# java设计模式学习

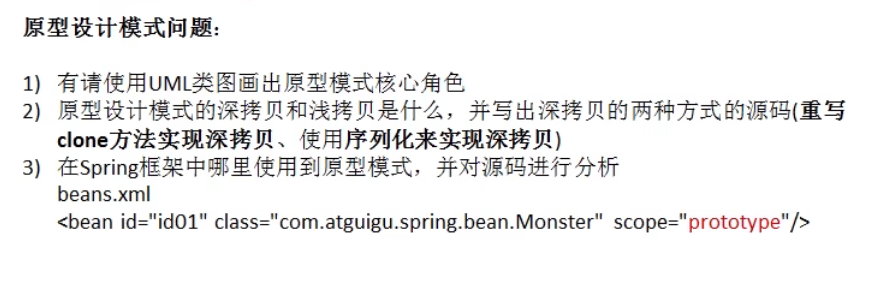
## java设计模式内容介绍

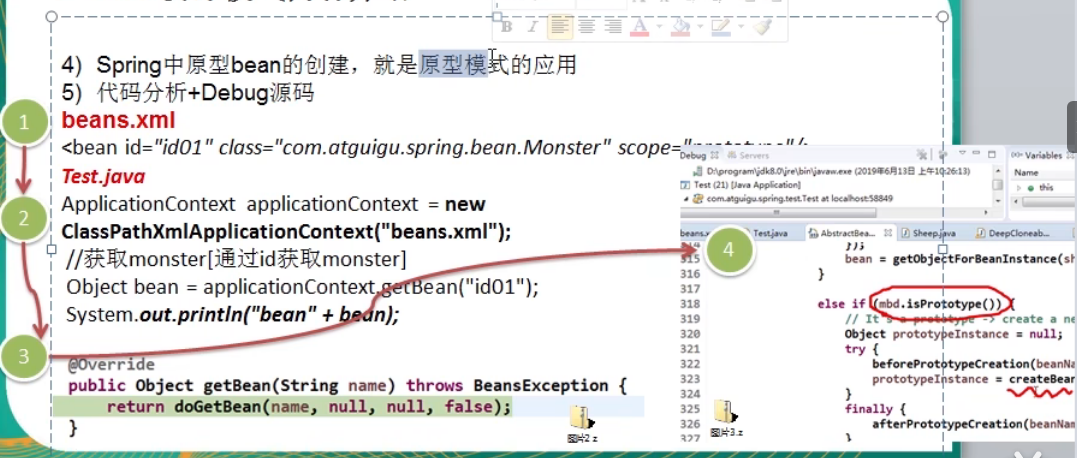
本笔记参考哔哩哔哩《尚硅谷Java设计模式（图解+框架源码剖析）》内容记录。

具体代码见项目：designPattern

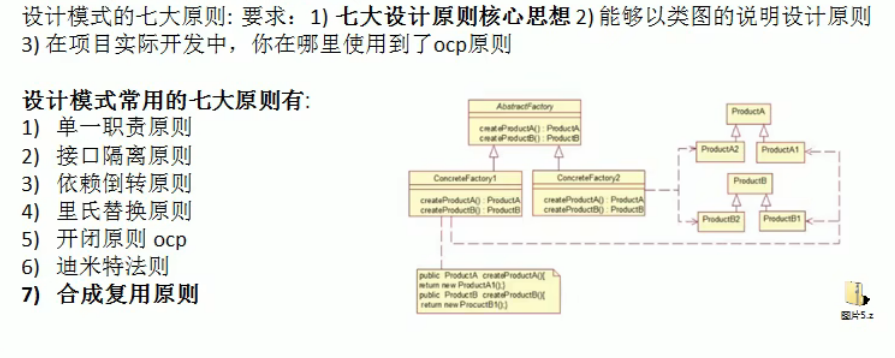
### 1.1 先看几个经典的面试题

* 原型设计模式问题：

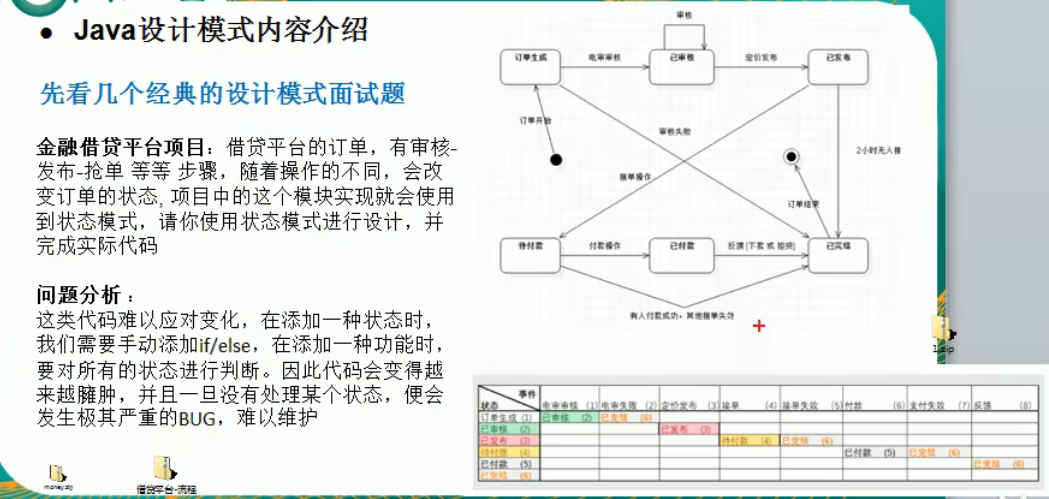




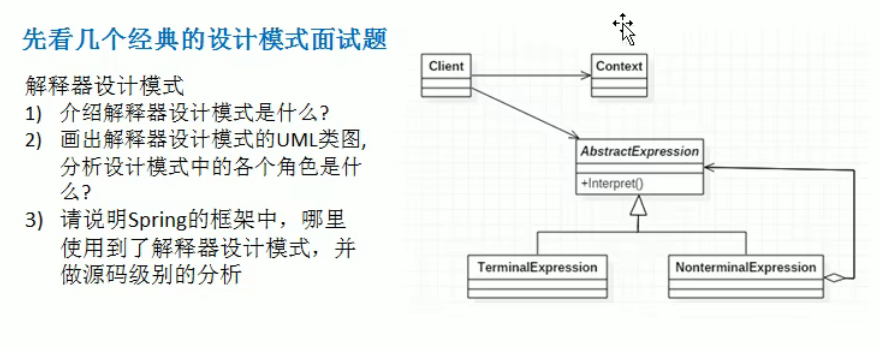
* 设计模式的七大原则：

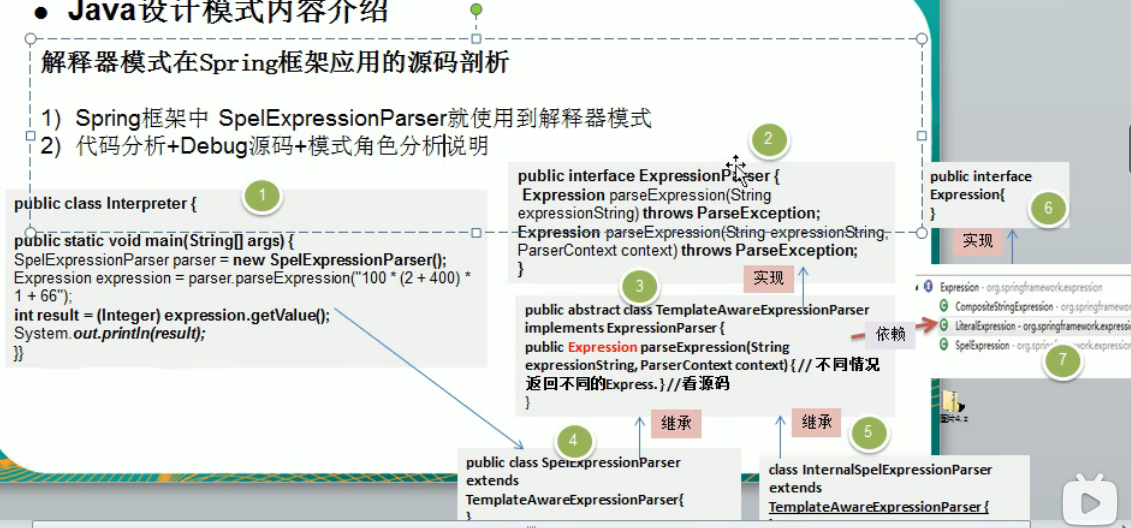


* 项目相关面试题

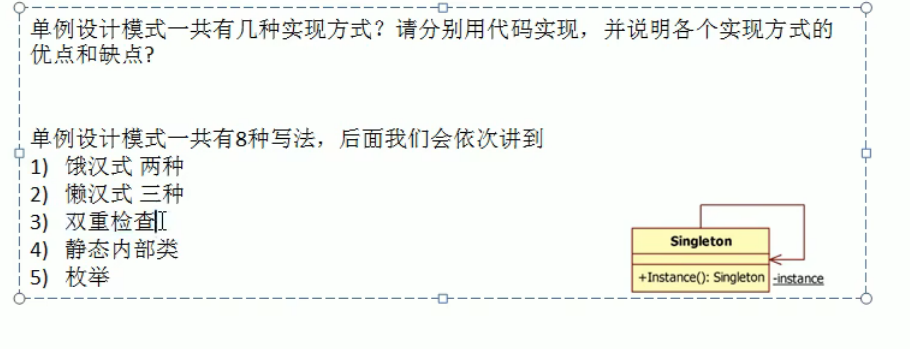


