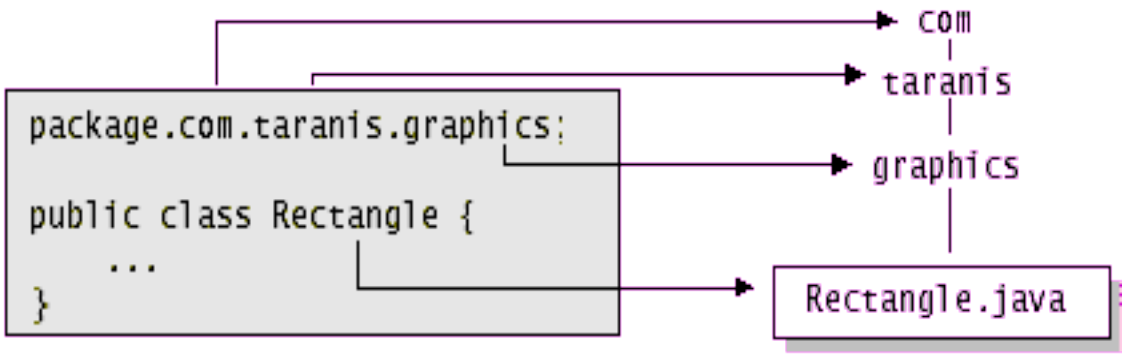
6.2.4 管理源代码和(class)字节码文件

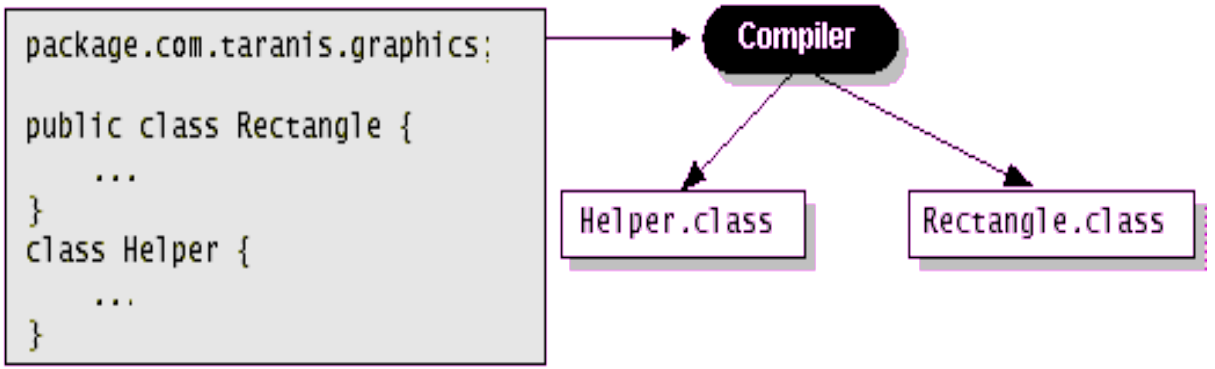
需要将源文件按照含有相应类或接口定义的包的命名方法放入对应的目录中

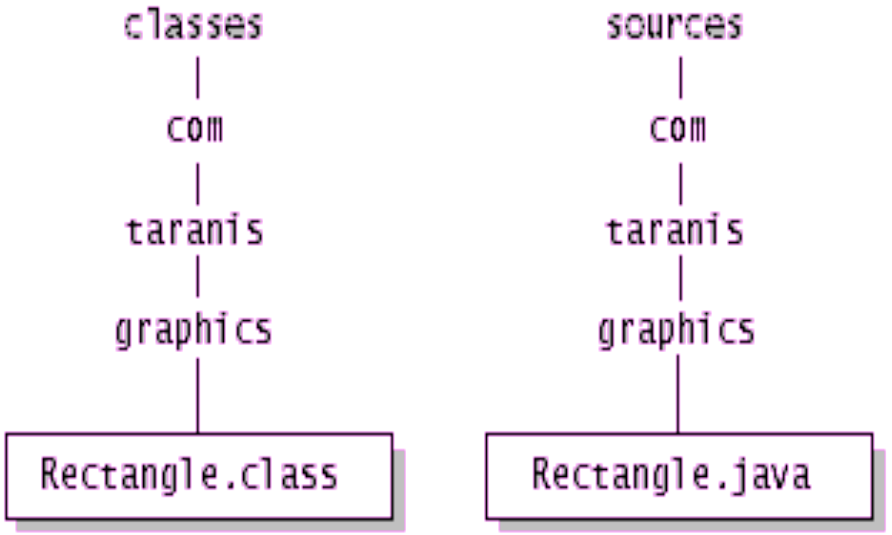


类名　　**graphics.Rectangle**
文件路径　**graphics/Rectangle.java (Unix)**
graphics\Rectangle.java (Win32)

举例：（包含源文件和class文件）






classes
com
taranis
graphics
Rectangle.class
sources
com
taranis
graphics
Rectangle.java

*** 源程序以及class文件需要按照目录对应的方式进行管理，这样编译器和解释器才能找到程序中使用的类和接口，编译器或解释器通过搜寻class path中的每一个目录或ZIP文件来查找需要的类或接口**

*** 列入class path中的目录应该为包对应目录路径中的第一层目录。**

***默认情况下，编译器和解释器搜寻当前目录以及包含Java平台class文件的ZIP 文件，大多数情况下，在这两个位置能找到需要的类或接口。但特殊情况下，仍然需要仔细设置你自己的class path以便帮助编译器和解释器寻找到需要的类或接口。**