设计模式

讲师：高淇

设计模式GOF23

- **将设计者的思维融入大家的学习和工作中，更高层次的思考！**

- **创建型模式：**
  - 单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式。
- **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。
- **行为型模式：**
  - 模版方法模式、命令模式、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、状态模式、策略模式、职责链模式、访问者模式。

单例模式

- 核心作用：
  - 保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。

- 常见应用场景：
  - Windows的Task Manager（任务管理器）就是很典型的单例模式
  - windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。
  - 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，每次new一个对象去读取。
  - 网站的计数器，一般也是采用单例模式实现，否则难以同步。
  - 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
  - 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
  - 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。
  - Application 也是单例的典型应用（Servlet编程中会涉及到）
  - 在Spring中，每个Bean默认就是单例的，这样做的优点是Spring容器可以管理
  - 在servlet编程中，每个Servlet也是单例
  - 在spring MVC框架/struts1框架中，控制器对象也是单例

单例模式

- 单例模式的优点：
  - 由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决
  - 单例模式可以在系统设置全局的访问点，优化环共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理

- 常见的五种单例模式实现方式：
  - 主要：
    - 饿汉式（线程安全，调用效率高。 但是，不能延时加载。 ）
    - 懒汉式（线程安全，调用效率不高。 但是，可以延时加载。 ）
  - 其他：
    - 双重检测锁式（由于JVM底层内部模型原因，偶尔会出问题。不建议使用）
    - 静态内部类式(线程安全，调用效率高。 但是，可以延时加载)
    - 枚举单例(线程安全，调用效率高，不能延时加载)

单例模式

- 饿汉式实现（单例对象立即加载）

```
public class SingletonDemo02 {
        private static  /*final*/   SingletonDemo02 s = new SingletonDemo02();

        private SingletonDemo02(){} //私有化构造器

        public static /*synchronized*/ SingletonDemo02 getInstance(){
                return s;
        }
}
```

```
public class Client {
        public static void main(String[] args) {
                SingletonDemo02 s = SingletonDemo02.getInstance();
                SingletonDemo02 s2 = SingletonDemo02.getInstance();
                System.out.println(s==s2); //结果为true
        }
}
```

- 饿汉式单例模式代码中，static变量会在类装载时初始化，此时也不会涉及多个线程对象访问该对象的问题。虚拟机保证只会装载一次该类，肯定不会发生并发访问的问题。因此，可以省略synchronized关键字。

- 问题：如果只是加载本类，而不是要调用getInstance()，甚至永远没有调用，则会造成资源浪费！

单例模式

- 懒汉式实现（单例对象延迟加载）

```
public class SingletonDemo01 {
    private static SingletonDemo01 s;

    private SingletonDemo01(){} //私有化构造器


    public static synchronized SingletonDemo01 getInstance(){
        if(s==null){
            s = new SingletonDemo01();
        }
        return s;
    }

}
```

- 要点：
  - lazy load!　　延迟加载，　懒加载！　真正用的时候才加载！
- 问题：
  - 资源利用率高了。但是，每次调用getInstance()方法都要同步，并发效率较低。

单例模式

- 双重检测锁实现

```
public class SingletonDemo03 {

  private static SingletonDemo03 instance = null;

  public static SingletonDemo03 getInstance() {
    if (instance == null) {
      SingletonDemo03 sc;
      synchronized (SingletonDemo03.class) {
        sc = instance;
        if (sc == null) {
          synchronized (SingletonDemo03.class) {
            if(sc == null) {
              sc = new SingletonDemo03();
            }
          }
          instance = sc;
        }
      }
    }
    return instance;
  }

  private SingletonDemo03() {

  }

}
```

- 这个模式将同步内容下方到if内部，提高了执行的效率，不必每次获取对象时都进行同步，只有第一次才同步，创建了以后就没必要了。

- 问题：
- 由于编译器优化原因和JVM底层内部模型原因，偶尔会出问题。不建议使用。

单例模式

- 静态内部类实现方式(也是一种懒加载方式)

```
public class SingletonDemo04 {

        private static class SingletonClassInstance {
                private static final SingletonDemo04 instance = new SingletonDemo04();
        }

         public static SingletonDemo04 getInstance() {
                  return SingletonClassInstance.instance;
         }

         private SingletonDemo04() {
         }
}
```

- 要点：
  - 外部类没有static属性，则不会像饿汉式那样立即加载对象。
  - 只有真正调用getInstance(),才会加载静态内部类。加载类时是线程 安全的。 instance是static final类型，保证了内存中只有这样一个实例存在，而且只能被赋值一次，从而保证了线程安全性.
  - 兼备了并发高效调用和延迟加载的优势！

