

【软件构造】第五章第二节 设计可复用的软件

---

## 第五章第二节 设计可复用的软件

5-1节学习了可复用的层次、形态、表现；本节从类、API、框架三个层面学习如何设计可复用软件实体的具体技术。

### Outline

- 设计可复用的类——LSP
  - 行为子结构
  - 协变与逆变
  - Liskov替换原则(LSP)
- 各种应用中的LSP
  - 数组是协变的
  - 泛型中的LSP
  - 为了解决类型擦除的问题-----Wildcards（通配符）
- 设计可复用的类——委派与组合
- 设计可复用库与框架
  - 白盒框架
  - 黑盒框架

### Notes

#### ## 设计可复用的类——LSP

- 在OOP之中设计可复用的类
  - 封装和信息隐藏
  - 继承和重写
  - 多态、子类和重载
  - 泛型编程
  - LSP原则
  - 委派和组合（Composition）

#### 【行为子结构】

- 子类型多态（Subtype polymorphism）：客户端可用统一的方式处理不同类型的对象。
- 栗子：

```
Animal a = new Animal();  
Animal c1 = new Cat();  
Cat c2 = new Cat();
```

在可以使用a的场景，都可以用c1和c2代替而不会有任何问题。

- 在java的静态类型检查之中，编译器强调了几条规则：
  - 子类型可以增加方法，但不可删
  - 子类型需要实现抽象类型中的所有未实现方法
  - 子类型中重写的方法必须有相同或子类型的返回值
  - 子类型中重写的方法必须使用同样类型的参数
  - 子类型中重写的方法不能抛出额外的异常
- 行为子结构也适用于指定的方法：
  - 更强的不变量
  - 更弱的前置条件
  - 更强的后置条件

行为子结构的示例一：

<pre>abstract class Vehicle {      int speed, limit;     //@ invariant speed &lt; limit;      //@ requires speed != 0;     //@ ensures speed &lt; \old(speed)     void brake(); }</pre>	<pre>class Car extends Vehicle {     int fuel;     boolean engineOn;     //@ invariant speed &lt; limit;     //@ invariant fuel &gt;= 0;      //@ requires fuel &gt; 0 &amp;&amp; !engineOn;     //@ ensures engineOn;     void start() { ... }     void accelerate() { ... }      //@ requires speed != 0;     //@ ensures speed &lt; \old(speed)     void brake() { ... } }</pre>
---	---

- 子类满足相同的不变量（同时附加了一个）
- 重写的方法有相同的前置条件和后置条件
- 故该结构满足LSP

行为子结构的示例二：

<pre>class Car extends Vehicle {     int fuel;     boolean engineOn;     //@ invariant speed &lt; limit;     //@ invariant fuel &gt;= 0;      //@ requires fuel &gt; 0 &amp;&amp; !engineOn;     //@ ensures engineOn;     void start() { ... }     void accelerate() { ... }      //@ requires speed != 0;     //@ ensures speed &lt; \old(speed)     void brake() { ... } }</pre>	<pre>class Hybrid extends Car {     int charge;     //@ invariant charge &gt;= 0;      //@ requires (charge &gt; 0        fuel &gt; 0) &amp;&amp; !engineOn;     //@ ensures engineOn;     void start() { ... }     void accelerate() { ... }      //@ requires speed != 0;     //@ ensures speed &lt; \old(speed)     //@ ensures charge &gt; \old(charge)     void brake() { ... } }</pre>
---	--

- 子类满足相同的不变量（同时附加了一个）
- 重写的方法 start 的前置条件更弱
- 重写的方法 brake 的后置条件更强
- 故该结构满足LSP

行为子结构的示例三：

```

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

```

Is this Square a behavioral subtype of Rectangle?

- 子类满足不变量条件更强，故满足LSP

### 【逆变与协变】

- 逆变与协变综述：如果A、B表示类型， $f(\cdot)$ 表示类型转换， $\leq$ 表示继承关系（比如， $A \leq B$ 表示A是由B派生出来的子类）：
  - $f(\cdot)$ 是逆变（contravariant）的，当 $A \leq B$ 时有 $f(B) \leq f(A)$ 成立；
  - $f(\cdot)$ 是协变（covariant）的，当 $A \leq B$ 时有 $f(A) \leq f(B)$ 成立；
  - $f(\cdot)$ 是不变（invariant）的，当 $A \leq B$ 时上述两个式子均不成立，即 $f(A)$ 与 $f(B)$ 相互之间没有继承关系。
- 协变（Co-variance）：
  - 父类型→子类型：越来越具体(specific)。
  - 在LSP中，返回值和异常的类型：不变或变得更具体。
  - 栗子：

▪ See this example:

```

class T {
    Object a() { ... }
}

class S extends T {
    @Override
    String a() { ... }
}