* 解释器模式





* 单例模式



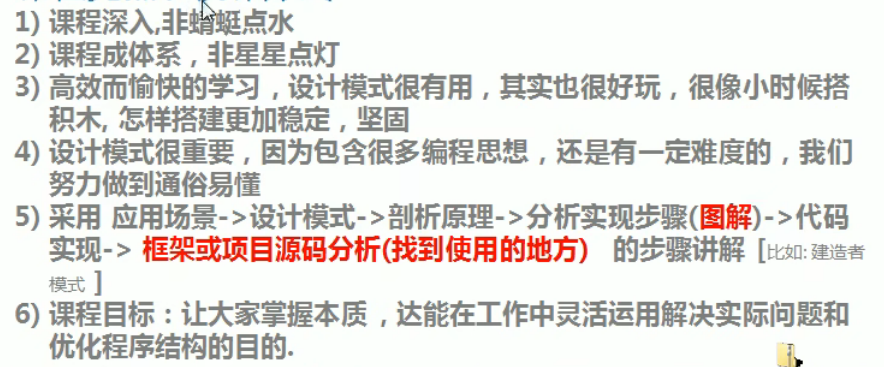
### 1.2 何时会用到设计模式

#### 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。

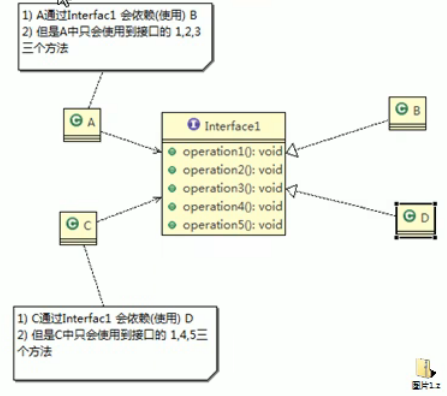
* **代码包路径：**

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：

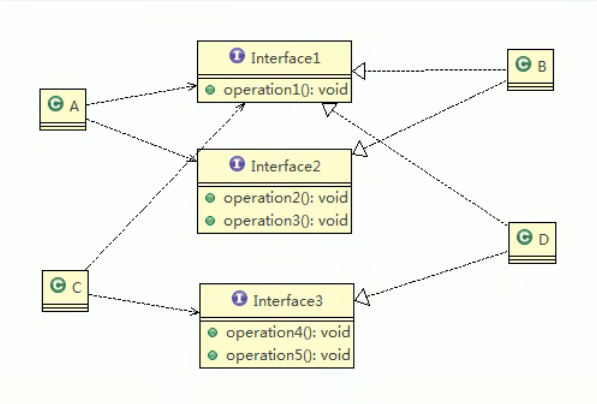
将接口Interface1拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

* **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**