单例模式

- 问题：
  - 反射可以破解上面几种(不包含枚举式)实现方式！（可以在构造方法中手动抛出异常控制）
  - 反序列化可以破解上面几种((不包含枚举式))实现方式！
    - 可以通过定义readResolve()防止获得不同对象。
      - 反序列化时，如果对象所在类定义了readResolve()，（实际是一种回调），定义返回哪个对象。

```
public class SingletonDemo01  implements Serializable {
        private static SingletonDemo01 s;

        private  SingletonDemo01() throws Exception{
                if(s!=null){
                        throw new Exception("只能创建一个对象");
//通过手动抛出异常，避免通过反射创建多个单例对象！
                }
        } //私有化构造器


        public static synchronized SingletonDemo01 getInstance() throws Exception{
                if(s==null){
                        s = new SingletonDemo01();
                }
                return s;
        }

//反序列化时，如果对象所在类定义了readResolve()，（实际是一种回调），定义返回哪个对象。
        private Object readResolve() throws ObjectStreamException {
                return s;
        }
}
```

单例模式

- 使用枚举实现单例模式

```
public enum SingletonDemo05 {
    /**
    * 定义一个枚举的元素，它就代表了Singleton的一个实例。
    */
   INSTANCE;
   /**
    * 单例可以有自己的操作
    */
   public void singletonOperation(){
      //功能处理
   }
}
```

```
public static void main(String[] args) {
                    SingletonDemo05  sd = SingletonDemo05.INSTANCE;
                    SingletonDemo05  sd2 = SingletonDemo05.INSTANCE;
                    System.out.println(sd==sd2);
}
```

- 优点：
  - 实现简单
  - 枚举本身就是单例模式。由JVM从根本上提供保障！避免通过反射和反序列化的漏洞！
- 缺点：
  - 无延迟加载

单例模式

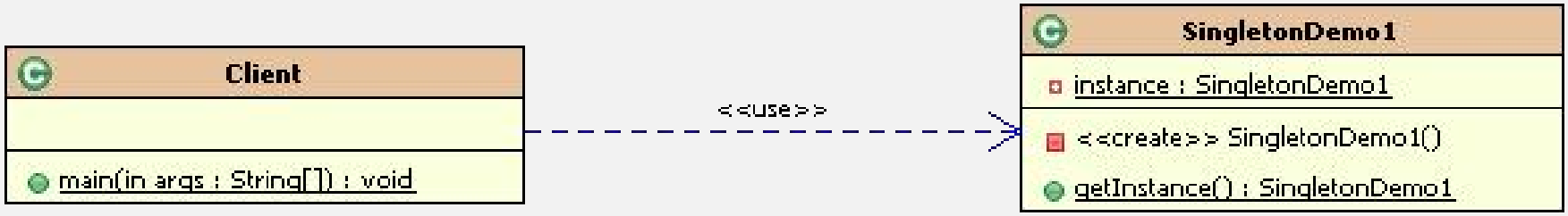
- 常见的五种单例模式在多线程环境下的效率测试
  - 大家只要关注相对值即可。在不同的环境下不同的程序测得值完全不一样

饿汉式	**22ms**
懒汉式	636ms
静态内部类式	28ms
枚举式	32ms
双重检查锁式	65ms

- CountDownLatch
  - 同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
    - countDown() 当前线程调此方法，则计数减一(建议放在 finally里执行)
    - await()， 调用此方法会一直阻塞当前线程，直到计时器的值为0

单例模式

- 使用myeclipse的UML插件画出类图
  - 大家也可以使用：rational rose 、 metamill等。



单例模式总结

- 常见的五种单例模式实现方式
  - 主要：
    - 饿汉式（线程安全，调用效率高。 但是，不能延时加载。 ）
    - 懒汉式（线程安全，调用效率不高。 但是，可以延时加载。 ）
  - 其他：
    - 双重检测锁式（由于JVM底层内部模型原因，偶尔会出问题。**不建议使用**）
    - 静态内部类式(线程安全，调用效率高。 但是，可以延时加载)
    - 枚举式(线程安全，调用效率高，不能延时加载。并且可以天然的防止反射和反序列化漏洞！)
- 如何选用?
  - 单例对象 占用 资源 少，不需要 延时加载：
    - 枚举式 好于 饿汉式
  - 单例对象 占用 资源 大，需要 延时加载：
    - 静态内部类式 好于 懒汉式

工厂模式

- 工厂模式：
  - 实现了创建者和调用者的分离。
  - 详细分类：
    - 简单工厂模式
    - 工厂方法模式
    - 抽象工厂模式

- 面向对象设计的基本原则：
  - OCP（开闭原则，Open-Closed Principle）：一个软件的实体应当对扩展开放，对修改关闭。
    DIP（依赖倒转原则，Dependence Inversion Principle）：要针对接口编程，不要针对实现编程。
    LoD（迪米特法则，Law of Demeter）：只与你直接的朋友通信，而避免和陌生人通信。

工厂模式

- 核心本质：
  - 实例化对象，用工厂方法代替new操作。
  - 将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

- 工厂模式：
  - 简单工厂模式
    - 用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
  - 工厂方法模式
    - 用来生产同一等级结构中的固定产品。（支持增加任意产品）
  - 抽象工厂模式
    - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）