```

▪ More specific classes may have more specific return types

```

class T {
    void b( ) throws Throwable {...}
}

class S extends T {
    @Override
    void b( ) throws IOException {...}
}

class U extends S {
    @Override
    void b( ) {...}
}

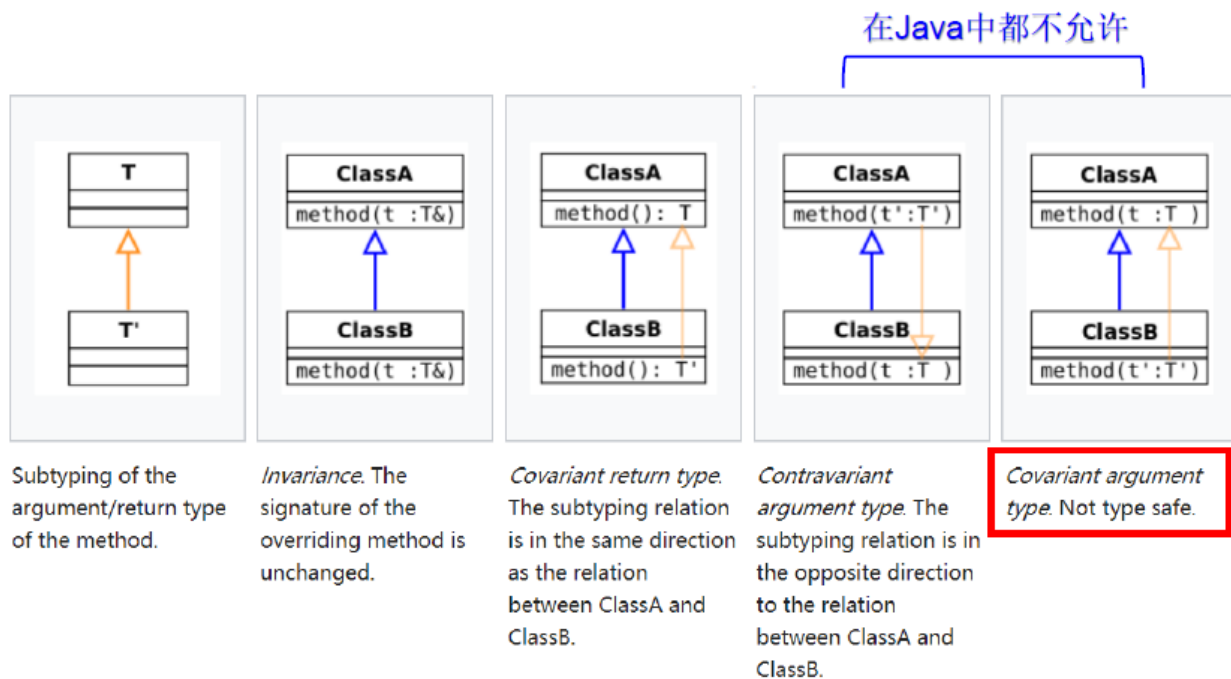
```

- 逆变(Contra-variance):
  - 父类型→子类型：越来越抽象。
  - 参数类型：要相反的变化，不变或越来越抽象。

```
class T {  
    void c( String s ) { ... }  
}  
  
class S extends T {  
    @Override  
    void c( Object s ) { ... }  
}
```

- 栗子：
- 但这在Java中是不允许的，因为它会使重载规则复杂化。

总结：



(1.子类型（属性、方法）关系；2.不变性，重写方法；3.协变，方法返回值变具体；4.逆变，方法参数变抽象；5.协变，参数变的更具体，协变不安全)


【Liskov替换原则(LSP)】 [更多参考：LSP的笔记](#)

- 里氏替换原则的主要作用就是规范继承时子类的一些书写规则。其主要目的就是保持父类方法不被覆盖。
- 含义：
  - 子类必须完全实现父类的方法
  - 子类可以有个性
  - 覆盖或实现父类的方法时输入参数可以被放大
  - 覆盖或实现父类的方法时输出结果可以被缩小
- LSP是子类型关系的一个特殊定义，称为（强）行为子类型化。在编程语言中，LSP依赖于以下限制：
  - 前置条件不能强化
  - 后置条件不能弱化
  - 不变量要保持或增强
  - 子类型方法参数：逆变
  - 子类型方法的返回值：协变
  - 异常类型：协变

## ## 各种应用中的LSP

### 【数组是协变的】


- 数组是协变的：一个数组`T[]`，可能包含了`T`类型的实例或者`T`的任何子类型的实例
- 即子类型的数组可以赋予父类型的数组进行使用，但数组的类型实际为子类型。
- 下面报错的原因是`myNumber`指向的还是一个`Integer[]` 而不是`Number[]`



```
Number[] numbers = new Number[2];
numbers[0] = new Integer(10);
numbers[1] = new Double(3.14);

Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;

myNumber[0] = 3.14; //run-time error!
```



