设计模式

为什么要学设计模式

1) 软件工程中，**设计模式**（design pattern）是对软件设计中**普遍存在（反复出现）**的各种问题，所提出的**解决方案**。这个术语是由埃里希·伽玛（Erich Gamma）等人在 1990 年代从建筑设计领域引入到计算机科学的

2) 大厦 VS 简易房

3) 拿实际工作经历来说, 当一个项目开发完后，如果**客户提出增新功能**，怎么办?。（可扩展性,使用设计模式，软件具有很好的扩展性）

4) 如果项目开发完后，原来程序员离职，你接手维护该项目怎么办? (**维护性**[可读性、规范性])

5) 目前程序员门槛越来越高，一线 IT 公司(大厂)，都会问你在实际项目中**使用过什么设计模式，怎样使用的，解决了什么问题**。

6) **设计模式在软件中哪里**？面向对象(oo)=>功能模块[设计模式+算法(数据结构)]=>框架[使用到多种设计模式]=>架构 [服务器集群]

7) 如果想成为合格软件**工程师**，那就花时间来研究下设计模式是非常必要的.

- **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，设计模式是为了让**程序(软件)**，具有更好

1) 代码重用性 (即：相同功能的代码，不用多次编写)
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，称为可维护
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++》就曾经说过：C++老手和 C++新手的区别就是前者手背上有很多伤疤

一、设计模式的七大原则

1.1 单一职责原则（鄙人称之为专一原则一）

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负责两个不同职责两个不同职责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

```java
// 交通工具类
// 方式1
// 1. 生方式1 的run方法中，违反了单一职责原则
// 2. 解决的方案非常的简单，根据交通工具运行方法不同，分解成不同类即可
class Vehicle {
    public void run(String vehicle) {
        System.out.println(vehicle + " 在公路上运行....");
    }
}
```

```java
//方案2的分析
//1. 遵守单一职责原则
//2. 但是这样做的改动很大，即将类分解，同时修改客户端
//3. 改进：直接修改Vehicle 类，改动的代码会比较少=>方案3

class RoadVehicle {
    public void run(String vehicle) {
        System.out.println(vehicle + "公路运行");
    }
}

class AirVehicle {
    public void run(String vehicle) {
        System.out.println(vehicle + "天空运行");
    }
}

class WaterVehicle {
    public void run(String vehicle) {
        System.out.println(vehicle + "水中运行");
    }
}
```

```java
//方式3的分析
//1. 这种修改方法没有对原来的类做大的修改，只是增加方法
//2. 这里虽然没有在类这个级别上遵守单一职责原则，但是在方法级别上，仍然是遵守单一职责
class Vehicle2 {
    public void run(String vehicle) {
        //处理

        System.out.println(vehicle + " 在公路上运行....");

    }

    public void runAir(String vehicle) {
        System.out.println(vehicle + " 在天空上运行....");
    }

    public void runWater(String vehicle) {
        System.out.println(vehicle + " 在水中行....");
    }
}
```

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。 **一项职责不等于一个方法**
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职责原则

1.2 接口隔离原则（鄙人称之为专一原则二）

1.3 依赖倒转原则（鄙人称之为接口使用原则）

基本介绍

依赖倒转原则 (Dependence Inversion)是指：

1. 高层模块不应该依赖低层模块，二者都应该依赖其抽象
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：相对于细节的多变性，抽象东西要稳定多。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，细节就具体实现类
5. 使用接口或抽象类的目是制定好规范，而不涉及任何具体操作把展现细节任务交给他们的实现类去完成

应用实例

```java
class Email {
    public String getInfo() {
        return "电子邮件信息: hello,world";
    }
}
```

其实就是面向接口编程

```java
//定义接口
interface IReceiver {
    public String getInfo();
}

class Email implements IReceiver {
    public String getInfo() {
        return "电子邮件信息: hello,world";
    }
}

//增加微信
class WeiXin implements IReceiver {
    public String getInfo() {
        return "微信信息: hello,ok";
    }
}

//方式2
class Person {
    //这里我们是对接口的依赖
    public void receive(IReceiver receiver ) {
        System.out.println(receiver.getInfo());
    }
}
```

```java
//完成Person接收消息的功能
//方式1分析
//1. 简单，比较容易想到
//2. 如果我们获取的对象是 微信，短信等等，则新增类，同时Perons也要增加相应的接收方法
//3. 解决思路：引入一个抽象的接口IReceiver， 表示接收者， 这样Person类与接口IReceiver发生依赖
//   因为Email, WeiXin 等等属于接收的范围，他们各自实现IReceiver 接口就ok， 这样我们就符号依赖倒转原则
class Person {
    public void receive(Email email ) {
        System.out.println(email.getInfo());
    }
}
```