工厂模式

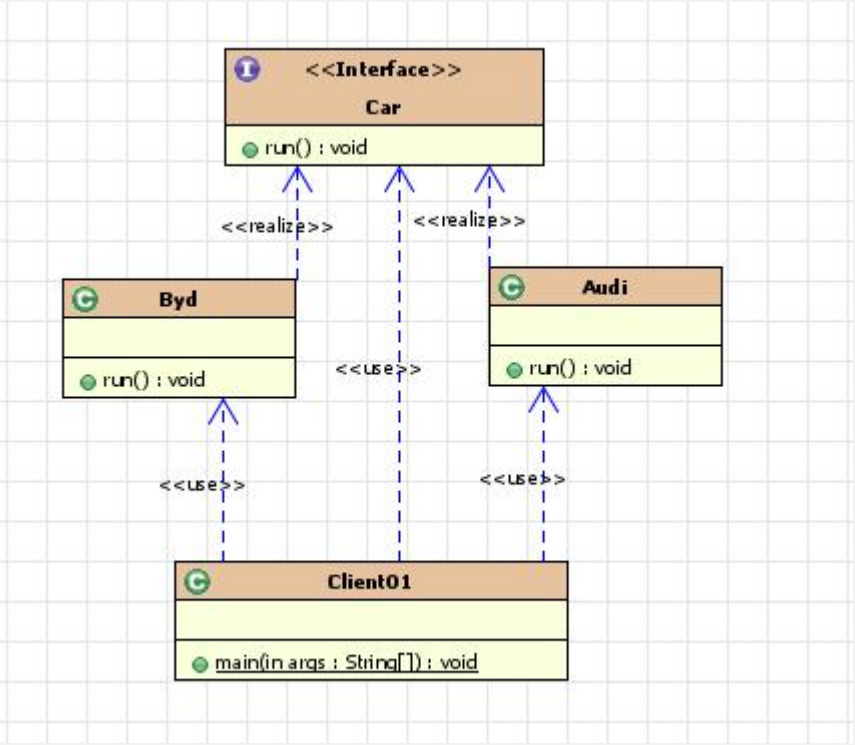
- 不使用简单工厂的情况

```
public class Client01 {    //调用者

public static void main(String[] args) {
Car c1 = new Audi();
Car c2 = new Byd();

c1.run();
c2.run();

}
}
```



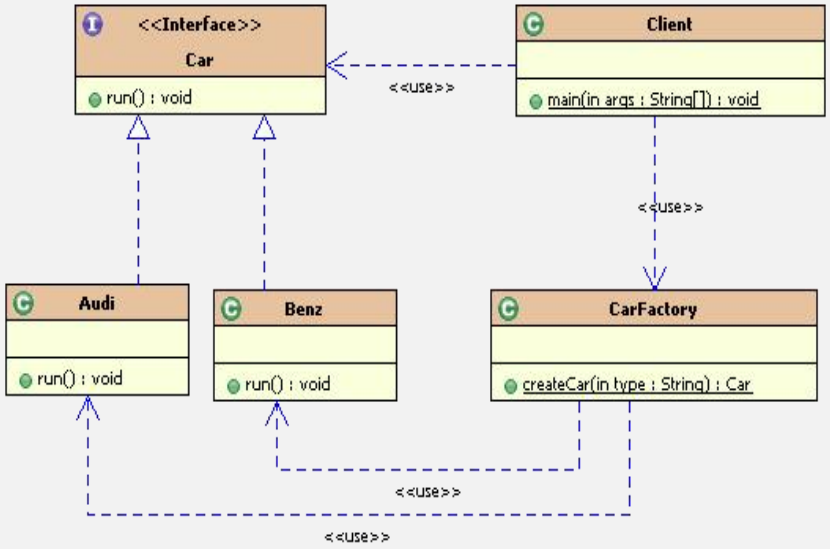
简单工厂模式

- 要点：
  - 简单工厂模式也叫静态工厂模式，就是工厂类一般是使用静态方法，通过接收的参数的不同来返回不同的对象实例。
  - 对于增加新产品无能为力！不修改代码的话，是无法扩展的。

```
package com.bjsxt.simpleFactory;
public class CarFactory {
            public static Car createCar(String type){
                        Car c = null;
                        if("奥迪".equals(type)){
                                    c  = new Audi();
                        }else if("奔驰".equals(type)){
                                    c = new Benz();
                        }
                        return c;
            }
}
```

