# java设计模式学习

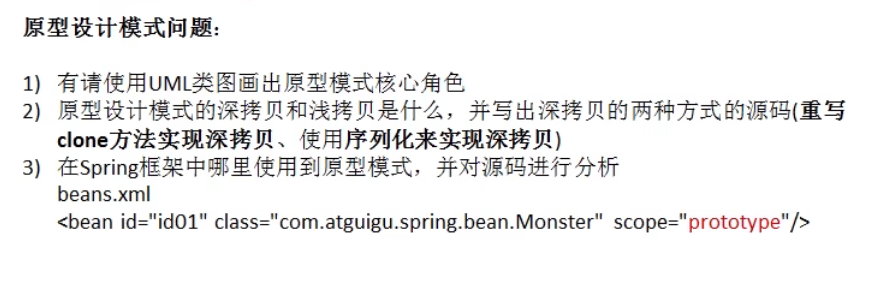
## java设计模式内容介绍

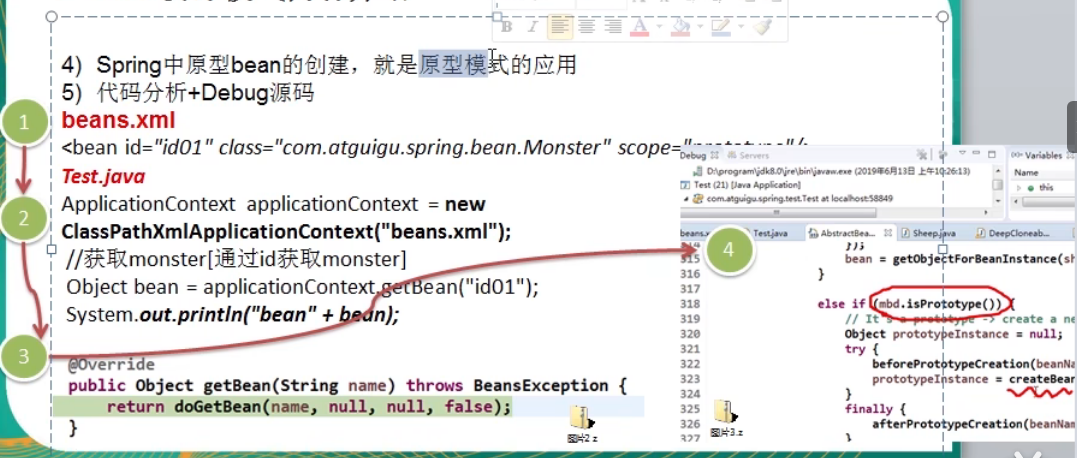
本笔记参考哔哩哔哩《尚硅谷Java设计模式（图解+框架源码剖析）》内容记录。

具体代码见项目：designPattern

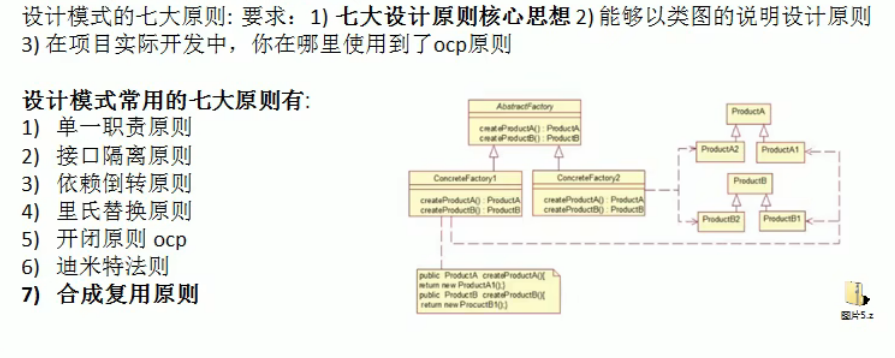
### 1.1先看几个经典的面试题

* 原型设计模式问题：

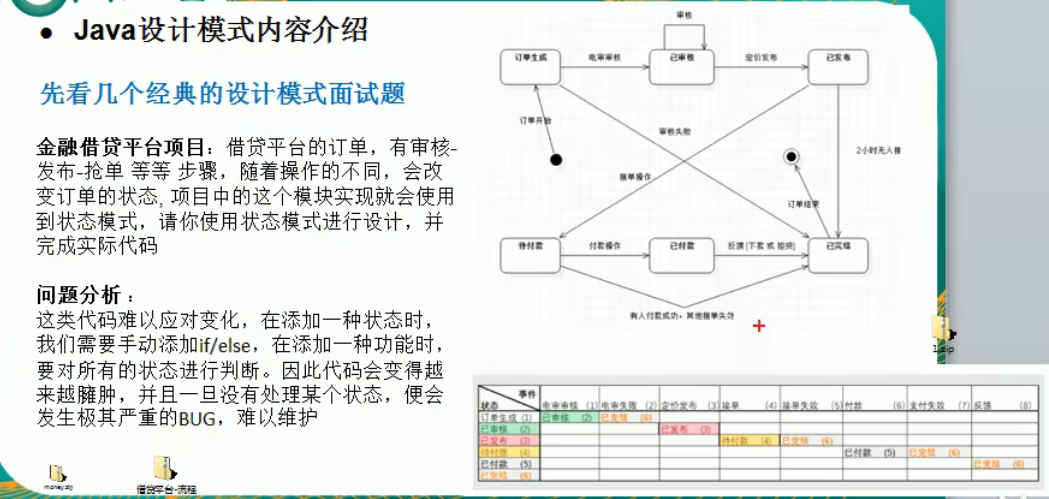




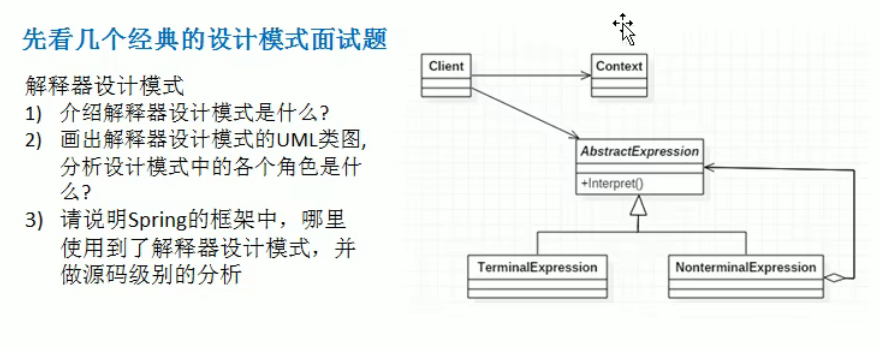
* 设计模式的七大原则：

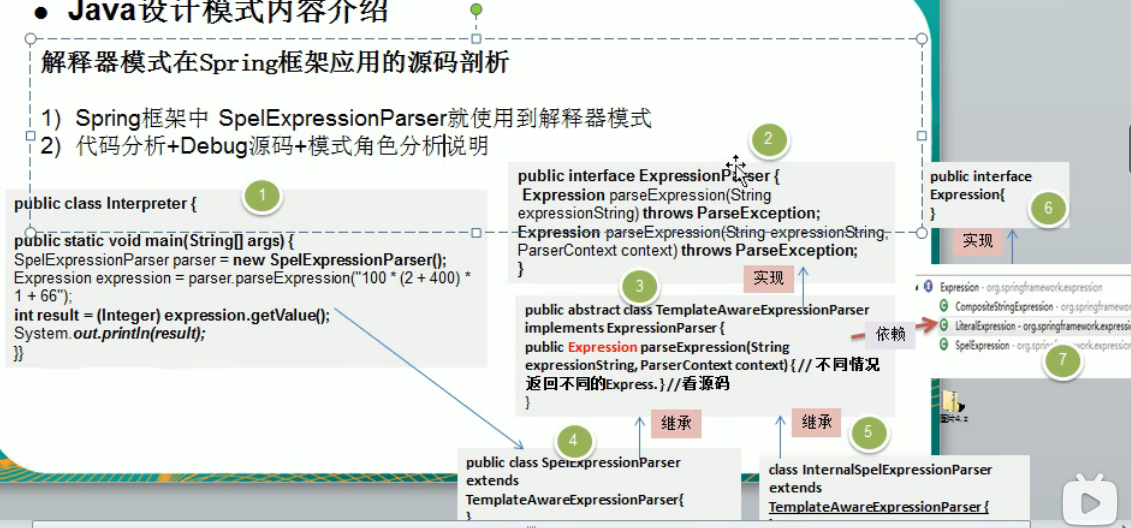


* 项目相关面试题

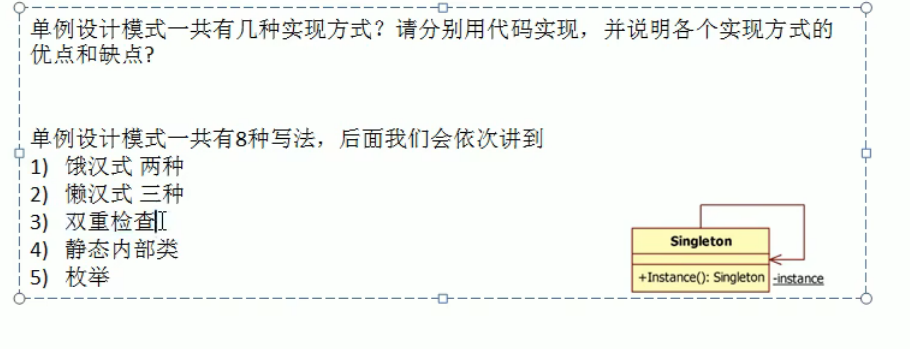


* 解释器模式





* 单例模式



