

# 设计模式

## Design Pattern

主讲：辜希武

华中科技大学计算机学院IDC实验室

<http://idc.hust.edu.cn/>

Email: [guxiwu@mail.hust.edu.cn](mailto:guxiwu@mail.hust.edu.cn)

# 设计模式 (Design Pattern)

## 什么是设计模式

在面向对象程序设计 (OOP) 过程中, 我们经常会遇到很多重复出现的问题, 总结解决这些问题的成功经验和最佳实践便形成了设计模式 (Design Pattern) 。

其核心思想是将可重用的解决方案总结出来, 并分门别类。从而指导设计, 减少代码重复和优化体系结构。

# 设计模式 (Design Pattern)

## 采用设计模式的好处

- 重用，避免代码重复冗余
- 优化体系结构
- 提升系统的可维护性和弹性
- 代码更加容易测试，利于测试驱动
- 为性能优化提供便利
- 使软件质量更加有保证
- 增强代码可读性，便于团队交流
- 有助于整体提升团队水平

# 设计模式 (Design Pattern)

## 设计模式与UML

设计模式是OOP的方法论，其内容就是如何设计对象的结构及其相互间的协作关系。因此需要一种直观的模型将上述内容清晰地表示出来。

统一建模语言 (UML) 是OOP的建模语言，其核心就是把软件的设计思想通过建模的方法表达出来。故非常适合于表达设计模式。同时UML已经被广泛用于软件设计，这也推动了设计模式的应用。

# 模式的基本要素

- ✓ **模式名称**(Pattern Name)
- ✓ **问题**(Problem): 描述应该在何时使用模式。解释了设计问题和问题存在的前因后果,可能还描述模式必须满足的先决条件
- ✓ **解决方案**(Solution): 描述了设计的组成成分、相互关系及各自的职责和协作方式。模式就像一个模板,可应用于多种场合,所以解决方案并不描述一个具体的设计或实现,而是提供设计问题的抽象描述和解决问题所采用的元素组合(类和对象)
- ✓ **效果**(consequences): 描述模式的应用效果及使用模式应权衡的问题

# 如何描述设计模式

## ◆模式名和分类

◆**意图**：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？

◆**动机**：说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景

◆**适用性**：什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？如何识别这些情况？

◆**结构**：采用对象建模技术对模式中的类进行图形描述

## 描述设计模式（续）

- ◆ **参与者**：指设计模式中的类 和/或 对象以及它们各自的职责
- ◆ **协作**：模式的参与者如何协作以实现其职责
- ◆ **效果**：模式如何支持其目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？
- ◆ **实现**：实现模式时需了解的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题
- ◆ **代码示例**：用来说明怎样实现该模式的代码片段
- ◆ **相关模式**：与这个模式紧密相关的模式有哪些？其不同之处是什么？这个模式应与哪些其他模式一起使用？

# 设计模式的原则

- ◆ "开-闭"原则(Open Closed Principal)
- ◆ 单一职责原则
- ◆ 里氏代换原则
- ◆ 依赖倒置原则
- ◆ 接口隔离原则

- ◆ 设计模式就是实现了上述原则，从而达到代码复用、增加可维护性的目的



# 开闭原则 (OCP)

- ◆ **定义：** 软件对扩展是开放的，对修改是关闭的。开发一个软件时，应可以对其进行功能扩展(开放)，在进行扩展的时候，不需要对原来的程序进行修改(关闭)
- ◆ **好处：** 在软件可用性上非常灵活。可以在软件完成后对软件进行扩展，加入新的功能。这样，软件就可通过不断的增加新模块满足不断变化的新需求
  - ◆ 由于不修改软件原来的模块，不用担心软件的稳定性

# 开闭原则 (OCP)

## ◆ 实现的主要原则

- ◆ **抽象原则**：把系统的所有可能的行为抽象成一个底层；  
由于可从抽象层导出一个或多个具体类来改变系统行为，  
因此对于可变部分，系统设计对扩展是开放的
- ◆ **可变性封装原则**：对系统所有可能发生变化的部分进行评估和分类，每一个可变的因素都单独进行封装