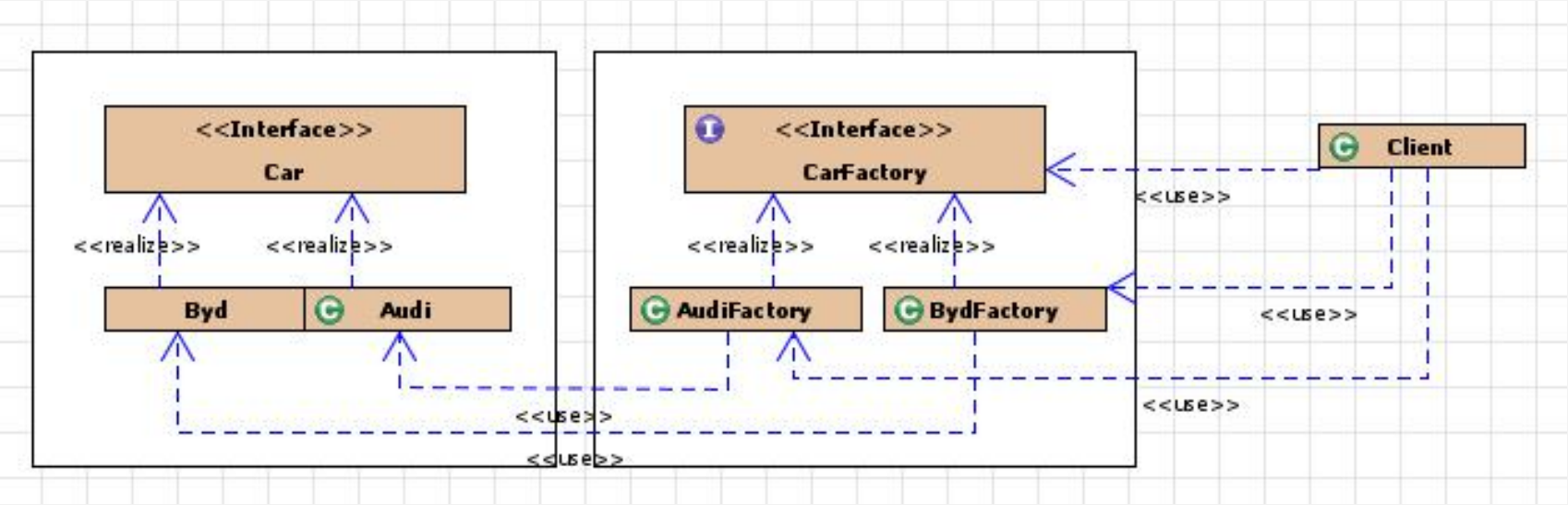
```
public class CarFactory {
            public static Car createAudi(){
                        return  new Audi();
            }
            public static Car createBenz(){
                        return  new Benz();
            }

}
```



工厂方法模式

- 工厂方法模式要点：
  - 为了避免简单工厂模式的缺点，不完全满足OCP。
  - 工厂方法模式和简单工厂模式最大的不同在于，简单工厂模式只有一个（对于一个项目或者一个独立模块而言）工厂类，而工厂方法模式有一组实现了相同接口的工厂类。