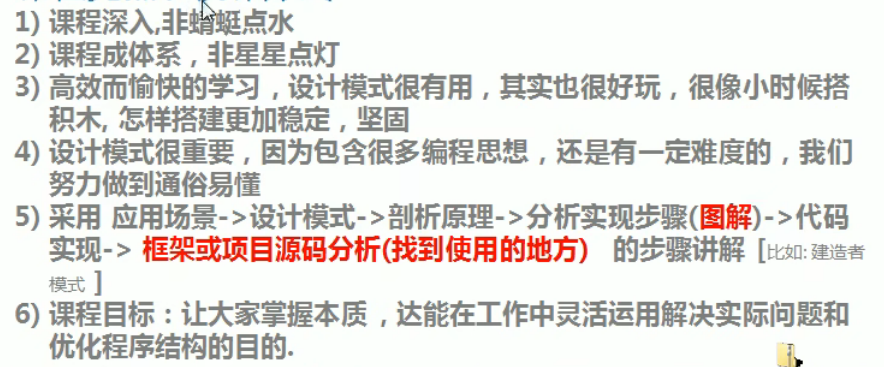
### 1.2何时会用到设计模式

#### 1.2.1 设计模式的重要性





### 1.3 课程两点和授课方式



## 设计模式的七大原则

### 2.1设计模式的目的

编写软件过程中，程序员面临着来自 耦合性，内聚性以及可维护性，可扩展性，重用性，灵活性 等多方面的挑战，设计模式就是为了让程序（软件），具有更好的：

1. 代码重用性（即：相同功能的代码，不用重复编写）
2. 可读性（即：编程规范性，便于其他程序员的阅读和理解）
3. 可扩展性（即：当需要增加新的功能时，非常的方便，也成为可维护性）
4. 可靠性（即：当我们增加新的功能后，对原来的功能没有影响）
5. 使程序呈现**高内聚**，**低耦合**的特性

分享金句：

设计模式包含了面向对象的精髓，“懂了设计模式，你就懂了面向对象分析和设计（OOA/D）的精要”

Scott Mayers在其巨著《Effective C++》就曾说过：C++老手和C++新手的区别就是前者手背上有很多伤疤。

### 2.2设计模式七大原则

设计模式原则，其实就是程序员在编程时，应当遵守的原则，也是各种设计模式的基础（即：设计模式为什么这样设计的依据）

1. 单一职责原则
2. 接口隔离原则
3. 依赖倒转（倒置）原则
4. 里氏替换原则
5. 开闭原则
6. 迪米特法则
7. 合成复用原则

#### 2.2.1单一职责原则

* **基本介绍**

对类来说的，即一个类应该只负责一项职责。如类A负责两个不同职责：职责1，职责2。当职责1需求变更而改变A时，可能造成职责2执行错误，所以需要将类A的粒度分解为A1，A2。

* **应用实例**

1. 以交通工具案例讲解
2. 代码部分
3. 方案1[分析说明]
4. 方案2[分析说明]
5. 方案3[分析说明]

* **单一职责原则注意事项和细节**

1. 降低类的复杂度，一个类只负责一项职责
2. 提高类的可读性，可维护性
3. 降低变更引起的风险
4. 通常情况下，我们应当遵守单一职责原则，只有逻辑足够简单，才可以在代码级违反单一职责原则；只有类中方法数量足够少，可以在方法级别保持单一原则。