### 【泛型中的LSP】

- Java中泛型是不变的,但可以通过通配符`"?"`实现协变和逆变：
  - `<? extends>`实现了泛型的协变：
    - `List<? extends Number> list = new ArrayList<Integer>();`
  - `<? super>`实现了泛型的逆变：
    - `List<? super Number> list = new ArrayList<Object>();`
- 由于泛型的协变只能规定类的上界，逆变只能规定下界，使用时需要遵循PECS（producer--extends, consumer-super）：
  - 要从泛型类取数据时，用`extends`；
  - 要往泛型类写数据时，用`super`；
  - 既要取又要写，就不用通配符（即`extends`与`super`都不用）。
- 泛型是类型不变的（泛型不是协变的）。举例来说
  - `ArrayList<String>` 是`List<String>` 的子类型
  - `List<String>` 不是`List<Object>` 的子类型
- 在代码的编译完成之后，泛型的类型信息就会被编译器擦除。因此，这些类型信息并不能在运行阶段时被获得。这一过程称之为类型擦除（type erasure）。
- 类型擦除的详细定义：如果类型参数没有限制，则用它

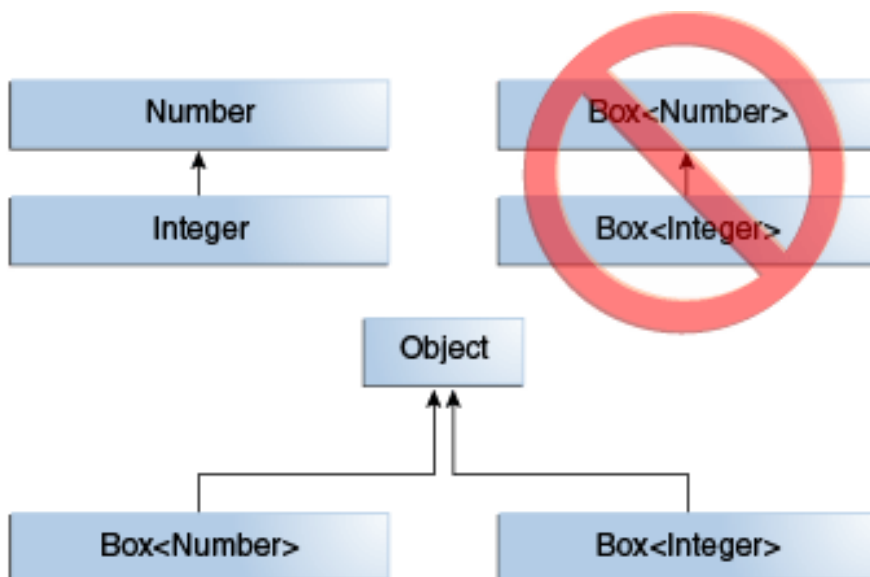
们的边界或Object来替换泛型类型中的所有类型参数。因此，产生的字节码只包含普通的类、接口和方法。

- 类型擦除的结果：<T>被擦除 T变成了Object

```
public class Node<T> {  
  
    private T data;  
    private Node<T> next;  
  
    public Node(T data, Node<T> next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public T getData() { return data; }  
    // ...  
}
```

```
public class Node {  
  
    private Object data;  
    private Node next;  
  
    public Node(Object data,  
                Node next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public Object getData() {  
        return data; }  
    // ...  
}
```

- Integer是number的子类型，但Box<Integer>也不是Box<Number>的子类型
- 这对于类型系统来说是不安全的，编译器会立即拒绝它。



【为了解决类型擦除的问题-----Wildcards（通配符）】

- 无界通配符类型使用通配符（?）指定，例如List<?>，这被称为未知类型的列表。
- 在两种情况下，无界通配符是一种有用的方法：
  - 如果您正在编写可使用Object类中提供的功能实现的方法。
  - 当代码使用泛型类中不依赖于类型参数的方法时。例如，List.size 或List.clear。事实上，Class<?>经常被使用，因为Class<T>中的大多数方法不依赖于T。