依赖(使用)关系的三种传递方式

接口传递方式

很标准的接口编程方式

```java
// 方式1: 通过接口传递实现依赖
// 开关的接口
// interface IOpenAndClose {
// public void open(ITV tv); //抽象方法,接收接口
// }
//
// interface ITV { //ITV接口
// public void play();
// }
//
// class ChangHong implements ITV {
//
//  @Override
//  public void play() {
//      // TODO Auto-generated method stub
//      System.out.println("长虹电视机，打开");
//  }
//
// }
//// 实现接口
// class OpenAndClose implements IOpenAndClose{
// public void open(ITV tv){
// tv.play();
// }
// }
```

构造方法传递方式

```
// 方式2: 通过构造方法依赖传递
// interface IOpenAndClose {
// public void open(); //抽象方法
// }
// interface ITV { //ITV接口
// public void play();
// }
// class OpenAndClose implements IOpenAndClose{
// public ITV tv; //成员
// public OpenAndClose(ITV tv){ //构造器
// this.tv = tv;
// }
// public void open(){
// this.tv.play();
// }
// }
```

其实说到底就是多态的那一套东西

类中定义一个接口成员变量，并生成一个以该接口为参数的构造器

setter方法传递方式

```
// 方式3 , 通过setter方法传递
interface IOpenAndClose {
    public void open(); // 抽象方法

    public void setTv(ITV tv);
}

interface ITV { // ITV接口
    public void play();
}

class OpenAndClose implements IOpenAndClose {
    private ITV tv;

    public void setTv(ITV tv) {
        this.tv = tv;
    }

    public void open() {
        this.tv.play();
    }
}

class ChangHong implements ITV {

    @Override
    public void play() {
        // TODO Auto-generated method stub
        System.out.println("长虹电视机，打开");
    }
```

其实也可以联想spring框架的依赖注入

注意事项与细节

依赖倒转原则的注意事项和细节 依赖倒转原则的注意事项和细节

1. 低层模块尽量都要有抽象类或接口，或者都有，程序稳定性更好 .
2. 变量的声明类型尽是抽象或接口, 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

1.4 里氏替换原则（鄙人称之为继承使用原则）

OO 中的继承性思考和说明中的继承性思考和说明:

1. 继承包含这样一层义：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循这些契约(遵循契约 在此也就是不覆盖父类的方法)，但是如果对些已经实现的方法任意修改(也就是覆盖)，就会对整个继承体系造成破坏 。
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侵 入性，程序的可移植降低，增加对象间耦合。如果一个类被其他所继承则当这个类需要修改时，必须考虑到所有的子类，并且父后涉及类的功能都有可产生故 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1988 年，由麻省理工学院的以为姓里 的 女士提出。
2. 如果对每个类型 为T1 的对象 o1 ，都有类型为 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使其子对象 。
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的方 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。 .

应用介绍

```java
// A类
class A {
    // 返回两个数的差
    public int func1(int num1, int num2) {
        return num1 - num2;
    }
}

// B类继承了A
// 增加了一个新功能：完成两个数相加，然后和9求和
class B extends A {
    //这里，重写了A类的方法， 可能是无意识
    public int func1(int a, int b) {
        return a + b;
    }

    public int func2(int a, int b) {
        return func1(a, b) + 9;
    }
}
```

尽量不要重写父类的方法

```java
//创建一个更加基础的基类
class Base {
    //把更加基础的方法和成员写到Base类
}

// A类
class A extends Base {
    // 返回两个数的差
    public int func1(int num1, int num2) {
        return num1 - num2;
    }
}

// B类继承了A
// 增加了一个新功能：完成两个数相加，然后和9求和
class B extends Base {
    //如果B需要使用A类的方法，使用组合关系
    private A a = new A();

    //这里，重写了A类的方法， 可能是无意识
    public int func1(int a, int b) {
        return a + b;
    }

    public int func2(int a, int b) {
        return func1(a, b) + 9;
    }

    //我们仍然想使用A的方法
    public int func3(int a, int b) {
        return this.a.func1(a, b);
    }
}
```