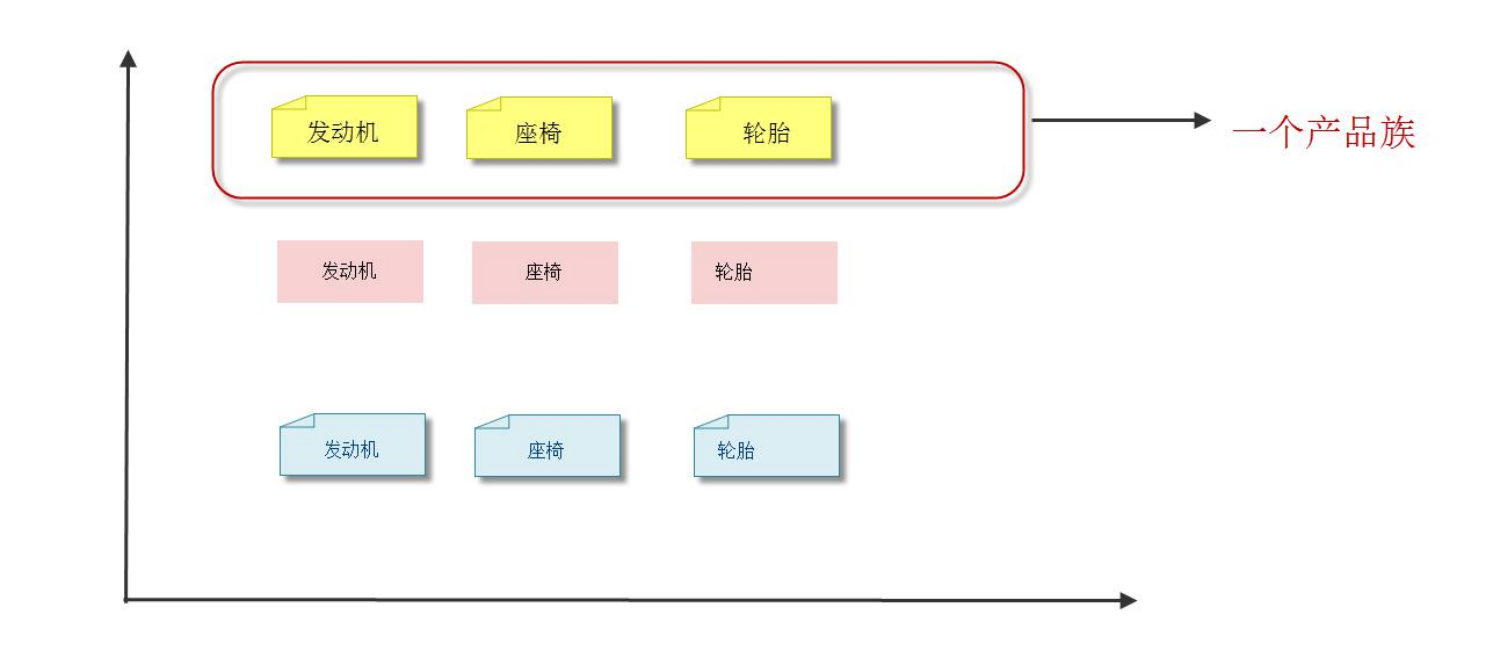


工厂模式

- 简单工厂模式和工厂方法模式PK:
  - **结构复杂度**
    从这个角度比较，显然简单工厂模式要占优。简单工厂模式只需一个工厂类，而工厂方法模式的工厂类随着产品类个数增加而增加，这无疑会使类的个数越来越多，从而增加了结构的复杂程度。
  - **代码复杂度**
    代码复杂度和结构复杂度是一对矛盾，既然简单工厂模式在结构方面相对简洁，那么它在代码方面肯定是比工厂方法模式复杂的了。简单工厂模式的工厂类随着产品类的增加需要增加很多方法（或代码），而工厂方法模式每个具体工厂类只完成单一任务，代码简洁。
  - **客户端编程难度**
    工厂方法模式虽然在工厂类结构中引入了接口从而满足了OCP，但是在客户端编码中需要对工厂类进行实例化。而简单工厂模式的工厂类是个静态类，在客户端无需实例化，这无疑是个吸引人的优点。
  - **管理上的难度**
    这是个关键的问题。
    我们先谈扩展。众所周知，工厂方法模式完全满足OCP，即它有非常良好的扩展性。那是否就说明了简单工厂模式就没有扩展性呢？答案是否定的。简单工厂模式同样具备良好的扩展性——扩展的时候仅需要修改少量的代码（修改工厂类的代码）就可以满足扩展性的要求了。尽管这没有完全满足OCP，但我们不需要太拘泥于设计理论，要知道，sun提供的java官方工具包中也有想到多没有满足OCP的例子啊。
    然后我们从维护性的角度分析下。假如某个具体产品类需要进行一定的修改，很可能需要修改对应的工厂类。当同时需要修改多个产品类的时候，对工厂类的修改会变得相当麻烦（对号入座已经是个问题了）。反而简单工厂没有这些麻烦，当多个产品类需要修改是，简单工厂模式仍然仅仅需要修改唯一的工厂类（无论怎样都能改到满足要求吧？大不了把这个类重写）。
- **根据设计理论建议：工厂方法模式。但实际上，我们一般都用简单工厂模式。**

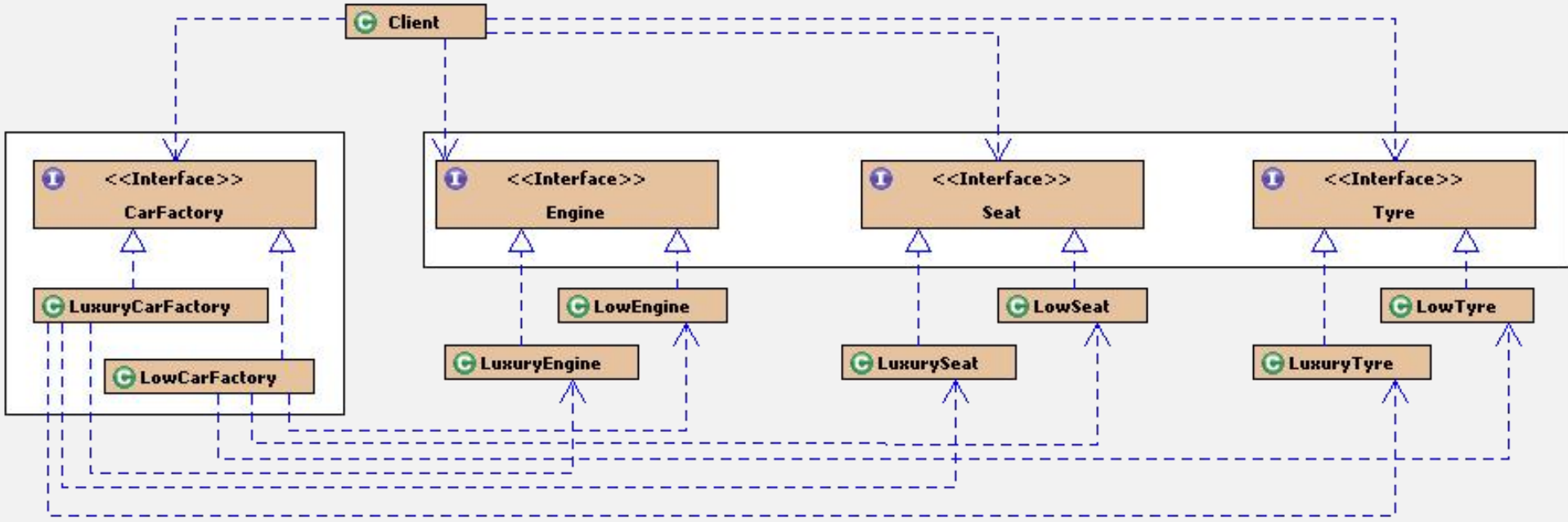
抽象工厂模式

- 抽象工厂模式
  - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）
  - 抽象工厂模式是工厂方法模式的升级版本，在有多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方式。