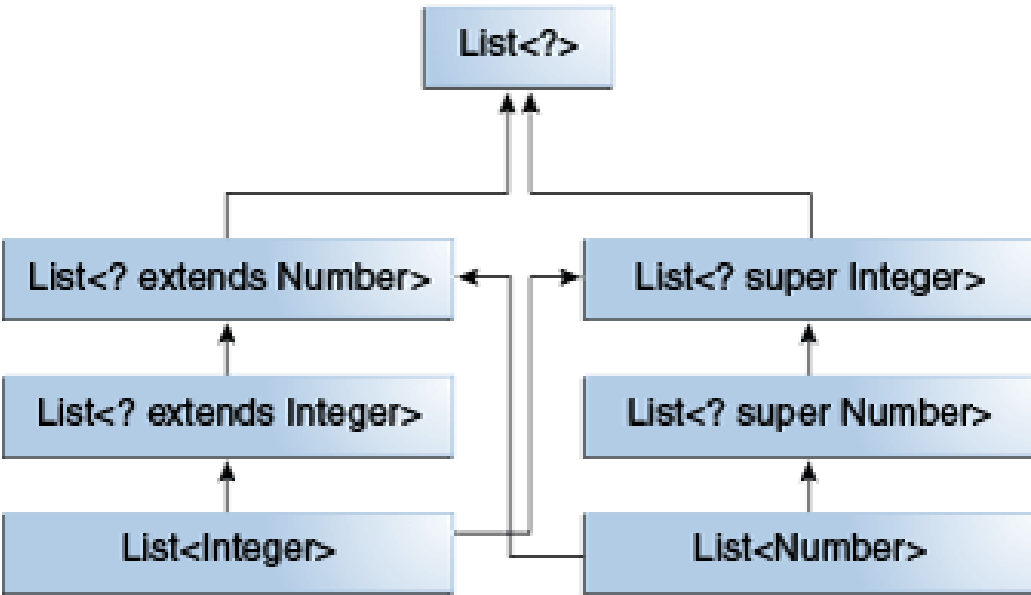
栗子：

```
public static void printList(List<Object> list)
{
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

printList的目标是打印任何类型的列表，但它无法实现该目标，它仅打印Object 实例列表; 它不能打印List <Integer>, List <String>, List <Double> 等，因为它们不是List <Object> 的子类型。

- 要编写通用的printList 方法，请使用List<?>
- 低边界通配符<? super A> e.g. List<? super Integer> List<Number>
- 上边界通配符<? extends A> e.g. List<? extends Number> List<Integer>

```
1 public static void printList(List<?> list) {
2     for (Object elem: list)
3 System.out.println();
4 }
5
6 list<Integer> li = Arrays.asList(1, 2, 3);
7 List<String> ls = Arrays.asList("one", "two",
"three");
8 printList(li);
9 printList(ls);
```



## ## 委派与组合

【Comparator 和 Comparable（比较器与可比较的）】

我们先看一个比较排序的例子：

## ■ Version A:

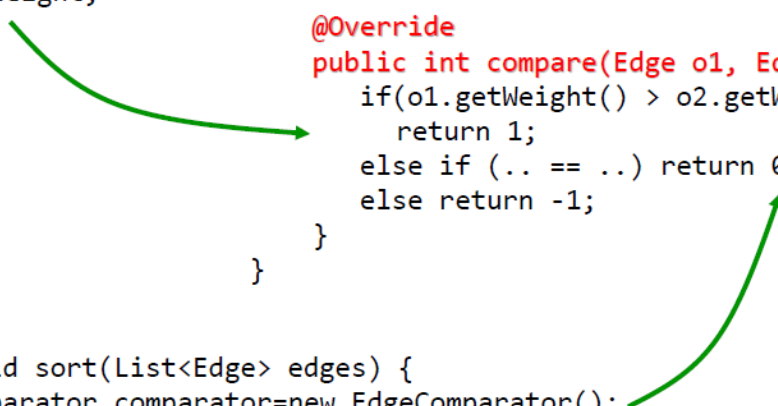
```
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
    ...
}
```

## ■ Version B:

```
interface Comparator {
    boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
```

- 引入 `int compare(T o1, T o2)`：用于比较两个变量的大小。
- 如果你的ADT需要比较大小，或者要放入Collections或Arrays进行排序，可实现Comparator接口并override `compare()`函数。下面为具体例子：



```
public class Edge {
    Vertex s, t;
    double weight;
    ...
}

public class EdgeComparator
    implements Comparator<Edge>{

    @Override
    public int compare(Edge o1, Edge o2) {
        if(o1.getWeight() > o2.getWeight())
            return 1;
        else if (.. == ..) return 0;
        else return -1;
    }
}

public void sort(List<Edge> edges) {
    EdgeComparator comparator=new EdgeComparator();
    Collections.sort(steps, comparator);
}
```

另一种方法：让你的ADT实现Comparable 接口，然后override `compareTo()` 方法。与使用Comparator 的区别：不需要构建新的Comparator 类，比较代码放在ADT内部。下面为具体例子。



```

public class Edge implements Comparable<Edge> {
    Vertex s, t;
    double weight;
    ...

    public int compareTo(Edge o) {
        if(this.getWeight() > o.getWeight())
            return 1;
        else if (.. == ..) return 0;
        else return -1;
    }
}

```

### 【委派(Delegation)】

- 委派/委托：一个对象请求另一个对象的功能。
- 委派是复用的一种常见形式。
- 分为显性委派：将发送对象传递给接收对象；
- 以及隐性委派：由语言的成员查找规则。
- 下面是一个栗子：可以看到，想在B中调用A，需要先委派一个A。