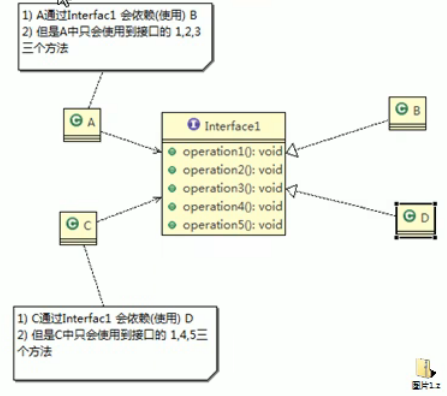
* **代码包路径：**

src/com/example/principle/singleresponsibility

#### 2.2.2接口隔离原则

* **基本介绍**

1. 客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应建立在最小的接口上。
2. 先看一张图：



1. 类A通过接口Interface1依赖类B，类C通过解耦Interface1依赖类D，如果接口Interface1对于类A和类C来说不是最小接口，那么类B和类D必须去实现它们不需要的方法
2. 按隔离原则应当这样处理：

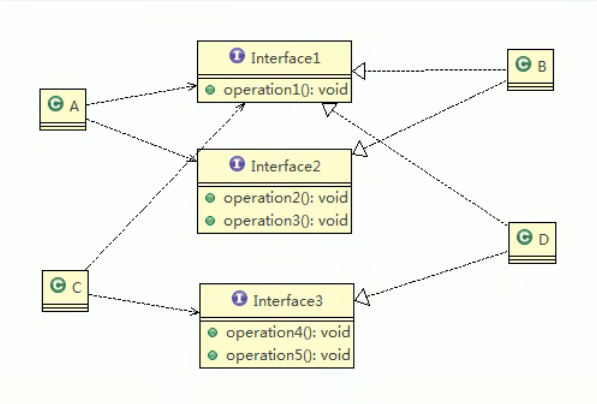
将接口Interface1拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

* **应用实例**

类A通过接口Interface1依赖类B，类C通过接口Interface1依赖类D，请编写代码完成此应用实例。

* **应传统方法的问题和使用接口隔离原则改进**

1. 类A通过接口Interface1依赖类B，类C通过接口Interface1依赖类D，如果接口Interface1对于类A和类C来说不是最小接口，那么类B和类D必须去实现它们不需要的方法。
2. 将接口Interface1拆分为独立的几个接口，类A和类C分别与它们需要的接口建立依赖关系。也就是采用接口隔离原则。
3. 接口Interface1中出现的方法，根据实际情况拆分为三个接口。
4. 代码实现。



* **代码包路径：**

src/com/example/principle/segregation

#### 2.2.3依赖倒转（倒置）原则

* **基本介绍**

依赖倒转原则（Dependence Inversion Principle）是指：

1. 高层模块不应该依赖低层模块，二者都应该依赖其对象
2. 抽象不应该依赖细节，细节应该依赖抽象
3. 依赖倒转（倒置）的中心思想是面向接口编程
4. 依赖倒转原则是基于这样的设计理念，相对于细节的多边形，抽象的东西要稳定的多。以抽象为基础搭建的架构比以细节为基础的架构要稳定得多，在java中，抽象指的是接口或抽象类，细节就是具体的实现类。
5. 使用接口或抽象类的目的是制定好规范，而不涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

* **应用实例**

请编程完成Person接收消息的功能

实现方案1+分析说明

实现方案1+分析说明

* **依赖关系传递的三种方式和应用案例**

1. 接口传递

应用代码案例

1. 构造方法传递

应用代码案例

1. setter方式传递

应用代码案例

* **依赖倒转原则的注意事项和细节**

1. 底层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好。
2. 变量的声明类型尽量是抽象类或接口，这样我们的变量引用和实际对象间，就存在一个缓冲层，利于程序扩展和优化。
3. 继承时遵循里氏替换原则

* **代码包路径：**

src/com/example/principle/inversion

#### 2.2.4里氏替换原则

* **oo中的继承性的思考和说明**

1）继承包含这样一层含义：父类中凡是已经实现好的方法，实际上是在设定规范和契约，虽然它不强制要求所有的子类必须遵守这些契约，但是如果子类对这些已经实现的方法任意修改，就会对整个集成体系造成破坏。