1.5 开闭原则（鄙人称之为不修改但扩展原则）

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，模块和函数应该对扩展开放(对提供方)，对修改关闭(对使用方)。用抽象构建框架，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. 编程中遵循其它原则，以及使用设计模式的目就是遵循开闭原则。

应用实例

1.6 迪米特法则（鄙人称之为非局部变量原则）

基本介绍

1. 一个对象应该其他保持最少的了解
2. 类与类关系越密切，耦合度大
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 知道越少好。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：只与直接朋友通信
5. **直接的朋友**：每个对象都会与其他象有耦合关系， 只要两个对象之间**有耦合**我们就说这两个对象之间是**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、返回值**的类为直接朋友，而出现在局部变量中的类不是直接朋友。也就说，陌生的类最好不要以局部变量的形式出现在类内部。

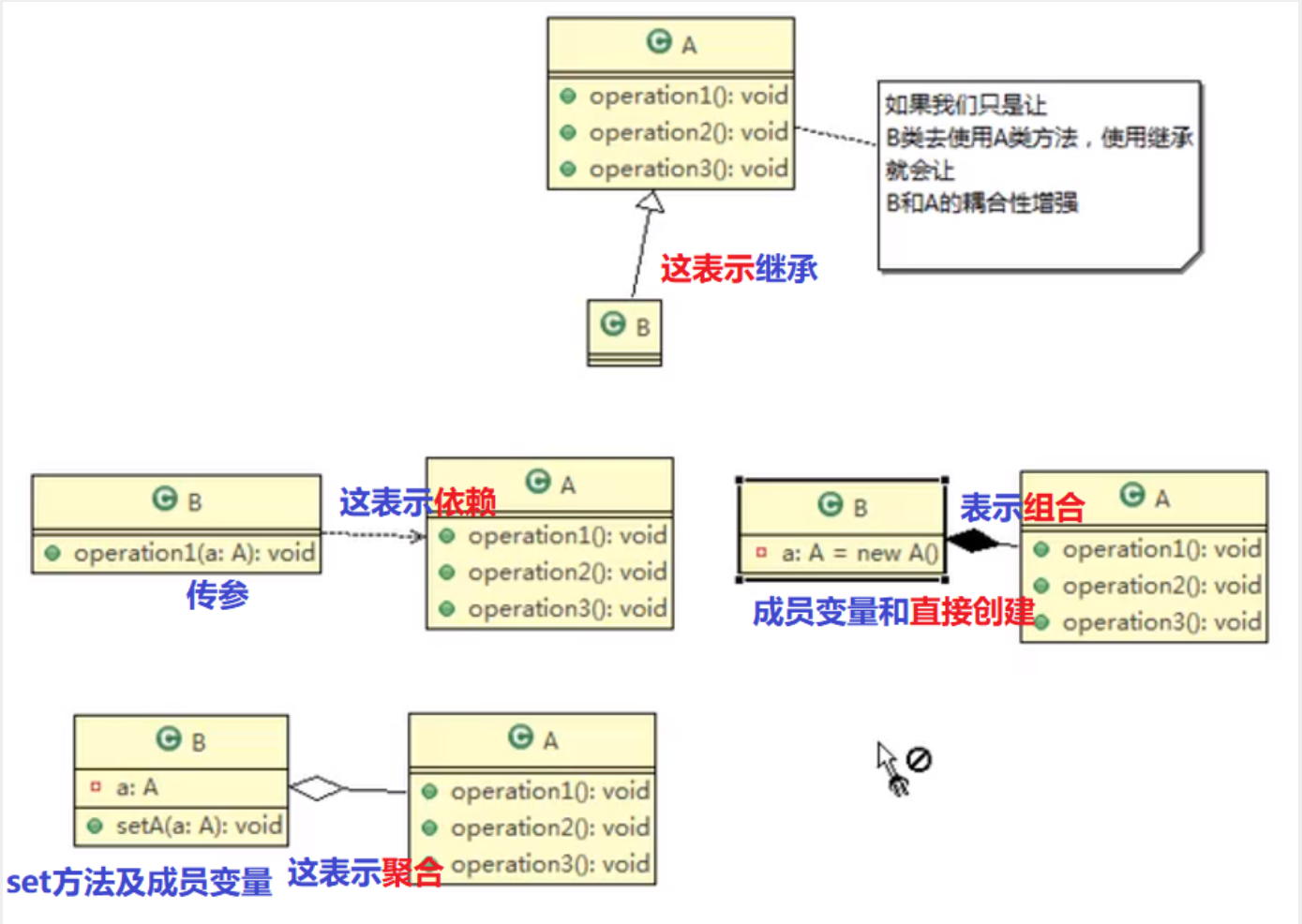
应用实例

迪 米特法则 的 核心是降低类之间的耦合

但 是注意：由 于每个类都减少了不必要的依赖，因此迪米特 则只是要求降低类间对象间耦合关系， 并不是要求完全没有依赖关系

1.7 合成复用原则（鄙人称之为最好合成与聚合原则）

原则是尽量使用合成、聚合的方式，而不是使用继承

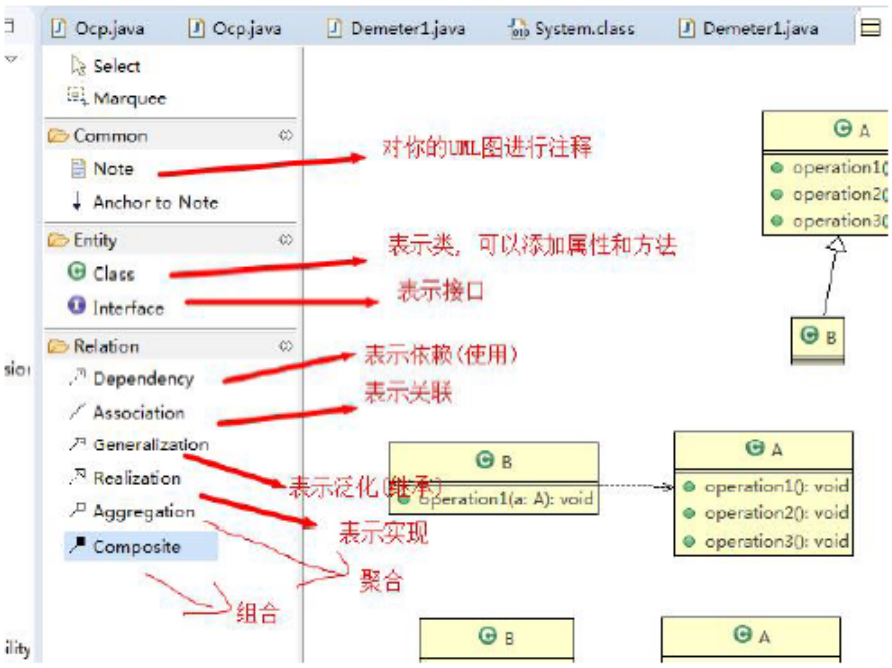