# 单一职责原则 (SRP)

- ◆ 一个类一个职责
- ◆ 当类具有多职责时,应把多余职责分离出去, 分别创建一些类来完成每一个职责
- ◆ 每一个职责都是一个变化的轴线, 当需求变化时会反映为类的职责的变化

举例

```
interface Modem{  
    public void dial(String pno);  
    public void hangup();  
    public send(char c);  
    public char recv();  
}
```

Modem类有两个职责: 连接管理和数据通信, 应将它们分离

# IsKov替换原则 (LSP)

◆ **定义：** “继承必须确保超类所拥有的性质在子类中仍然成立”，当一个子类的实例能够替换任何其超类的实例时，它们之间才具有is-A关系

◆ 里氏替换原则是继承复用的基石，只有当派生类可以替换掉其基类，而软件功能不受影响时，基类才能真正被复用，派生类也才能够在基类的基础上增加新的行为

◆ **LSP本质：** 在同一个继承体系中的对象应该有共同的行为特征

◆ 例子：企鹅是鸟吗？

生物学：企鹅属于鸟类

LSP原则：企鹅不属于鸟类，因为企鹅不会“飞”

◆ 违反LSP的后果：有可能需要修改客户代码

## listKov替换原则 (LSP)

◆ **定义1**: 如果对每一个类型为 $T_1$ 的对象 $o_1$ , 都有类型为 $T_2$ 的对象 $o_2$ , 使得以 $T_1$ 定义的所有程序 $P$ 在所有的对象 $o_1$ 都代换成 $o_2$ 时, 程序 $P$ 的行为没有发生变化, 那么类型 $T_2$ 是类型 $T_1$ 的子类型。

◆ **定义2**: 所有引用基类的地方必须能透明地使用其子类的对象。

◆ **问题由来**: 有一功能 $P_1$ , 由类 $A$ 完成。现需要将功能 $P_1$ 进行扩展, 扩展后的功能为 $P$ , 其中 $P$ 由原有功能 $P_1$ 与新功能 $P_2$ 组成。新功能 $P$ 由类 $A$ 的子类 $B$ 来完成, 则子类 $B$ 在完成新功能 $P_2$ 的同时, 有可能会导导致原有功能 $P_1$ 发生故障。

◆ **解决方案**: 当使用继承时, 遵循里氏替换原则。类 $B$ 继承类 $A$ 时, 除添加新的方法完成新增功能 $P_2$ 外, 尽量不要重写父类 $A$ 的方法, 也尽量不要重载父类 $A$ 的方法。

# listKov替换原则 (LSP)

```
class A{  
    public int func1(int a, int b){  
        return a-b;  
    }  
}  
  
public class Client{  
    public static void main(String[] args){  
        A a = new A();  
        System.out.println("100-50="+a.func1(100, 50));  
        System.out.println("100-80="+a.func1(100, 80));  
    }  
}
```

# listKov替换原则 (LSP)

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

两数相减。

两数相加，然后再加100。

```
class B extends A{  
    public int func1(int a, int b){  
        return a+b;  
    }  
    public int func2(int a, int b){  
        return func1(a,b)+100;  
    }  
}
```

# listKov替换原则 (LSP)

```
public class Client{  
    public static void main(String[] args){  
        A b = new B();  
        System.out.println("100-50="+b.func1(100, 50));  
        System.out.println("100-80="+b.func1(100, 80));  
        System.out.println("100+20+100="+b.func2(100, 20));  
    }  
}
```