```

class A {
    void foo() {
        this.bar();
    }
    void bar() {
        print("a.bar");
    }
}

class B {
    private A a; // delegation link

    public B(A a) {
        this.a = a;
    }

    void foo() {
        a.foo(); // call foo() on the a-instance
    }

    void bar() {
        print("b.bar");
    }
}

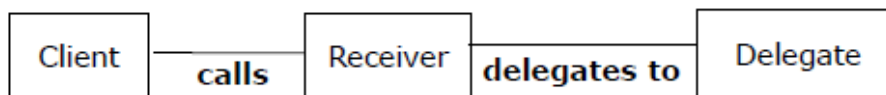
```

```

a = new A();
b = new B(a); // establish delegation between two objects

```

- 委派设计模式：是一种用来实现委派的软件设计模式；
- 委派依赖于动态绑定，因为它要求给定的方法调用可以在运行时调用不同的代码段；
- 委派的过程如下：



Receiver对象将操作委托给Delegate对象，同时Receiver对象确保客户端不会滥用委托对象；

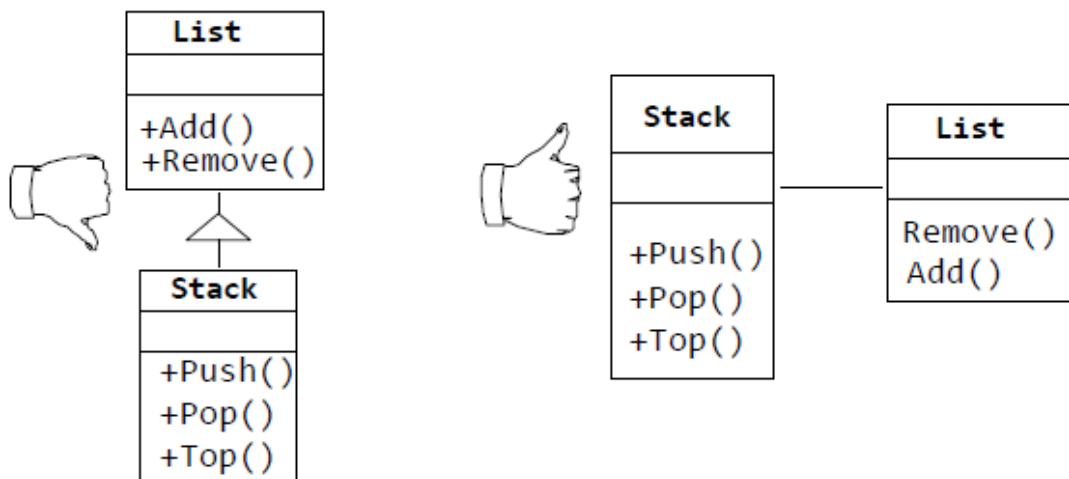
- 例子二：使用委派可以继承委派类的功能，这里list是被

委派的对象，List的方法add和remove方法都可以通  
过list在LoggingList中被调用。

```
public class LoggingList<E> implements List<E> {  
    private final List<E> list;  
    public LoggingList<E>(List<E> list) { this.list = list; }  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return list.add(e);  
    }  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return list.remove(index);  
    }  
}
```

#### 【委派与继承】

- 继承：通过新操作扩展基类或覆盖操作。
- 委托：捕获操作并将其发送给另一个对象。
- 许多设计模式使用继承和委派的组合。



- **Problem:** 如果子类只需要复用父类中的一小部分方法，
- **Solution:** 可以不需要使用继承，而是通过委派机制来实现。
- 本质上，一个类不需要继承另一个类的全部方法，通过委托机制调用部分方法。

```
class RealPrinter {
    void print() {
        System.out.println("Printing Data");
    }
}
class Printer extends RealPrinter {
    void print(){
        super.print();
    }
}

Printer printer = new Printer();
printer.print();
```

## Inheritance

## Delegation

```
class RealPrinter {
    void print() {
        System.out.println("The Delegate");
    }
}
class Printer {
    RealPrinter p = new RealPrinter();

    void print() {
        p.print();
    }
}
Printer printer = new Printer();
printer.print();
```

### 【复合继承原则 (CRP)】

- 复合复用原则(CRP): 类应当通过它们之间的组合 (通过包含其它类的实例来实现期望的功能) 达到多态表现和代码复用, 而不仅仅是从基础类或父类继承。
- 我们可以将组合 (Composition) 理解为 (has a) 而继承理解为(is a);
- 委派可以看做Object层面的复用机制, 而继承可以看做是类的层面;
- 下面我们看一个关于CRP的栗子:

- Employee 类具有计算员工年度奖金的方法:

```
class Employee {
    Money computeBonus() {... // default
    computation}

    ...
}
```

- Employee 的不同子

类: Manager , Programmer , Secretary 等可能希望重写此方法以反映某些类型的员工比其他员工获得更慷慨的奖金这一事实:

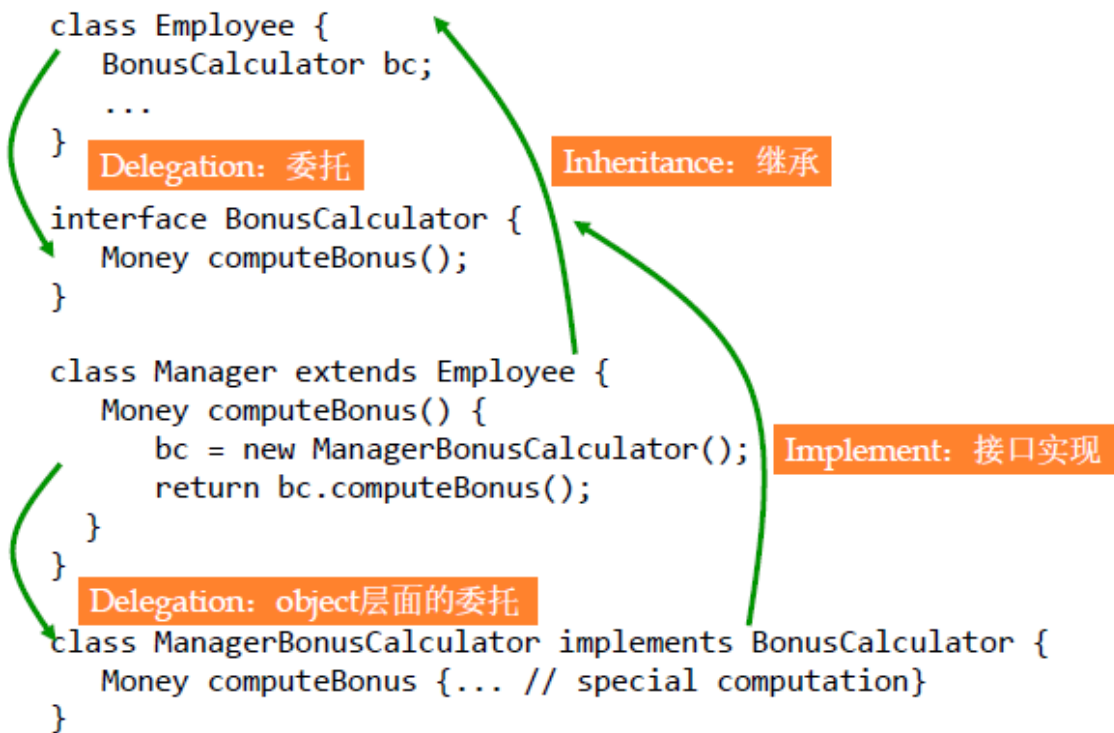
```
class Manager extends Employee {
    @Override Money computeBonus() {...
    // special computation}

    ...
}
```

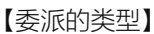
- 这个解决方案有几个问题。所有Manager 对象获得相同的奖金。 如果我们想改变管理者之间的奖金计算怎么办? 引入Manager 的特殊子类?