二、UML类图

2.1 UML类图介绍

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



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

Eclipse 安装 UML插件(AmaterasUML).zip

AmaterasUML_1.3.4.rar

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(use case)
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

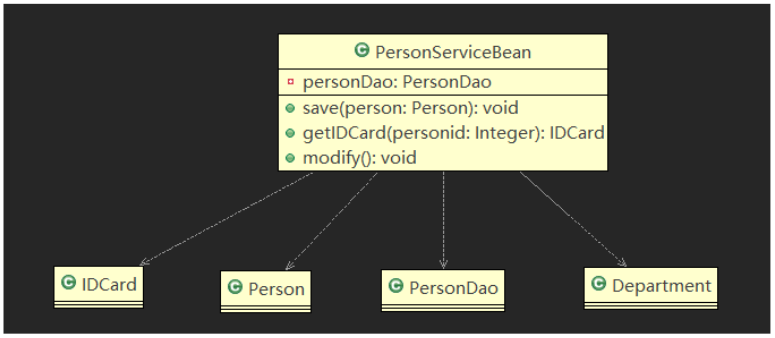
2.2 类的依赖、泛化和实现

2.2.1 依赖：类中用到了对方，就存在依赖关系

```
public class PersonServiceBean {
    private PersonDao personDao;//类
    public void save(Person person){}
                public IDCard getIDCard(Integer personid){}
    public void modify(){
        Department department = new Department();
                }
}
public class PersonDao{}
public class IDCard{}
public class Person{}
public class Department{}
```