抽象工厂模式

- 类图

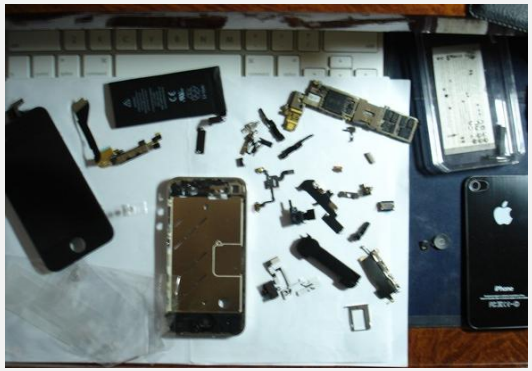


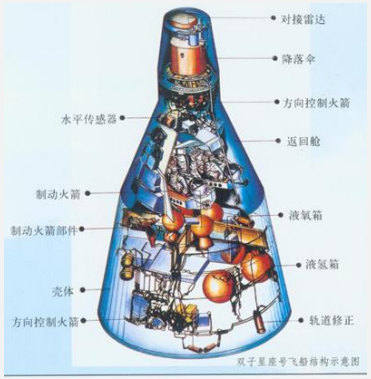
工厂模式

- 工厂模式要点：
  - 简单工厂模式(静态工厂模式)
    - 虽然某种程度不符合设计原则，但实际使用最多。
  - 工厂方法模式
    - 不修改已有类的前提下，通过增加新的工厂类实现扩展。
  - 抽象工厂模式
    - 不可以增加产品，可以增加产品族！
- 应用场景
  - JDK中Calendar的getInstance方法
  - JDBC中Connection对象的获取
  - Hibernate中SessionFactory创建Session
  - spring中IOC容器创建管理bean对象
  - XML解析时的DocumentBuilderFactory创建解析器对象
  - 反射中Class对象的newInstance()

建造者模式

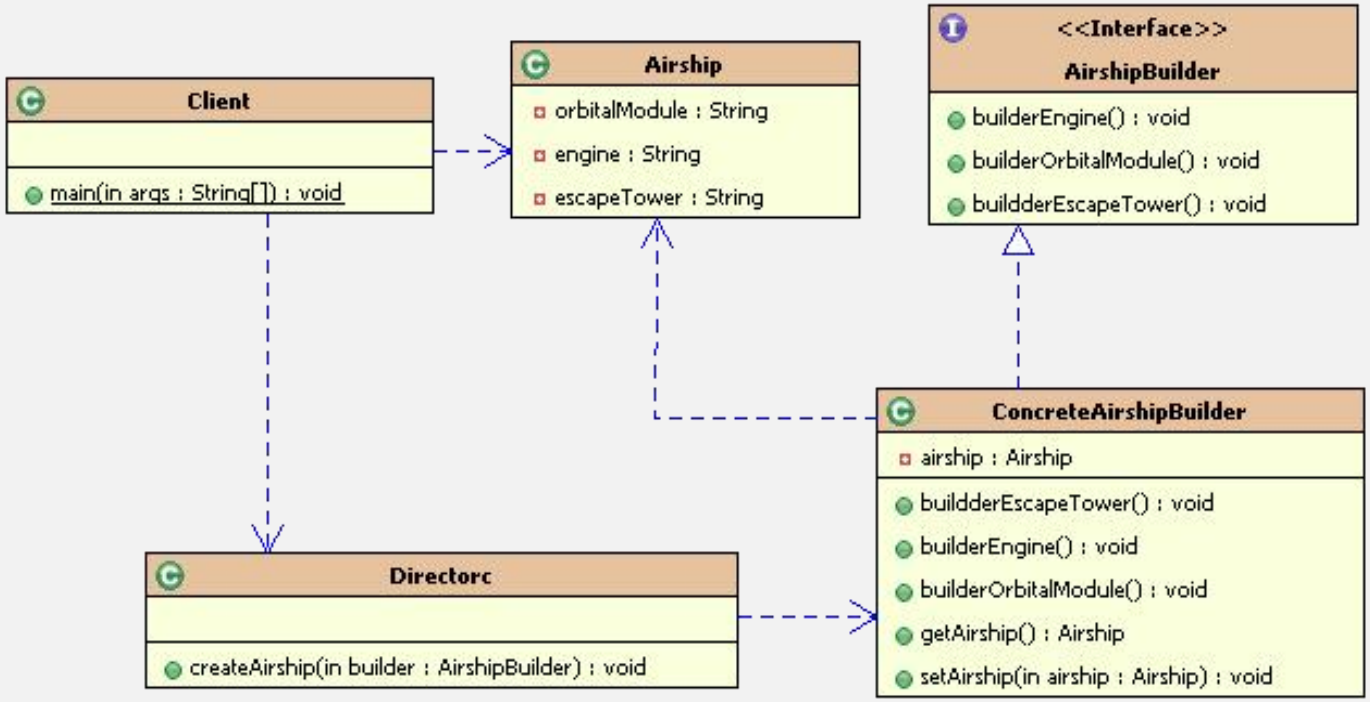
- 场景：
  - 我们要建造一个复杂的产品。比如：神州飞船,Iphone。这个复杂的产品的创建。有这样一个问题需要处理：
    - 装配这些子组件是不是有个步骤问题?
  - 实际开发中，我们所需要的对象构建时，也非常复杂，有很多步骤需要处理时。
- **建造模式的本质：**
  - 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。 从而可以构造出复杂的对象。这个模式适用于：某个对象的构建过程复杂的情况下使用。
  - 由于实现了构建和装配的解耦。不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用。