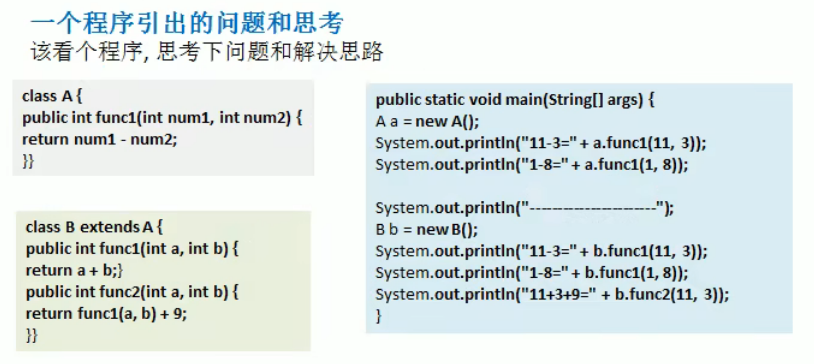
1）里氏替换原则（Liskov Substitution Principle）在1998年，由麻省理工学院的一位姓里的女士提出的。

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**



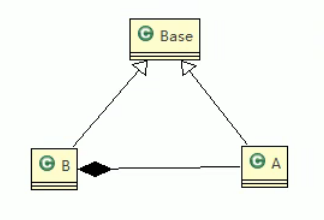
* **解决方法**

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**

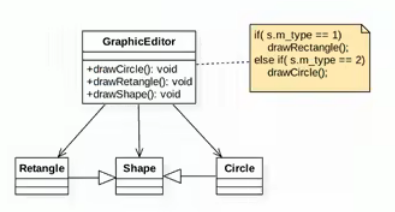
1）开闭原则（Open closed Principle）是编程中最基础、最重要的设计原则。

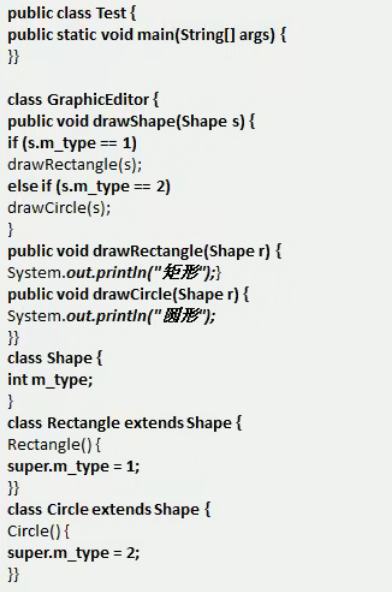
2）一个软件实体如类、模块和函数应对扩展开放（对提供方），对修改关闭（对使用方）。用抽象构建框架，用实现扩展细节。