2）**继承在给程序设计带来便利的同时**，也带来了**弊端**。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加对象的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能产生故障。

3）问题提出：**在编程中，如何正确的使用继承？=>里氏替换原则**。

* **基本介绍**

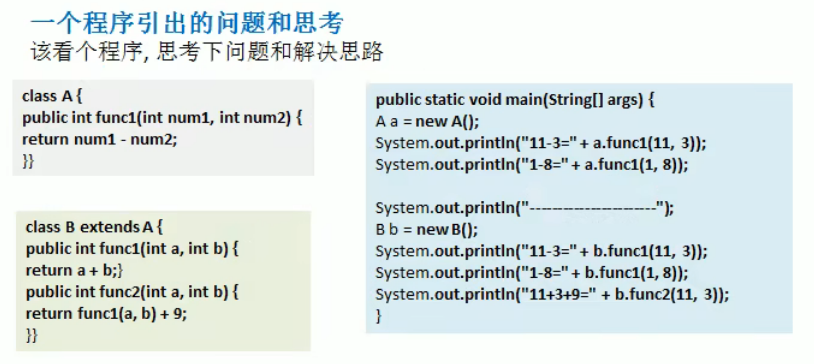
1）里氏替换原则（Liskov Substitution Principle）在1998年，由麻省理工学院的一位姓里的女士提出的。

2）如果对每个类型为T1的对象o1，都有类型T2的对象o2，使得以T1定义的所有程序P在所有的对象o1都代换成o2时，程序P的行为没有发生变化，那么类型T2是类型T1的子类型。换句话说，所有引用基类的地方必须能透明地使用其子类的对象。

3）在使用继承时，遵循里氏替换原则，在子类中尽量不要重写父类方法。

4）里氏替换原则告诉我们，继承实际上让两个类耦合性增强了，在适当的情况下，可以通过聚合、组合、依赖来解决问题。

* **一个程序引出的问题和思考**



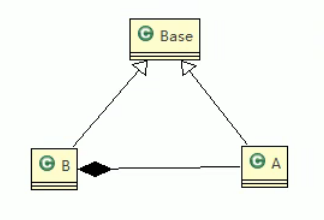
* **解决方法**

1）我们发现原来运行正常的相减功能发生了错误。原因就是类B无意中重写了父类的方法，造成原有功能出现错误。在实际编程中，我们常常会通过重写父类的方法完成新的功能，这样写起来虽然简单，但整个继承体系的复用性会比较差。特别是运行多态比较频繁的时候。

2）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、组合等关系替代。

3）改进方案：代码实现，见类：

src/com/example/principle/liskov/improve/Liskov.java



* **代码包路径：**

src/com/example/principle/liskov

#### 2.2.5开闭原则

* **基本介绍**

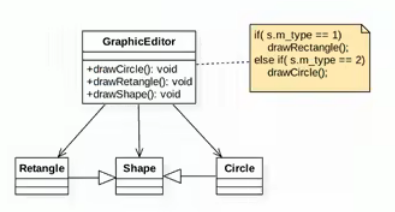
1）开闭原则（Open closed Principle）是编程中最基础、最重要的设计原则。

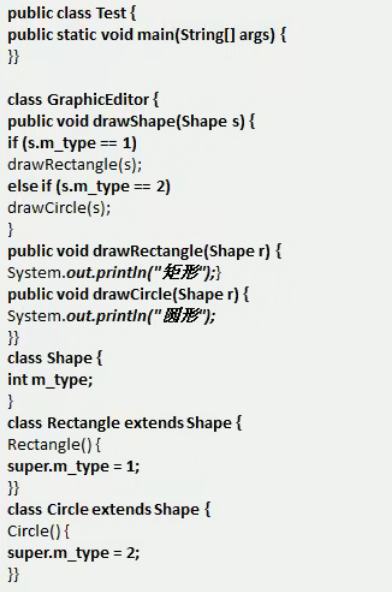
2）一个软件实体如类、模块和函数应对扩展开放（对提供方），对修改关闭（对使用方）。用抽象构建框架，用实现扩展细节。