我们发现原本运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。



## listKov替换原则 (LSP)

◆ 在本例中，引用基类A完成的功能，换成子类B之后，发生了异常。

可复用性

◆ 在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。

◆ 如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。

◆ 里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。

# 依赖倒置原则 (DIP)

◆ 定义：**高层模块不应依赖低层模块，二者都应该依赖于抽象**

◆ 高层模块只应该包含重要的业务模型和策略选择，低层模块则是不同业务和策略的实现

◆ 高层抽象不依赖高层和低层模块的具体实现，最多只依赖于低层的抽象

◆ 低层抽象和实现也只依赖于抽象

◆ 辅助原则

◆ 任何变量的类型都不应该是具体类

◆ 任何类都不应该从具体类派生

◆ 任何方法都不应覆盖其任何基类中已经实现了的方法

# 依赖倒置原则 (DIP) 举例

## 应用程序

存储的需求

`saveToFloppy()`

```
....  
void saveToFloppy(){  
    ....  
}
```

直接在高层的应用程序  
调用低层模块的API时,  
导致高层模块对低层模块  
的依赖

## 依赖倒置原则 (DIP) 举例

```
public class Business{  
    private FloppyWriter writer = new FloppyWriter ();  
    ...  
    public void save(){  
        writer.saveToFloppy();  
    }  
}
```

高层业务类 Business的实现依赖于低层FloppyWriter 类，如果想把存储介质改为USB盘，则必须修改Business类，使得无法重用Business类

违反了依赖倒置原则！

# 依赖倒置原则 (DIP) 举例

利用接口抽象，可以改进

```
public interface IDeviceWriter{  
    public void saveToDevice();  
}  
  
public class Business{  
    private IDeviceWriter writer;  
    ...  
    public void setDeviceWriter(IDeviceWriter writer){  
        this.writer = writer;  
    }  
    public void save(){  
        writer.saveToDevice();  
    }  
}
```

高层 (Business类) 的  
实现依赖于抽象 (接口)

## 依赖倒置原则 (DIP) 举例

在这样的设计下，Business就是可重用的。如果今天有存储至Floppy或USB的需求，只要针对这两种存储需求分别实现IDeviceWriter接口即可。

```
public class FloppyWriter implements IDeviceWriter{
    public void saveToDevice{
        ... //存储至软盘的代码
    }
}

public class USBWriter implements IDeviceWriter{
    public void saveToDevice{
        ...//存储至USB的代码
    }
}
```

低层类的实现也  
依赖于抽象（接口）

## 依赖倒置原则（DIP）举例

如果应用程序需要USB存储，则编写如下配置程序：

```
Business business= new Business();  
business.setDeviceWriter(new USBWriter()); //对象注入  
business.save();
```

可以看到，无论低层的存储实现如何变动，对于Business类来说都无需修改。

如果采用对象工厂模式来动态创建实现存储功能的低层对象，连上面三行代码都不用写，只需要编写一个XML的配置文件，在配置文件里指定要注入到Business里面的低层存储对象即可。

这就是著名的Spring框架的一个重要功能：**对象注入，或者叫依赖注入（Dependency Injection）**

## 接口隔离原则 (ISP)

- ◆ 多个和客户相关的接口要好于一个通用接口
- ◆ 如果一个类有几个使用者，与其让这个类载入所有使用者需要使用的所有方法，还不如为每个使用者创建一个特定接口，并让该类分别实现这些接口



```
public class SeaPlane{  
    //与飞行有关的方法  
    void takeOff() { ... }  
    void fly() { ... }  
    void land() {...}  
  
    //与海上航行有关的方法  
    void dock() { ... }  
    void cruise() {...}  
}
```

```
//客户代码1  
//只使用飞行功能
```

```
void f1(SeaPlane o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}  
f1(new SeaPlane());
```

```
//客户代码2  
//只使用海上航行功能
```

```
void f2(SeaPlane o){  
    o.dock();  
    o.cruise();  
}  
f2(new SeaPlane());
```

类SeaPlane封装了二类不同的功能。有二段客户代码分别只需要使用其中一类功能。但不得不声明SeaPlane类型的对象引用作为方法参数，导致客户代码被绑定到SeaPlane类型。

违反了一个原则：基于接口编程，不要基于类编程

```
public interface Flyer
//与飞行有关的方法
void takeOff();
void fly();
void land();
} //声明Flyer接口
```

```
public interface Sailer
//与海上航行有关的方法
void dock();
void cruise();
} //声明Sailer接口
```

```
public class SeaPlane implements Flyer,
Sailer{
//实现与飞行有关的方法
void takeOff() { ... }
void fly() { ... }
void land() {...}

//实现与海上航行有关的方法
void dock() { ... }
void cruise() {...}
}
```