无论以何种方式，都算用到了

注意标识

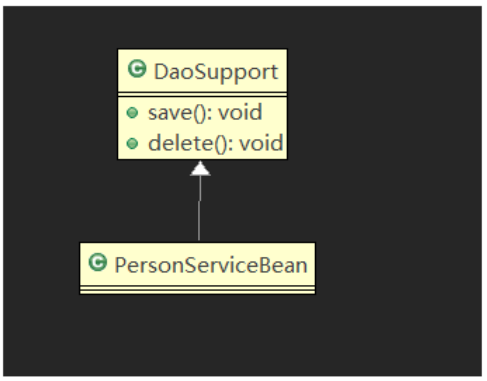


2.2.2 泛化：就是继承关系，是依赖关系的特例

```
public abstract class DaoSupport{
    public void save(Object entity){
    }
    public void delete(Object id){
    }
}

public class PersonServiceBean extends Daosupport{
}
```

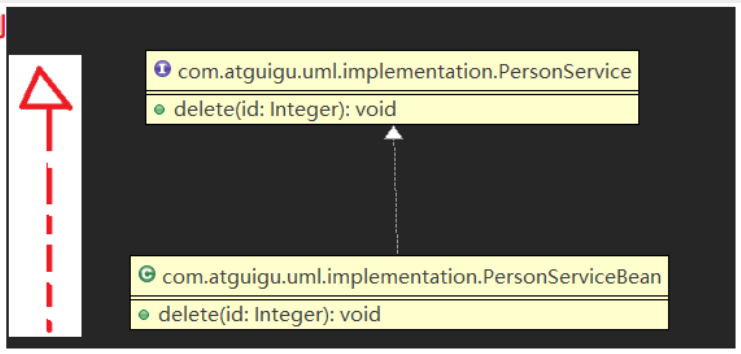
继承就是泛化



2.2.3 实现：类实现接口，同样是依赖关系的特例

类实现接口，同样也是依赖的特例

```
public interface PersonService {
public void delete(Interger id);
}
public class PersonServiceBean implements PersonService {
public void delete(Interger id){}
}
```



2.3 类的关联、聚合和组合

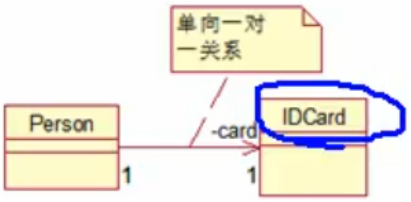
2.3.1 类的关联关系：也是依赖的一种特例

- 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个或者多个），“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{}
```



双向一对一关系

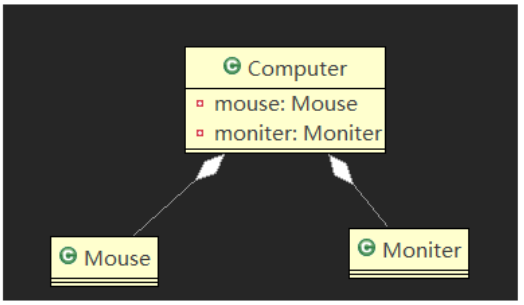
```
public class Person {
    private IDCard card;
}
public class IDCard{
    private Person person
```

2.3.2 类的聚合关系：整体与部分可以分离

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，使用带空心菱形的实线来表示：

```
3  public class Computer {
4      private Mouse mouse; //鼠标可以和computer分离
5      private Moniter moniter;//显示器可以和Computer分离
6      public void setMouse(Mouse mouse) {
7          this.mouse = mouse;
8      }
9      public void setMoniter(Moniter moniter) {
10         this.moniter = moniter;
11     }
12
13 }
14
```



2.3.3 类的组合关系：整体与部分不可分离

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

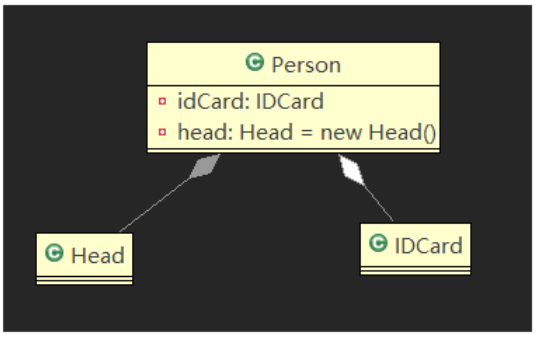
再看一个案例：在程序中我们定义实体：Person 与 IDCard、Head，那么 Head 和 Person 就是 组合，IDCard 和 Person 就是聚合。

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;
    private Head head = new Head();
}
public class IDCard{}
public class Head{}
```

显然，Person类对象创建，Head类必然创建

也就是不可分离



2.4 类图的六大关系总结

三、设计模式概述和分类

3.1 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.