3）当软件需要变化时，尽量**通过扩展**软件实体的行为来实现变化，而**不是通过修改**已有的代码来实现变化。

4）编程中遵循其他原则，以及使用设计模式的目的就是遵守**开闭原则**。

* **看下面一段代码**





* **方式1的优缺点(src/com/example/principle/ocp/Ocp.java)**

1)优点是比较好理解，简单易操作。

2)缺点是违反了设计模式的ocp原则，即**对扩展开放(提供方)**，**对修改关闭(使用方)**。即当我们给类增加新功能的时候，尽量不要修改代码，或者尽可能少修改代码。

3)比如我们这时要增加一个图形类(三角形)，我们需要做如下修改，修改的地方较多。

4)代码演示（见下方《方式1的改进的思路分析》）。

* **方式1的改进的思路分析**

改进的思路分析

思路：把创建Shape类做成抽象类，并提供一个抽象的draw方法，让子类去实现即可，这样我们有新的图形种类时，只需要让新的图形类继承Shape，并实现draw方法即可，使用方的代码就不需要修改->满足了开闭原则。

* **方式2来解决**

1)方式2的设计方案：定义一个Shape抽象类。

2)看代码示例。

3)从方式2看，代码满足了ocp原则。

代码路径：

src/com/example/principle/ocp/improve/Ocp.java

#### 2.2.6迪米特法则

* **基本介绍**

1)一个对象应该对其他对象保持最少的了解。

2)类与类关系越密切，耦合度越大。

3)迪米特法则(Demeter Principle)又叫**最少知道原则**，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的public方法，不对外泄露任何信息。

4)迪米特法则还有个更简单的定义：只与直接的朋友通信。

5)直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间都是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友。也就是说，陌生的类最好不要以局部变量的形式出现在类的内部。

* **应用实例**

1. 有一个学校，下属有各个学院和总部，现要求打印出学校总部员工ID和学院员工的id
2. 编程实现上面的功能，看代码演示。

src/com/example/principle/demeter/Demeter1.java

* **应用实例改进**

1)前面设计的问题在于SchoolManager中，CollegeEmployee类并不是SchoolManager类的直接朋友(分析)。

2)按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。

3)对代码按照迪米特法则进行改进(看老师演示)。

src/com/example/principle/demeter/improve/Demeter1.java

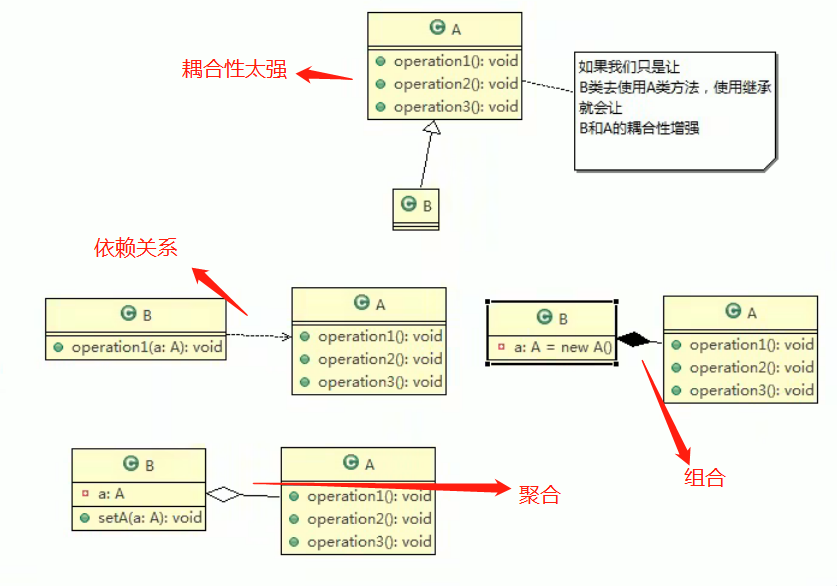
* **迪米特法则注意事项和细节**

1. 迪米特法则的核心是降低类之间的低耦合。
2. 但是注意：由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低(对象间)耦合关系，并不是要求完全没有依赖关系。