现在首先声明二个分离的功能接口Flyer和Sailer

再用SeaPlane实现这二个接口，这样SeaPlane同时具有了二类不同的行为特征

```
//客户代码1  
//只使用飞行功能
```

```
void f1(Flyer o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}
```

```
f1(new SeaPlane());
```

```
//客户代码2  
//只使用海上航行功能
```

```
void f2(Sailer o){  
    o.dock();  
    o.cruise();  
}
```

```
f2(new SeaPlane());
```

接口类型的引用变量可以直接指向实现该接口的对象而不用强制类型转换！！实参传递给形参时，相当于  
`Flyer o = new SeaPlane();`

当接口类型的引用变量指向不同的实现了接口的对象时，会表现出多态性

现在通过接口，将二类功能分离。同时方法的参数类型是接口，只要接口定义不变，接口的实现类再怎么修改，方法f1和f2都不需要修改。更重要的是，当传递新的接口实现类对象给方法时，方法会自动具有新的行为。这就是接口分离和基于接口编程的好处！

# 设计模式的类型

- ◆在设计模式经典著作《GOF95》中，设计模式从应用的角度被分为三个大的类型

  - ◆创建型模式/结构型模式/行为型模式

- ◆根据模式的范围分，模式用于类还是用于对象

  - ◆**类模式**：处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来了

  - ◆**对象模式**：处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性

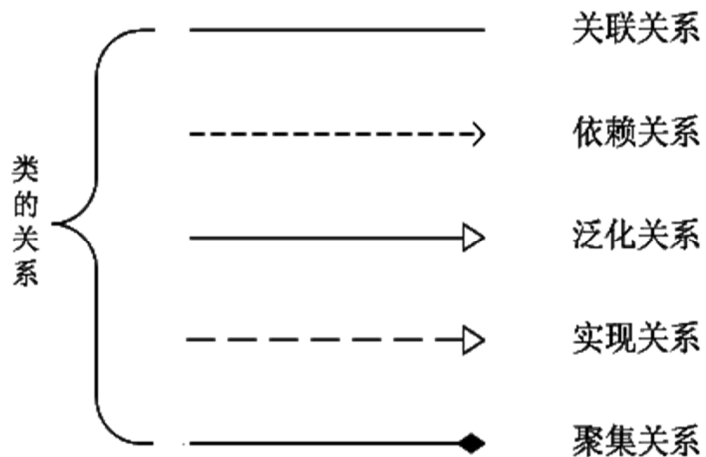
- ◆从某种意义上来说，几乎所有模式都使用继承机制，所以“类模式”只指那些集中于处理类间关系的模式，而大部分模式都属于对象模式的范畴

# Object Modeling Technique (OMT)

## 类的关系总结

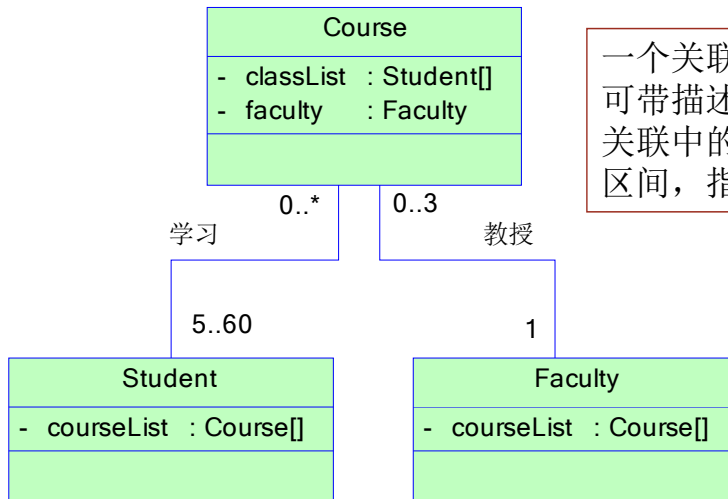
- 类与类之间的关系
  - 关联关系
  - 聚合关系
  - 组合关系
  - 依赖关系
  - 继承关系
- 类与接口之间的关系
  - 实现关系