3.2 分类

3.2.1 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

3.2.2 结构型模式

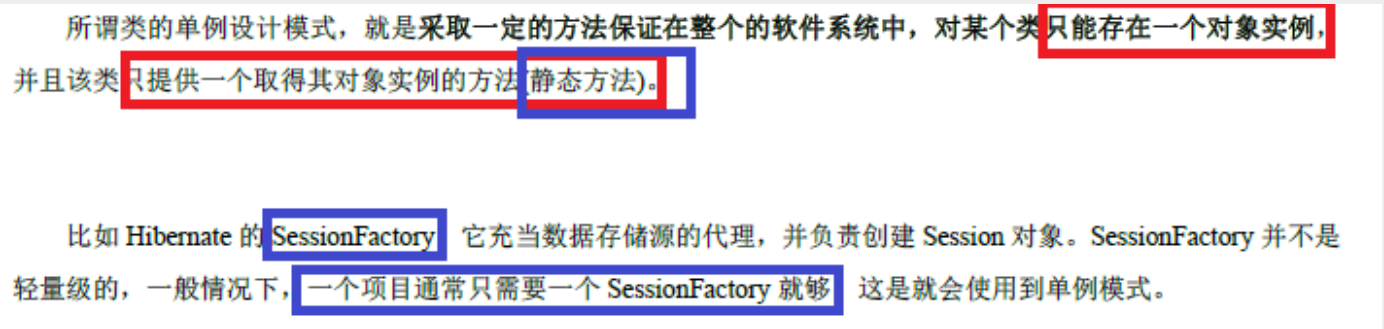
- 适配器模式
- 桥接模式
- 装饰模式
- 组合模式
- 外观模式
- 享元模式
- 代理模式

3.2.3 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
- 中介模式
- 备忘录模式
- 解释器模式
- 状态模式
- 策略模式
- 职责链模式

四、单例模式

4.1 单例模式介绍



4.2 单例设计模式的八种方法

4.3 两种饿汉式（类加载的时候就会创建对象，完成实例化）

因为类加载就创建，完成实例化，避免了线程同步的问题，因为是立即的，马上的，相当于原子的

4.3.1 饿汉式一（使用静态变量）：内部静态方法返回静态变量

```
//饿汉式(静态变量)

class Singleton {

    //1. 构造器私有化, 外部不能new
    private Singleton() {

    }

    //2.本类内部创建对象实例
    private final static Singleton instance = new Singleton();

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

优缺点说明：

1) 优点：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，没有达到 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 classloder 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，因此不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 就没有达到 lazy loading 的效果
4) 结论：这种单例模式可用，可能造成内存浪费

```
public class SingletonTest01 {

    public static void main(String[] args) {
        //测试
        Singleton instance = Singleton.getInstance();
        Singleton instance2 = Singleton.getInstance();
        System.out.println(instance == instance2); // true
        System.out.println("instance.hashCode=" + instance.hashCode());
        System.out.println("instance2.hashCode=" + instance2.hashCode());
    }

}
```

4.3.2 饿汉式二（使用静态代码块）：创建对象在静态代码块中，返回对象在静态方法中

```
class Singleton {

    //1. 构造器私有化, 外部不能new
    private Singleton() {

    }
        优缺点与饿汉式一相同

    //2.本类内部创建对象实例
    private  static Singleton instance;

    static { // 在静态代码块中，创建单例对象
        instance = new Singleton();
    }

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }

}
```

4.4 三种懒汉式（都不能用，要么效率太低，要么不安全，类加载的时候不会立即实例化）

4.4.1 懒汉式一：不同步，线程不安全（判断和创建过程没有加锁或者别的同步方法）