#### 2.2.7合成复用原则(Composite Reuse Principle)

* **基本介绍**

原则是尽量使用合成/聚合的方式，而不是使用继承。



* **设计原则的核心思想**

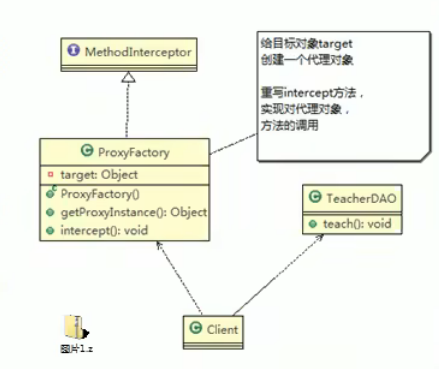
1. 找出应用中可能需要变化之处，把他们独立起来，不要和那些不需要变化的代码混在一起。
2. 针对接口编程，而不是针对实现编程。
3. 为了**交互对象之间的松耦合**设计而努力。

## UML类图

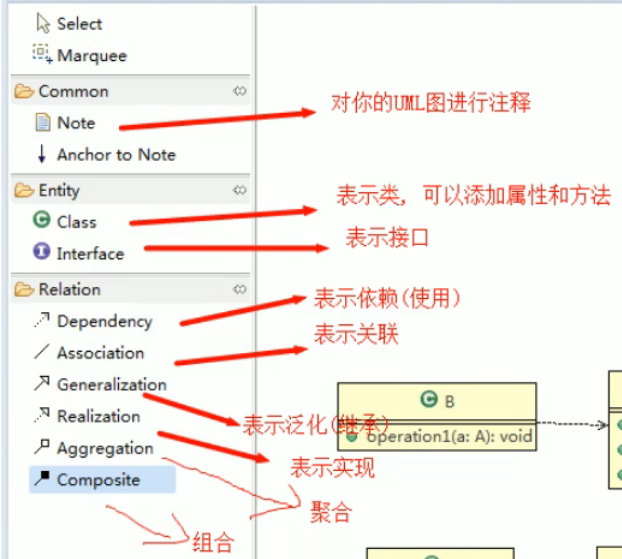
### 3.1 UML基本介绍

1)UML—Unified modeling language UML(统一建模语言)，是一种用于软件系统分析和设计的语言工具，它用于帮助软件开发人员进行思考和记录思路的结果。

2)UML本身是一套符号的规定，就像数学符号和化学符号一样，这些符号用于描述软件模型中的各个元素和他们之间的关系，比如类、接口、实现、泛化、依赖、组合、聚合等，如下图：



3)使用UML来建模，常用的工具有Rational Rose，也可以使用一些插件来建模。



### 3.2 UML图

画UML图与写文章差不多，都是把自己的思想描述给别人看，关键在于思路和条理，UML图分类：

1. 用例图(usecase)
2. 静态结构图：类图、对象图、包图、组件图、部署图
3. 动态行为图：交互图（时序图与协作图）、状态图、活动图

说明：

1. 类图是描述类与类之间的关系的，是UML图中最核心的。
2. 在讲解设计模式时，我们必然会使用类图，为了让学员们能够把设计模式学到位，需要先给大家讲解类图。
3. 温馨提示：如果已经掌握UML类图的学员，可以直接听设计模式的章节。

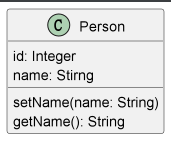
### 3.3 UML类图

1）用于描述系统中的类（对象）本身的组成和类（对象）之间的各种静态关系。

2）类之间的关系：依赖、泛化（继承）、实现、关联、聚合和组合。

3）类图简单举例：

|  |
| --- |
| public class Person {//代码形式->类图   private Integer id;   private String name;   public void setName(String name) {  this.name = name;  }   public String getName() {  return name;  }  } |

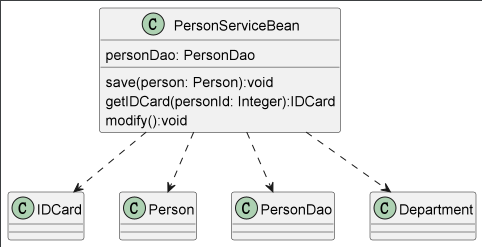


#### 3.3.1 类图-依赖关系（Dependence）

只要是在类中用到了对方，那么他们之间就存在依赖关系。如果没有对方，连编译都通过不了。

|  |
| --- |
| public class PersonServiceBean {  private PersonDao personDao;//类  public void save(Person person) {  }  public IDCard getIDCard(Integer personId) {  return null;  }  public void modify() {  Department department = new Department();  }  } |
| public class Department {  } |
| public class IDCard {  } |
| public class Person {  } |
| public class PersonDao {  } |