# 类的关系的UML表示



# 关联关系

- 关联关系(association)是一种通用的二元关系，例如学生(Student)选学课程(Course)，教师(Faculty)教授课程(Course)，这些联系可以在UML中表示。



一个关联可以用两个类之间的实线表示。  
可带描述关系的标签。  
关联中的每个类可以指定一个数目或数字区间，指出这个关系涉及多少个对象

# 关联关系的实现

- 在Java代码中，关联关系可以用数据域或方法来实现。对于方法，一个类中的方法包含另一个类的参数。

```
public class Student{  
    private Course[] courseList;  
    public void addCourse( Course c){ ..... }  
}
```

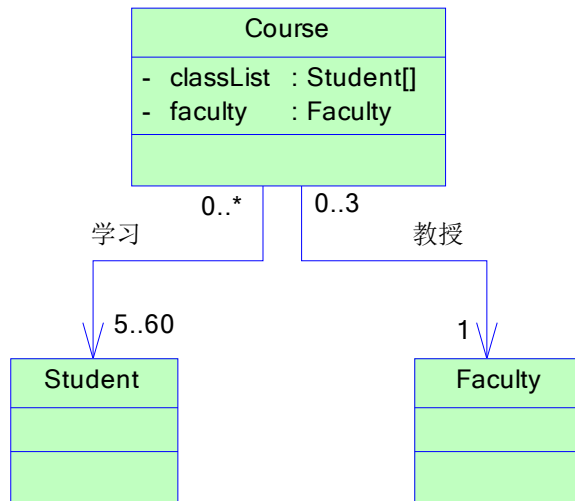
```
public class Course{  
    private Student[] classList;  
    private Faculty faculty;  
    public void addStudent( Students s){ ..... }  
    public void setFaculty( Faculty f) { ..... }  
}
```

```
public class Faculty{  
    private Course[] courseList;  
    public void addCourse( Course c){ ..... }  
}
```



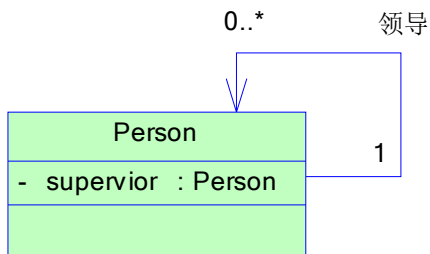
# 单向关联

- 如果学生或教师不需要知道课程的信息，可以去掉courseList域，形成单向关联。这时可将Student类和Faculty类中的数据域courseList和addCourse方法去掉。



# 自关联

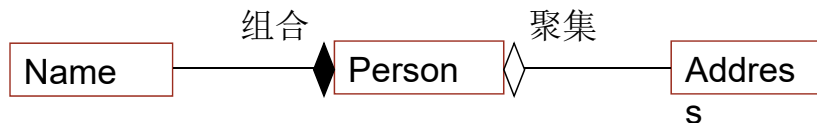
- 同一个类的对象间存在关联，称自关联。例如，一个人有一个领导。



```
public class Person{  
    private Person supervisor;  
    .....  
}
```

# 聚合和组合

- 聚合关系(aggregation)是一种特殊的关联关系，表示整体与部分之间的关系，即has-a的关系。所有者成为聚集者，从属对象称为被聚集者。在聚合关系中，一个从属者可以被多个聚集者拥有(Weak has a)。
- 组合关系(composition)是聚合关系的一种特殊形式，表示从属者强烈依赖于聚集者。一个从属者只能被一个聚集者所拥有，聚集者负责从属者的创建和销毁(Strong has a)。



一个Name对象只能为一个Person所有，但一个Address对象可以被多个Person共享

# 聚合和组合

- 聚集关系和组合关系在代码中通常表示为聚集类中的数据域，如上图中的关系可以表示为

```
public class
Name{
    ...
}
```

```
public class Person{
    private Name name;
    private Address address;
    ...
}
```

```
public class
Address{
    ...
}
```