建造者模式

- 构建“尚学堂牌”神舟飞船的示例



建造者模式

- 开发中应用场景：
  - StringBuilder类的append方法
  - SQL中的PreparedStatement
  - JDOM中，DomBuilder、SAXBuilder

原型模式prototype

- 场景：
  - 思考一下：克隆技术是怎么样的过程？克隆羊多利大家还记得吗？
  - javascript语言中的，继承怎么实现?那里面也有prototype，大家还记得吗?
- **原型模式：**
  - 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。
  - 就是java中的克隆技术，以某个对象为原型，复制出新的对象。显然，新的对象具备原型对象的特点
  - 优势有：效率高(直接克隆，避免了重新执行构造过程步骤)、“接口造接口”。
  - 克隆类似于new，但是不同于new。new创建新的对象属性采用的是默认值。克隆出的对象的属性值完全和原型对象相同。并且克隆出的新对象改变不会影响原型对象。然后，再修改克隆对象的值。
- 原型模式实现：
  - Cloneable接口和clone方法
  - Prototype模式中实现起来最困难的地方就是内存复制操作，所幸在Java中提供了clone()方法替我们做了绝大部分事情。
- 注意用词：克隆和拷贝一回事！

原型模式prototype

- 浅克隆存在的问题
  - 被复制的对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。
- 深克隆如何实现?
  - 深克隆把引用的变量指向复制过的新对象，而不是原有的被引用的对象。
  - 深克隆：让已实现Clonable接口的类中的属性也实现Clonable接口
  - 基本数据类型能够自动实现深度克隆（值的复制）
  - 有时候增加克隆的代码比较麻烦！

  - 可以利用序列化和反序列化实现深克隆！

原型模式prototype

- 短时间大量创建对象时，原型模式和普通new方式效率测试

- 开发中的应用场景
  - 原型模式很少单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。
    - spring中bean的创建实际就是两种：单例模式和原型模式。（当然，原型模式需要和工厂模式搭配起来）

创建型模式的总结

- **创建型模式：都是用来帮助我们创建对象的！**
  - 单例模式
    - 保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。
  - 工厂模式
    - 简单工厂模式
      - 用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
    - 工厂方法模式
      - 用来生产同一等级结构中的固定产品。（支持增加任意产品）
    - 抽象工厂模式
      - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）
  - 建造者模式
    - 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。 从而可以构造出复杂的对象。
  - 原型模式
    - 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式

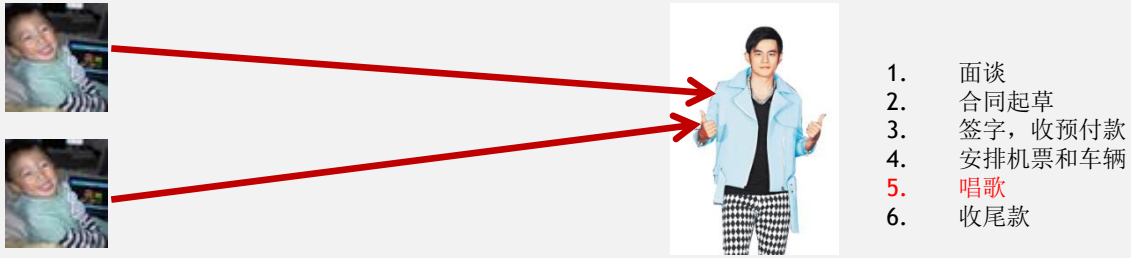
结构型模式

- **结构型模式**：
  - 核心作用：是从程序的结构上实现松耦合，从而可以扩大整体的类结构，用来解决更大的问题。
  - 分类：
    - 代理模式、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