* **对应的类图：**



* **小结：**

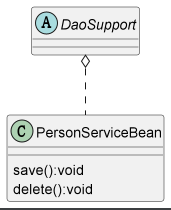
1. 类中用到了对方。
2. 如果是类的成员属性。
3. 如果是方法的返回类型。
4. 如果是方法接收的参数类型。
5. 方法中使用到。

#### 3.3.2 类图-泛化关系（generalization）

泛化关系实际上就是继承关系，他是依赖关系的特例。

|  |
| --- |
| public abstract class DaoSupport {  public void save(Object entity) {  }  public void delete(Object id) {  }  } |
| public class PersonServiceBean extends DaoSupport {  } |

* **对应的类图：**



* **小结：**

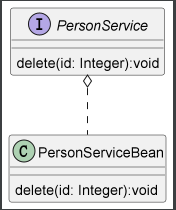
1. 泛化关系实际上就是继承关系。
2. 如果A类继承了B类，我们就说A和B存在泛化的关系。

#### 3.3.3 类图-实现关系（Implementation）

实现关系实际上就是A类实现B接口，他是依赖关系的特例。

|  |
| --- |
| public interface PersonService {  public void delete(Integer id);  } |
| public class PersonServiceBean implements PersonService{  @Override  public void delete(Integer id) {  }  } |

* **对应的类图：**



#### 3.3.4 类图-关联关系（Association）

关联关系实际上就是类与类之间的联系，他是依赖关系的特例。

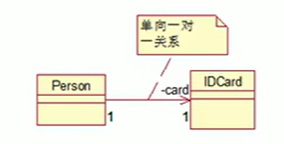
关联具有导航性：即双向关系或单向关系。

关系具有多重性：如“1”（表示有且仅有一个），“0...”（表示0个或多个），“0,1”（表示0个或者一个），“n...m”（表示n到m个都可以），“m...”（表示至少m个）。

* **单向一对一关系**

|  |
| --- |
| public class Person {  private IDCard card;  } |
| public class IDCard {  } |

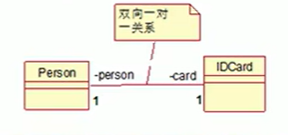
**对应的类图：**



* **双向一对一关系**

|  |
| --- |
| public class Person {  private IDCard card;  } |
| public class IDCard {  private Person person;  } |

**对应的类图：**



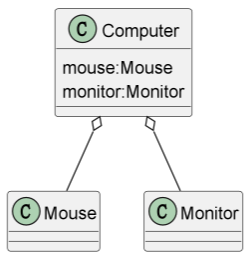
#### 3.3.5 类图-聚合关系（Aggregation）

聚合关系（Aggregation）表示的是整体和部分的关系，整体与部分可以分开，聚合关系是关联关系的特例，所以他具有关联的导航性与多重性。

如：一台电脑由键盘（keyboard）、显示器（monitor）、鼠标等组成；组成电脑的各个配件是可以从电脑上分离出来的，使用空心菱形的实现来表示。

|  |
| --- |
| public class Computer {  private Mouse mouse;  private Monitor monitor;  public void setMouse(Mouse mouse) {  this.mouse = mouse;  }  public void setMonitor(Monitor monitor) {  this.monitor = monitor;  }  } |

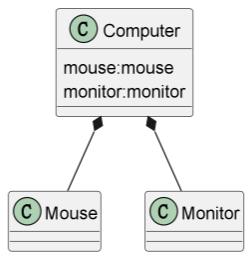
* **对应的类图：**



如果我们认为Mouse、Monitor和Computer是不可分离的，则升级为组合关系。

|  |
| --- |
| public class Computer {  private Mouse mouse = new Mouse();  private Monitor monitor = new Monitor();  } |

* **对应的类图：**



#### 3.3.6 类图-组合关系（Composition）

组合关系，也是整体与部分的关系，但是整体与部分不可以分开。

再看一个案例：在程序中我们定义实体类：Person和IDCard、Head，那么Head和Person就是组合，IDCard和Person就是聚合。

但是如果在程序中Person实体中定义了对IDCard进行级联删除，即删除Person时连同IDCard一起删除，那么IDCard和Person就是组合了。

|  |
| --- |
| public class Person {  private IDCard card;//聚合关系  private Head head = new Head();//组合关系  } |
| public class Head {  } |
| public class IDCard {  } |

* **对应的类图：**