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

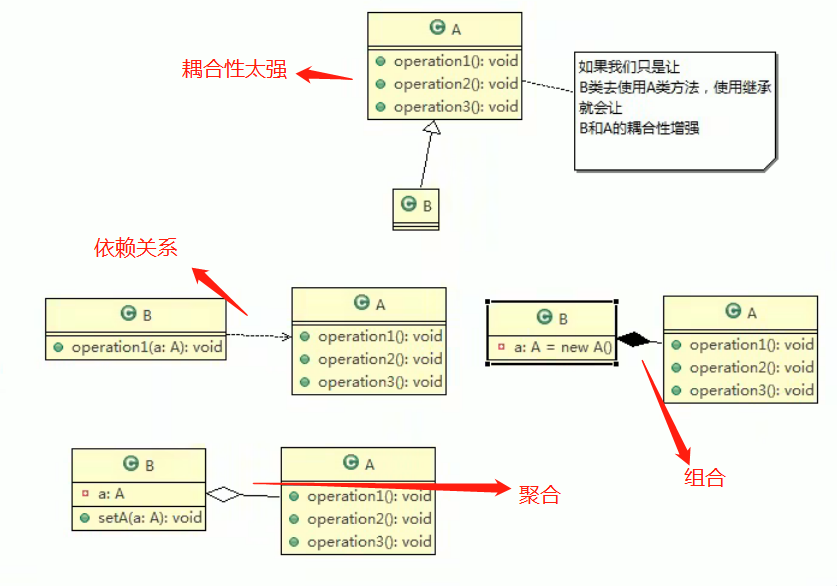
* **迪米特法则注意事项和细节**

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**

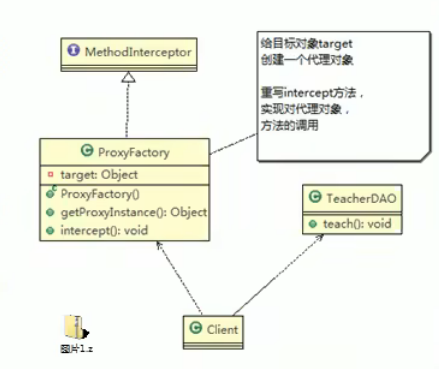
1. 找出应用中可能需要变化之处，把他们独立起来，不要和那些不需要变化的代码混在一起。
2. 针对接口编程，而不是针对实现编程。
3. 为了**交互对象之间的松耦合**设计而努力。

## UML类图

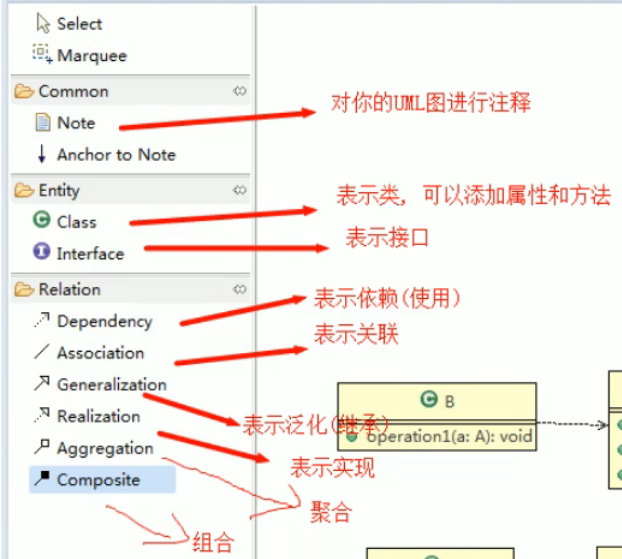
### 3.1 UML基本介绍

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。

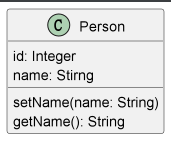
### 3.3 UML类图

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;  }  } |

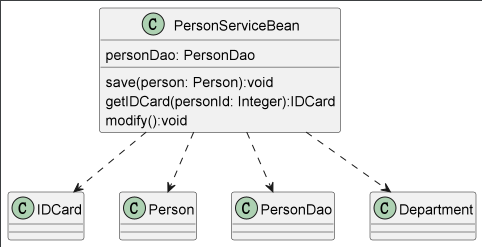


#### 3.3.1 类图-依赖关系（Dependence）

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：**

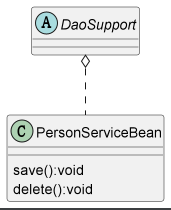
1. 类中用到了对方。
2. 如果是类的成员属性。
3. 如果是方法的返回类型。
4. 如果是方法接收的参数类型。
5. 方法中使用到。

#### 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：**

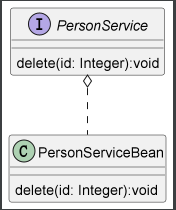
1. 泛化关系实际上就是继承关系。
2. 如果A类继承了B类，我们就说A和B存在泛化的关系。