```
    public class SingletonTest03 {
        public static void main(String[] args) {
            System.out.println("懒汉式1 ，  线程不安全~");
            Singleton instance = Singleton.getInstance();
            Singleton instance2 = Singleton.getInstance();
            System.out.println(instance == instance2); // true
            System.out.println("instance.hashCode=" + instance.hashCode());
            System.out.println("instance2.hashCode=" + instance2.hashCode
());
        }
    }

    class Singleton {
        private static Singleton instance;

        private Singleton() {}

        //提供一个静态的公有方法，当使用到该方法时，才去创建 instance
        //即懒汉式
        public static Singleton getInstance() {
            if(instance == null) {
                instance = new Singleton();
            }
            return instance;
        }
}
```

起到了懒加载的作用
多线程下，**如果一个线程进入了if(instance == null)判断停止了**，还未来得及执行下去，那么其他线程进行判断，将会产生多个实例
实际开发中，不要使用这种方式

4.4.2 懒汉式二：同步方法，线程安全（判断过程同步了创建的方法）

```
// 懒汉式(线程安全，同步方法)
    class Singleton {
        private static Singleton instance;

        private Singleton() {}

        //提供一个静态的公有方法，加入同步处理的代码，解决线程安全问题
        //即懒汉式
```

```
        public static synchronized Singleton getInstance() {
            if(instance == null) {
                instance = new Singleton();
            }
            return instance;
        }
    }
```

优缺点说明

解决了线 程不安全问题

效率太低了，每个线程在想获得类的实例时候，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**

结论： 在实际开发中， 不推荐使用这种方 式

4.4.3 懒汉式三： 同步代码块，线程安全，同样效率低

```
// 懒汉式(线程安全，同步方法)
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}


    public static  Singleton getInstance() {

                if(instance == null) {
                    synchronized (Singleton.class) {
                    instance = new Singleton();
                    }
                }

        return instance;
    }
}
```

4.5 双重检查（推荐使用）

```
public class Singleton {
    private static \*volatile*\ Singleton uniqueSingleton;

    private Singleton() {
    }

    public Singleton getInstance() {
        if (null == uniqueSingleton) {
            synchronized (Singleton.class) {
                if (null == uniqueSingleton) {
                    uniqueSingleton = new Singleton();   // error
                }
            }
        }
        return uniqueSingleton;
    }
}
```

如果这样写，运行顺序就成了：

- 检查变量是否被初始化(不去获得锁)，如果已被初始化则立即返回。
- 获取锁。
- 再次检查变量是否已经被初始化，如果还没被初始化就初始化一个对象。 执行双重检查是因为，如果多个线程同时了通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么**剩余通过了第一次检查的线程就不会再去实例化对象**。

这样，除了初始化的时候会出现加锁的情况，后续的所有调用都会避免加锁而直接返回，解决了性能消耗的问题。

Double Check 概 念 是 多 线程开 发中常使用到的， 如代码中所示，我们进行了两次 if (singleton == null) 检查，这样就可以保证线程安全了 。

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，也避免的反复进行方法同步

线程安全；延迟加载；效率较高

结论：在实际开发中，推荐使用这种单例设计模式

4.6 静态内部类（推荐使用）

```
// 静态内部类完成， 推荐使用
class Singleton {
    private static volatile Singleton instance;
```

```
    //构造器私有化
    private Singleton() {}

    //写一个静态内部类,该类中有一个静态属性 Singleton
    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }

    //提供一个静态的公有方法，直接返回SingletonInstance.INSTANCE

    public static synchronized Singleton getInstance() {

        return SingletonInstance.INSTANCE;
    }
}
```

静态内部类不会主动加载，又不是静态代码块

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 。

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 。

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的安全性，在类进行初始化时，别的线程是无法进入的 。

优 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

结 论：推荐使用

4.7 枚举（推荐使用）

```
public class SingletonTest08 {
    public static void main(String[] args) {
        Singleton instance = Singleton.INSTANCE;
        Singleton instance2 = Singleton.INSTANCE;
        System.out.println(instance == instance2);

        System.out.println(instance.hashCode());
        System.out.println(instance2.hashCode());

        instance.sayOK();
    }
```

```
}

//使用枚举，可以实现单例，推荐
enum Singleton {
    INSTANCE; //属性
    public void sayOK() {
        System.out.println("ok~");
    }
}
```

优缺点说明：

1) 这借助JDK1.5中添加的枚举来实现单例模式。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新的对象。
2) 这种方式是Effective Java作者Josh Bloch 提倡的方式
3) 结论：推荐使用

4.8 单例模式总结

单例模式注意事项和细节说明 单例模式 注意事项和细节说明

单例模式保证了 系 统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统性能

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

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 的对象 比如数据源、session 工厂 等

五、工厂模式