```
class SeniorManager extends Manager {
    @Override Money computeBonus() {...
    // more special computation
    ...
}
```

- 如果我们想改变特定员工的奖金计算会怎样？例如，如果我们想要将史密斯从Manager 推广到SeniorManager，该怎么办？
- 如果我们决定让所有Manager 获得与Programmer 相同的奖金呢？我们是否应该将Programmer 中的计算算法复制并粘贴到Manager 中？
- 核心问题：每个Employee对象的奖金计算方法都不同；如果能在object层面实现一定比class层面灵活很多。
- CRP的解决方法：

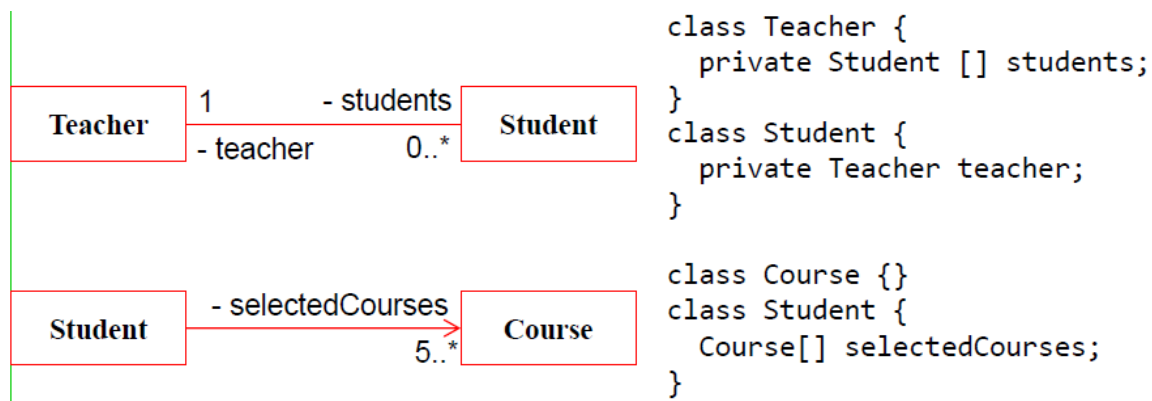


- 只需针对不同子类的对象，委派能够计算该子类的奖金的方法的BonusCalculator。这样一来就不需要在子类继承的时候进行重写。
- 【总结】组合来代替继承的更普遍实现：
  - 用接口来实现系统的最基础行为
  - 接口之间用extends来实现系统功能的扩展（接口组合）
  - 类implements 组合接口

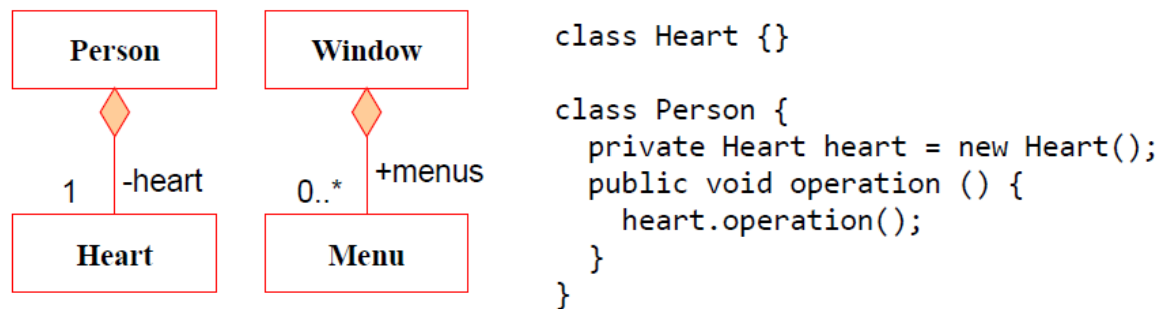


- 
- ```
classDiagram
    class CourseSchedule {
        +add(c:Course)
        +remove(c:Course)
    }
    class Course
    class Iterator
    CourseSchedule ..> Course
    CourseSchedule ..> Iterator
    Iterator --> CourseSchedule
    Note over CourseSchedule, Iterator: <<friend>>
```

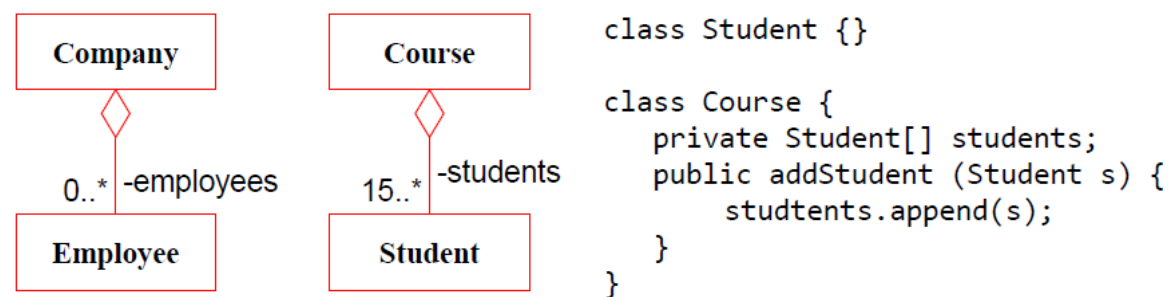
- 永久性委派 (Association) : 类之中有其它类的具体实例来作为一个变量 (has a)



- 更强的委派，组合（Composition）：更强的委派。将一些简单的对象组合成一个更为复杂的对象。（is part of）



- 聚合（Aggregation）：对象是在类的外部生成的，然后作为一个参数传入到类的内部构造器。（has a）



### 【组合与聚合】

在组合中，当拥有的对象被破坏时，被包含的对象也被破坏。在聚合中，这不一定是真的。以生活中的事物为例：大学拥有多个部门，每个部门都有一批教授。如果大学关闭，部门将不复存在，但这些部门的教授将继续存在。一位教授可以在一个以上的部门工作，但一个部门不能成为多个大学的一部分。大学与部门之间的关系即为组合，而部分与教授之间的关系为聚合。

## ## 设计可复用库与框架

之所以library和framework被称为系统层面的复用，是因为它们不仅定义了1个可复用的接口/类，而是将某个完整系统中的所有可复用的接口/类都实现出来，并且定义了这些类之间的交互关系、调用关系，从而形成了系统整体的“架构”。

- 相应术语：
  - API（Application Programming Interface）：库或

框架的接口

- **Client**（客户端）：使用API的代码
- **Plugin**（插件）：客户端定制框架的代码
- **Extension Point**：框架内预留的“空白”，开发者开发出符合接口要求的代码(即plugin)，框架可调用，从而相当于开发者扩展了框架的功能
- **Protocol**（协议）：API与客户端之间预期的交互序列。
- **Callback**（反馈）：框架将调用的插件方法来访问定制的功能。
- **Lifecycle method**：根据协议和插件的状态，按顺序调用的回调方法。

### 【API和库】

- **API**是程序员最重要的资产和“荣耀”，吸引外部用户，提高声誉。
- 建议：始终以开发**API**的标准面对任何开发任务；面向“复用”编程而不是面向“应用”编程。
- 难度：要有足够良好的设计，一旦发布就无法再自由改变。
- 编写一个**API**需要考虑以下方面：
  - **API**应该做一件事，且做得很好
  - **API**应该尽可能小，但不能太小
  - **Implementation**不应该影响**API**
  - 记录文档很重要
  - 考虑性能后果
  - **API**必须与平台和平共存
  - 类的设计：尽量减少可变性，遵循**LSP**原则
  - 方法的设计：不要让客户做任何模块可以做的事情，及时报错

### 【框架】（内容参考[白盒框架与黑盒框架](#)）

- 框架（**Framework**）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法；另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义。
- 为了增加代码的复用性，可以使用委派和继承机制。同时，在使用这两种机制增加代码复用的过程中，我们也相应地在不同的类之间增加了关系（委派或继承关系）。而对于一个项目而言，各个不同类之间的依赖关系就可以看做为一个框架。一个大规模的项目可能由许多不同的框架组合而成。
- 框架与设计模式：
  - 框架、设计模式这两个概念总容易被混淆，其实它们之间还是有区别的。构件通常是代码重用，而设计模式是设计重用，框架则介于两者之间，部分代码重用，部分设计重用，有时分析也可重用。在软件生产

中有三种级别的重用：内部重用，即在同一应用中能公共使用的抽象块；代码重用，即将通用模块组合成库或工具集，以便在多个应用和领域都能使用；应用框架的重用，即为专用领域提供通用的或现成的基础结构，以获得最高级别的重用性。

- 框架与设计模式虽然相似，但却有着根本的不同。设计模式是对在某种环境中反复出现的问题以及解决该问题的方案的描述，它比框架更抽象；框架可以用代码表示，也能直接执行或复用，而对模式而言只有实例才能用代码表示；设计模式是比框架更小的元素，一个框架中往往含有一个或多个设计模式，框架总是针对某一特定应用领域，但同一模式却可适用于各种应用。可以说，框架是软件，而设计模式是软件的知识。
- 框架分为白盒框架和黑盒框架。
  - 白盒框架：
    - 白盒框架是基于面向对象的继承机制。之所以说是白盒框架，是因为在这种框架中，父类的方法对子类而言是可见的。子类可以通过继承或重写父类的方法来实现更具体的方法。
    - 虽然层次结构比较清晰，但是这种方式也有其局限性，父类中的方法子类一定拥有，要么继承，要么重写，不可能存在子类中不存在的方法而在父类中存在。
    - 软件构造课程中有关白盒框架的例子：