#### 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。

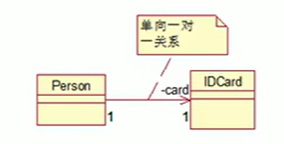
关联具有导航性：即双向关系或单向关系。

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 {  } |

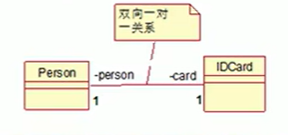
**对应的类图：**



* **双向一对一关系**

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

**对应的类图：**



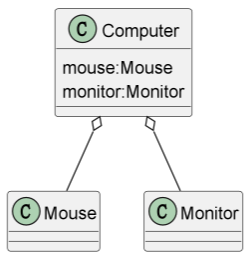
#### 3.3.5 类图-聚合关系（Aggregation）

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;  }  } |

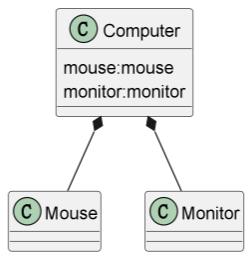
* **对应的类图：**



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）

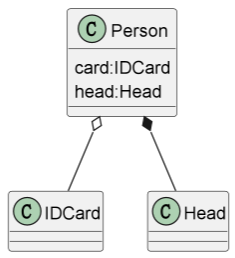
组合关系，也是整体与部分的关系，但是整体与部分不可以分开。

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：**



## Java设计模式

* **掌握设计模式的层次**

1. 第1层：刚开始学编程不久，听说过什么是设计模式。
2. 第2层：有很长时间的编程经验，自己写了很多代码，其中用到了设计模式，但是自己却不知道。
3. 第3层：学习过了设计模式，发现自己已经在使用了，并且发现了一些新的模式挺好用的。
4. 第4层：阅读了很多别人写的源码和框架，在其中看到别人设计模式，并且能够领会**设计模式的精妙和带来的好处**。
5. 第5层：代码写着写着，自己都没有意识到使用了设计模式，并且熟练地写了出来。

* **设计模式介绍**

1. 设计模式是程序员在面对同类软件工程设计问题所总结出来的有用的经验，**模式不是代码**，而是某类问题的通用解决方案，设计模式（Design pattern）代表了最佳的实践。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
2. 设计模式的本质提高**软件的维护性、通用性和扩展性，并降低软件的复杂度**。
3. 《设计模式》是经典的书，作者是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides Design（俗称“四人组GOF”）
4. 设计模式并不局限于某种语言，java、php、c++都有设计模式。

* **设计模式类型**

设计模式分为三种类型，共23种。

1. **创建型模式**：单例模式、抽象工厂模式、原型模式、建造者模式、工厂模式。
2. **结构性模式**：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
3. **行为型模式**：模板方法模式、命令模式、访问者模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模式、策略模式、职责链模式（责任链模式）。

注意：不同的书籍上对分类和名称略有差别。

### 4.1 单例模式

* **单例设计模式介绍**

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且改类只提供一个取的其对象实例的方法（静态方法）。

比如Hibernate和SessionFactory，它充当数据存储源的代理，并负责创建Session对象。SessionFactory并不是轻量级的，一般情况下，一个项目通常需要一个SessionFactory就够了，这时就会使用到单例模式。

* **单例设计模式的八种方式（见4.1.1-4.1.8）。**

#### 4.1.1 饿汉式（静态常量）

* **饿汉式（静态常量）应用实例**

**步骤如下：**

1. 构造器私有化(防止new)。
2. 类的内部创建对象。
3. 向外暴露一个静态的公共方法。getInstance
4. 代码实现(src/com/example/singleton/type1/SingletonTest01.java)。

**优缺点说明：**

1. 有点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。
2. 缺点：在类装载的时候就完成实例化，没有打到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。
3. 这种方式基于classloader机制避免了多线程的同步问题，不过instance在类装载时就实例化，在单例模式中大多数都是调用getInstance方法，但是导致类装载的原因有很多种，因此就不能确定有其他的方式（或其他的静态方法）导致类装载，这时候初始化instance就没有打到lazy loading的效果。
4. 结论：这种单例模式可用，可能造成内存浪费。

#### 4.1.2 饿汉式（静态代码块）

* **饿汉式（静态代码块）应用实例**

代码实现（src/com/example/singleton/type2/SingletonTest02.java）。

**优缺点说明：**

1. 这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，也是类装载的时候，就执行静态代码块中的代码，初始化类的实例。优缺点和上面是一样的。
2. 结论：这种单例模式可用，但是可能造成内存浪费。

#### 4.1.3 懒汉式（线程不安全）

* **懒汉式（线程不安全）应用实例**

代码实现（src/com/example/singleton/type3/SingletonTest03.java）。

**优缺点说明：**

1. 起到了lazy loading的效果，但是只能在单线程下使用。
2. 如果在多线程下，一个线程进入了if(singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式。
3. 结论：在实际开发中，不要使用这种方式。

#### 4.1.4 懒汉式（线程安全，同步方法）

* **懒汉式（线程安全，同步方法）应用实例**

代码实现（src/com/example/singleton/type4/SingletonTest04.java）。

**优缺点说明：**

1. 解决了线程不安全问题。
2. 效率太低了，每个线程在想获取类的实例时候，执行getInstance()方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接return就行了。方法进行同步效率太低。
3. 结论：在实际开发中，不推荐使用这种方式。

#### 4.1.5 懒汉式（线程安全，同步代码块）

* **懒汉式（线程安全，同步代码块）应用实例**

代码实现（src/com/example/singleton/type5/SingletonTest05.java）。

**优缺点说明：**

1. 这种方式，本意是想对第四种实现方式进行改进，因为前面同步方法效率太低，改为同步产生实例的代码块
2. 但是这种同步并不能起到线程同步的作用。跟第3中实现方式遇到的情形一直，假如一个线程进入了if(singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。
3. 结论：在实际开发中，不能使用这种方式。