- 对于组合关系，聚集者往往负责对从属者的创建和销毁，而聚集关系则不是。

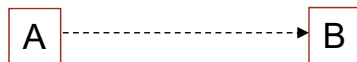
```
public class Person{
    private Name name;
    private Address address;
    public Person(Address a){
        name = new Name();
        address = a;
    }
}
```

Address对象是传递进来的一个引用，而Name对象是在Person对象创建时才创建。

当Person对象被析构时，Name对象也被析构，而Address对象可能还存在

# 依赖关系

- 依赖关系（**dependency**）指的是两个类之间一个（称为**client**）使用另一个（称为**supplier**）的关系。在UML中，从**client**画一条带箭头的虚线指向**supplier**类。
- 例如，**A**类每个功能的实现要依赖于**B**类的对象，因此**A**和**B**之间的关系可以用依赖描述。

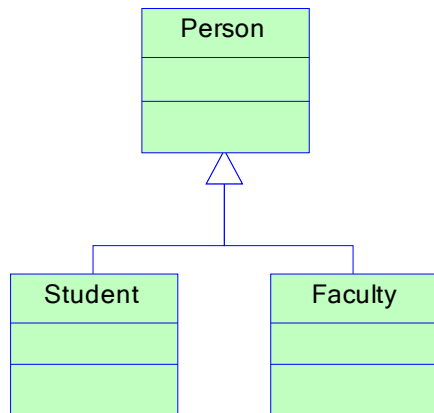


- 依赖关系在代码中通常表示为**client**类的成员函数以**supplier**类型的对象为参数

```
public class A{  
    public void fa(B o){  
        ...  
        o.fb();  
    }  
}
```

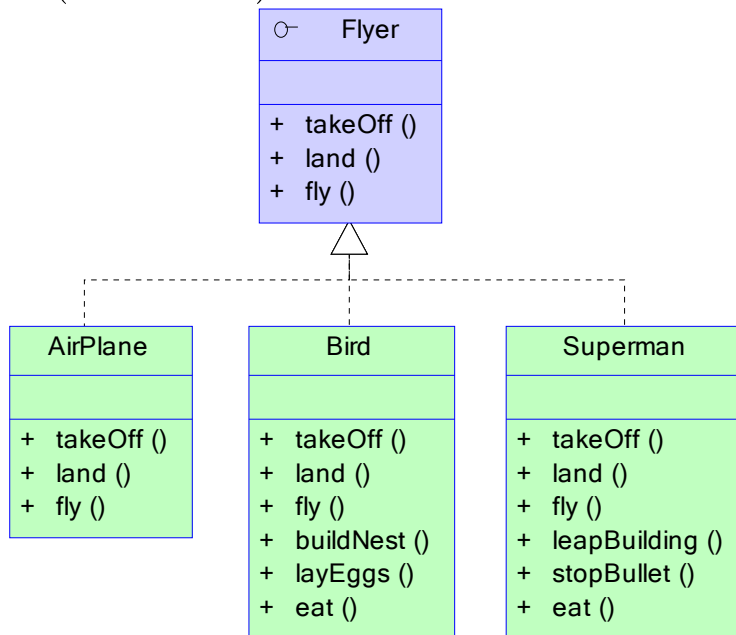
# 继承关系

- 继承关系(inheritance)表示is-a的关系。



# 实现关系

- 实现关系(realization)表示类和接口之间的关系。



# 实现关系和继承关系的区别

- 实现关系（接口）：接口通常用于描述一个类的外围能力，而不是核心特征，例如**Bird**类可以实现**Flyer**接口，而**Flyer**可以应用于其他不相关的对象。类与接口之间的实现关系是-able或者can do的关系。
- 继承关系：父类（特别是抽象类）定义了它的后代的核心特征。例如**Person**类包含了**Student**类的核心特征。派生类与父类之间的继承关系是is-a的关系。



# 继承与实现