```
public abstract class PrintOnScreen {
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame,
textToShow());
        frame.dispose();
    }
    protected abstract String textToShow();
}
```




```
public class MyApplication extends PrintOnScreen
{
    @Override protected String textToShow() {
        return "printing this text on " +
"screen using PrintOnScreen " + "white Box
Framework";
    }
}
```

- 通过子类化和重写方法进行扩展（使用继承）；





- 通用设计模式：模板方法；
- 子类具有主要方法但对框架进行控制。
- 允许扩展每一个非私有方法
- 需要理解父类的实现
- 一次只进行一次扩展
- 通常被认为是开发者框架
- 黑盒框架：
  - 黑盒框架时基于委派组合方式，是不同对象之间的组合。之所以是黑盒，是因为不用去管对象中的方法是如何实现的，只需关心对象上拥有的方法。
  - 这种方式较白盒框架更为灵活，因为可以在运行时动态地传入不同对象，实现不同对象间的动态组合；而继承机制在静态编译时就已经确定好。
  - 黑盒框架与白盒框架之间可以相互转换，具体例子可以看一下，软件构造课程中有关黑盒框架的例子，更改上面的白盒框架为黑盒框架：


```
public interface TextToShow {  
    String text();  
}
```



```
public class MyTextToShow implements TextToShow  
{  
    @Override  
    public String text() {  
        return "Printing";  
    }  
}
```



```
public final class PrintOnScreen {  
    TextToShow textToShow;  
    public PrintOnScreen(TextToShow tx) {  
        this.textToShow = tx;  
    }  
    public void print() {  
        JFrame frame = new JFrame();  
        JOptionPane.showMessageDialog(frame,  
textToShow.text());  
        frame.dispose();  
    }  
}
```



- 通过实现插件接口进行扩展（使用组合/委派）；
- 常用设计模式：**Strategy, Observer**；
- 插件加载机制加载插件并对框架进行控制。
- 允许在接口中对**public**方法扩展
- 只需要理解接口
- 通常提供更多的模块
- 通常被认为是终端用户框架，平台