#### 4.1.6 双重检查

* **双重检查应用实例**

代码实现（src/com/example/singleton/type6/SingletonTest06.java）。

**优缺点说明：**

1. Double-Check概念是多线程开发中经常使用到的，如代码所示，我们进行了两次if(singleton == null)检查，这样就可以保证线程安全了。
2. 这样，实例化代码只用执行一次，后面再次访问时，判断if(singleton == null)，直接return实例化对象，也避免反复进行方法同步。
3. 线程安全、延迟加载、效率较高。
4. 结论：在实际开发中，推荐使用这种单利设计模式。

#### 4.1.7 静态内部类

* **静态内部类应用实例**

代码实现（src/com/example/singleton/type7/SingletonTest07.java）。

**优缺点说明：**

1. 这种方式采用了类装载的机制来保证初始化实例时只有一个线程。
2. 静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化时，调用getInstance()方法，才会装载SingletonInstance类，从而完成Singleton的实例化。
3. 类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程安全，在类进行初始化时，别的线程是无法进入的。
4. 有点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高。
5. 结论：推荐使用。

#### 4.1.8 枚举

* **枚举应用实例**

代码实现（src/com/example/singleton/type8/SingletonTest08.java）。

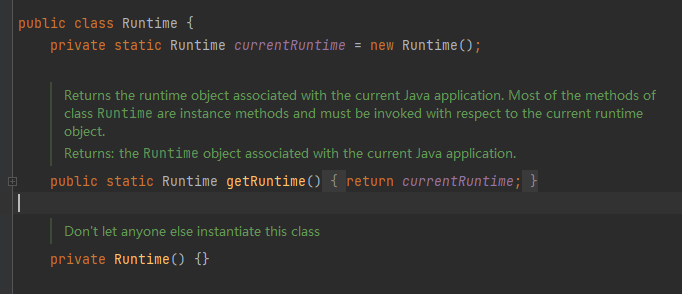
**优缺点说明：**

1. 借助JDK1.5中添加的枚举类来实现单例模式。不仅能避免多线程同步问题，而且还能防止反序列化重新创建对象。
2. 这种方式是Effective Java作者Josh Bloch提倡的方式。
3. 结论：推荐使用。

#### 4.1.9 单例模式在JDK应用的源码分析

* **单例模式在JDK应用的源码分析**

1. 我们JDK中，java.lang.Runtime就是经典的单例模式。
2. 代码分析+Debug源码+代码说明。



* **单例模式注意事项和细节说明**

1. 单例模式保证了系统内寸中改类只存在一个对象，节省了系统资源，对于一些需要频繁创建对象销毁的对象，使用单例模式可以提高系统性能。
2. 当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用new。
3. 单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时过多或耗费资源过多（即：重量级对象），但又经常用到的对象、工具类对象、频繁访问数据库或文件的对象（比如数据源、session工厂等）。

### 4.2 工厂模式

#### 4.2.1 传统方式解决

* **看一个需求：**

看一个披萨的项目：要便于披萨种类的扩展，要便于维护。

1. 披萨的种类有很多（比如GreekPizza、CheesePizza等）。
2. 披萨的制作有prepare、bake、cut、box。
3. 完成披萨店订购功能。

* **使用传统的方式来完成**

1. 思路分析（类图）
2. 代码演示（src/com/example/factory/simplefactory/pizzastore）。

* **传统的方式的优缺点**

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，对修改关闭。即当我们给类增加新功能的时候，尽量不修改代码，或者尽可能少修改代码。
3. 比如我们这时要新增加一个Pizza的种类(Pepper披萨)，我们需要做如下修改。

|  |
| --- |
| //新增下述类  public class PepperPizza extends Pizza{  @Override  public void prepare() {  System.out.println(" 给胡椒披萨 准备原材料");  }  } |
| 在OrderPizza中新增下述代码 |

1. 改进的思路分析

分析：修改代码可以接受，但是如果我们在其他的地方也有创建Pizza的代码，就意味着，也需要修改，而创建Pizza的代码，往往有多处。

思路：把创建Pizza对象封装到一个类中，这样我们有新的Pizza种类时，只需要修改类就可以，其他有创建到Pizza对象的代码就不需要修改了🡪简单工厂模式。

#### 4.2.2 简单工厂模式

* **简单工厂模式基本介绍**

1. 简单工厂模式是属于创建型模式，是工厂模式的一种。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式。
2. 简单工厂模式：定义了一个创建对象的类，由这个类来**封装实例化对象的行为**（代码）。
3. 在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就会用到工厂模式。

* **使用简单工厂模式**

1. 简单工厂模式的设计方案：定义一个可以实例化Pizza对象的类，封装创建对象的代码。
2. 代码示例（src/com/example/factory/simplefactory/pizzastore/order/SimpleFactory.java）。

#### 4.2.3 工厂方法模式

* **看一个新的需求**

披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如北京的奶酪pizza、北京的胡椒pizza或者是伦敦的奶酪pizza、伦敦的胡椒pizza。

思路1：

使用简单工厂模式，创建不同的简单工厂类，比如BJPizzaSimpleFactory、LDPizzaSimpleFactory等等。从当前这个案例来说，也是可以的，但是考虑到项目的规模，以及软件的可维护性、可扩展性不是特别好。

思路2：

使用工厂方法模式。

* **工厂方法模式介绍**

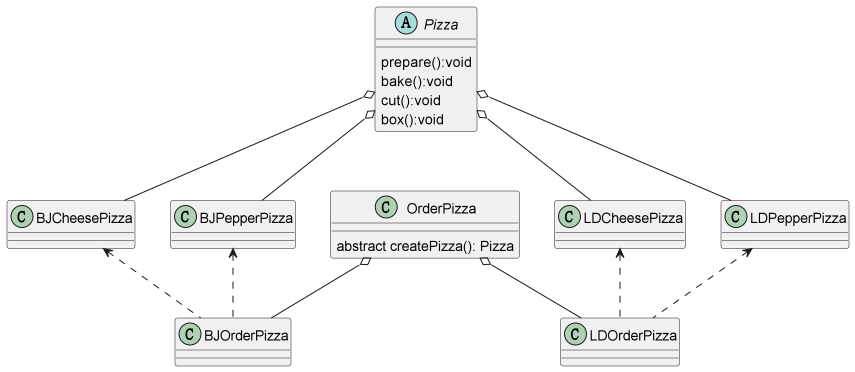
**工厂方法模式设计方案**：将披萨项目的实例化功能抽象成抽象方法，在不同的口味点餐子类中具体实现。

**工厂方法模式**：定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类。

* **工厂方法模式应用案例**

披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如北京的奶酪pizza、北京的胡椒pizza或者是伦敦的奶酪pizza、伦敦的胡椒pizza。

1. 思路分析图解。



1. 代码实现（src/com/example/factory/factorymethod）。

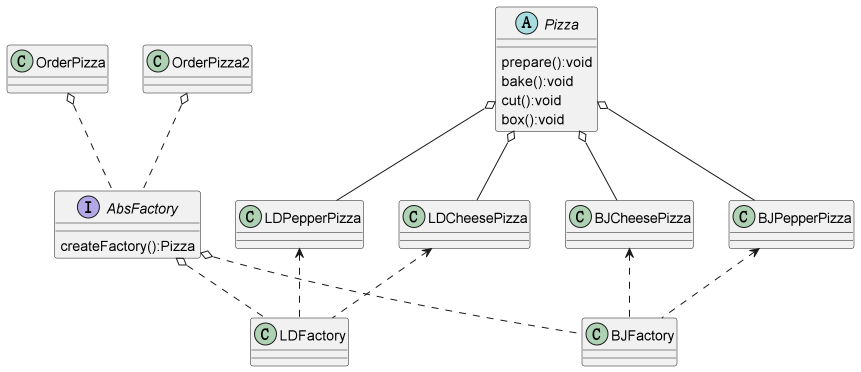
#### 4.2.4 抽象工厂模式

* **基本介绍**

1. 抽象工厂模式：定义了一个interface用于创建相关或有依赖关系的对象簇，而无需指明具体的类。
2. 抽象工厂模式可以将简单工厂模式和工厂方法模式进行整合。
3. 从设计层面看，抽象工厂模式就是对简单工厂模式的改进（或者称为进一步的抽象）。
4. 将工厂抽象成两层，AbsFactory（抽象工厂）和具体实现的工厂子类。程序员可以根据创建对象类型使用对应的工厂子类。这样将单个的简单工厂变成了工厂簇，更利于代码的维护和扩展。

* **抽象工厂模式来完成披萨项目**

1）类图：



2）代码实现（src/com/example/factory/abstractfactory）。

* **工厂模式在JDK-Calender应用的源码分析**

1. JDK中的Calender类中，就使用了简单工厂模式
2. 源码分析+Debug源码+说明。

* **工厂模式小结**

1. 工厂模式的意义：将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的依赖关系的解耦。从而提高项目的扩展性和维护性。
2. 三种工厂模式（简单工厂模式、工厂方法模式、抽象工厂模式）。
3. 设计模式的依赖抽象原则：

> 创建对象实例时，不要直接new，而是把这个new类的动作放在一个工厂的方法中，并返回。有的书上说，变量不要直接持有具体类的引用。

> 不要让类继承具体类，而是继承抽象类或者实现interface（接口）。

> 不要覆盖基类中已经实现的方法。

### 4.3 原型模式

* **克隆羊问题**

现在有一只羊tom，姓名为：tom，年龄为1，颜色为白色，请编写程序创建和tom羊属性完全相同的10只羊。

* **传统方式解决克隆羊问题**

1. 思路分析
2. 代码演示（src/com/example/prototype/Client.java）。

* **传统的方式的优缺点**

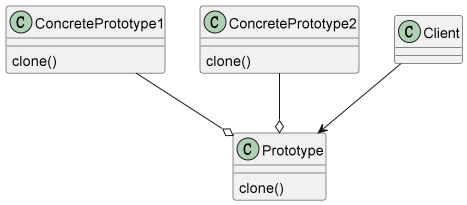
1. 优点是比较好理解，简单易操作。
2. 在创建新的对象时，总是需要重新获取原始对象的属性，如果创建的对象比较复杂时，效率较低。
3. 总是需要重新初始化对象，而不是动态地获得对象运行时的状态，不够灵活。
4. 改进的思路分析。

思路：Java和Object类是所有类的根类，Object类提供了一个clone()方法，该方法可以将一个Java对象复制一份，但是需要实现clone的Java类必须要实现一个接口Cloneable，该接口表示该类能够复制且具有复制的能力=>原型模式。

* **原型模式-基本介绍**

1. 原型模式（Prototype模式）是指：用原型实例指定创建对象的种类，并且通过拷贝这些原型，创建新的对象。
2. 原型模式是一种创建型设计模式，允许一个对象再创建另一个可定制的对象，无需知道如何创建的细节。
3. 工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型模式对象拷贝他们自己来实施创建，即对象.clone()。
4. 形象的理解：孙大圣拔出猴毛，变出其他的大圣。

* **原型模式原理结构图-uml类图**



* **原理结构图说明：**

1. Prototype：原型类：声明一个克隆自己的接口。
2. ConcreatePrototype：具体的原型类，实现一个克隆自己的操作。
3. Client：让一个原型对象克隆自己，从而创建一个新的对象（属性一样。）

* **原型模式解决克隆羊的应用实例**

使用**原型模式**改进传统方式，让程序具有更高的效率和扩展性。

代码（src/com/example/prototype/improve/Client.java）。

* **原型模式在Spring框架中源码分析**

1. Spring中原型bean的创建，就是原型模式的应用。
2. 代码分析+Debug源码。





* **深入讨论-浅拷贝和深拷贝**

**浅拷贝介绍：**

1. 对于数据类型是基本数据类型的成员变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象。
2. 对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组、某个类对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值。
3. 前面我们克隆羊就是浅拷贝。
4. 浅拷贝是使用默认的clone()方法来实现。

sheep = (Sheep) super.clone();

**深拷贝基本介绍**

1. 复制对象的所有基本类型的成员变量值。
2. 为所有引用数据类型的成员变量申请存储空间，并复制每个引用数据类型成员变量所引用的对象，直到该对象可达的所有对象。也就是说，**对象进行深拷贝要对整个对象（包括对象的引用类型）进行拷贝**。
3. 深拷贝实现方式1：重写**clone**方法来实现深拷贝。
4. 深拷贝实现方式2：通过**对象序列化**实现深拷贝（推荐）。
5. 代码实现（src/com/example/prototype/deepclone）。

* **原型模式的注意事项和细节**

1. 创建新的对象比较复杂时，可以利用原型模式简化**对象的创建过程，同时也能够提高效率**。
2. 不用重新初始化对象，而是**动态地获得对象运行时的状态**。
3. 如果原始对象发生变化（增加或减少属性），其他克隆对象也会发生相应的变化，无需修改代码。
4. 在实现深克隆的时候可能需要比较复杂的代码。
5. **缺点**：需要为每一个类配备一个克隆方法，这对全新的类来说不是很难，但对已有的类进行改造时，需要修改其源代码，违背了ocp原则，这点请同学们注意。

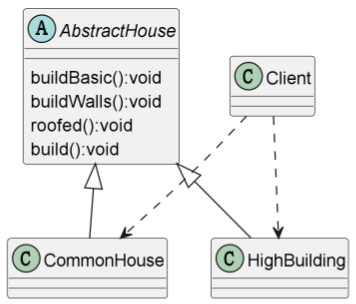
### 4.4 建造者模式

* **盖房项目需求**

1. 需要建房子：这一过程为打桩、砌墙、封顶。
2. 房子有各种各样的，比如普通楼房、高楼、别墅，各种房子的过程虽然一样，但是要求不要相同的。
3. 请编写程序，完成需求。

* **传统方式解决盖房需求**

1. 思路分析（图解）。



1. 代码演示（src/com/example/builder/Client.java）。

* **传统方式解决盖房需求问题分析**

1. 有点是比较好理解，简单易操作。
2. 设计的程序结构，过于简单，没有设计缓存对象，程序的扩展和维护不好，也就是说，这种设计方案，把产品（即：房子）和创建产品的过程（即：建房子流程）封装在一起，耦合性增强了。
3. 解决方案：将产品和产品建造过程解耦 => 建造者模式。

* **建造者模式基本介绍**

1. **建造者模式**（Builder Pattern）又叫生成器模式，是一种对象构建。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法构建出不同表现（属性）的对象。
2. **建造者模式**是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建他们，用户不需要知道内部的具体构建细节。

* **建造者模式的四个角色**

1. **Product（产品角色）**：一个具体的产品对象。
2. **Builder（抽象建造者）**：创建一个Product对象的各个部件指定的接口/抽象类。
3. **ConcreteBuilder（具体建造者）**：实现接口，构建和装配各个部件。
4. **Director（指挥者）**：构建一个使用Builder接口的对象，它主要用于创建一个复杂的对象。它主要有两个作用，一是：隔离了客户与对象的生产过程，二是：复杂控制产品对象的生产过程。