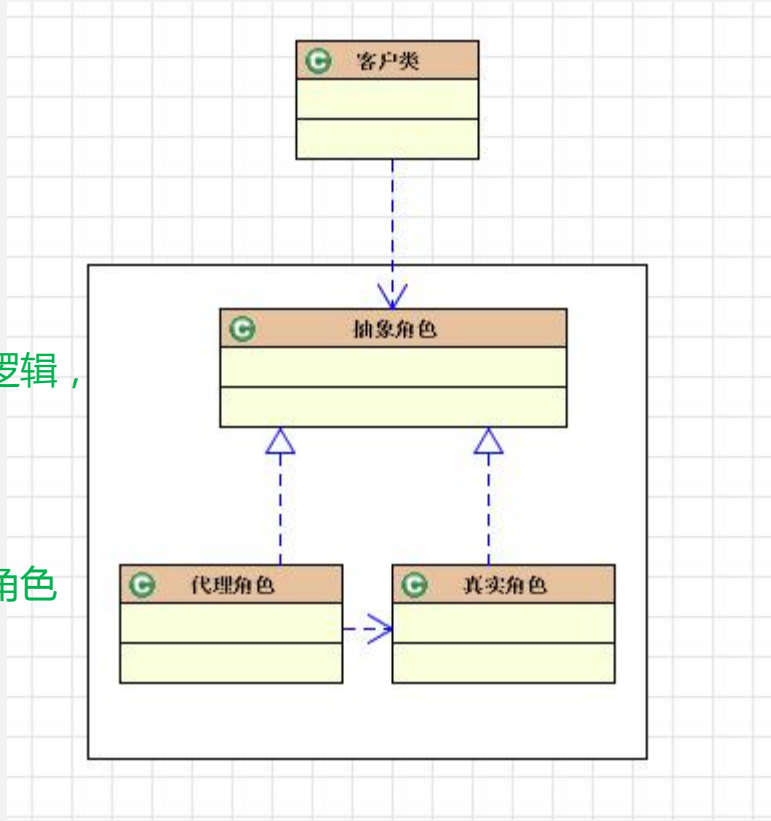
- **代理模式(Proxy pattern)：**
  - 核心作用：
    - 通过代理，控制对对象的访问！
      可以详细控制访问**某个（某类）对象**的方法，在调用这个方法前做前置处理，调用这个方法后做后置处理。（即：AOP的微观实现！）
  - AOP(Aspect Oriented Programming面向切面编程)的核心实现机制！

从而实现将统一流程代码放到代理类中处理



代理模式

- **代理模式(Proxy pattern)：**
  - 核心角色：
    - 抽象角色
      - 定义代理角色和真实角色的公共对外方法
    - 真实角色
      - 实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。
      - **关注真正的业务逻辑！**
    - 代理角色
      - 实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。
      - **将统一的流程控制放到代理角色中处理！**

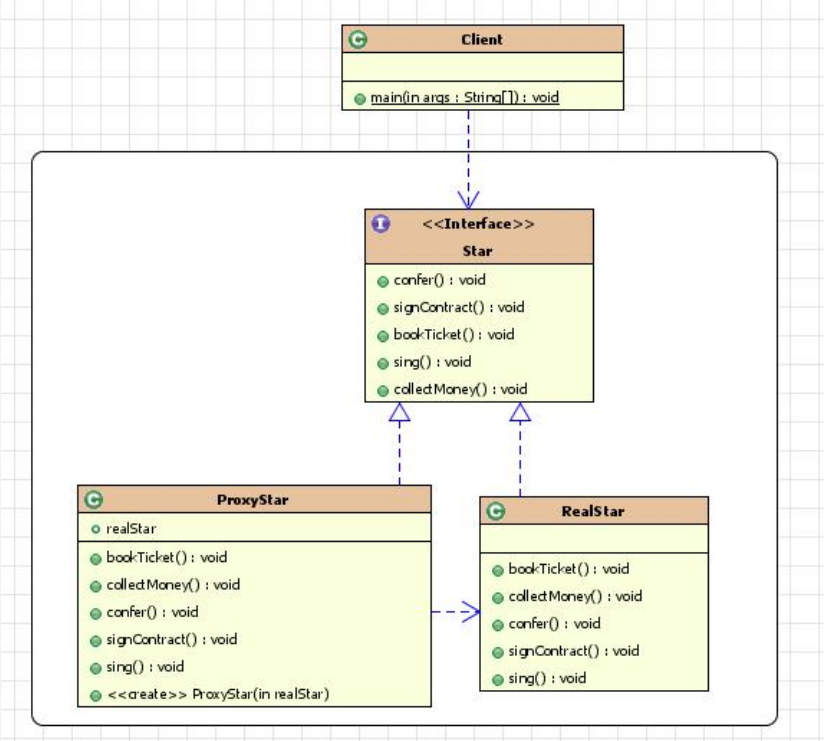


代理模式

- 应用场景：
  - 安全代理：屏蔽对真实角色的直接访问。
  - 远程代理：通过代理类处理远程方法调用(RMI)
  - 延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象。
    - 比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有100MB，在打开文件时不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开。
- 分类：
  - 静态代理(静态定义代理类)
  - 动态代理(动态生成代理类)
    - JDK自带的动态代理
    - javaassist字节码操作库实现(推荐)
    - CGLIB
    - ASM(底层使用指令，可维护性较差)

静态代理（static proxy）

- 静态代理(静态定义代理类)



动态代理(dynamic proxy)

- 动态代理(动态生成代理类)
  - JDK自带的动态代理
  - javaassist字节码操作库实现(推荐)
  - CGLIB
  - ASM(底层使用指令，可维护性较差)

代理模式

- 开发框架中应用场景：
  - struts2中拦截器的实现
  - 数据库连接池关闭处理
  - Hibernate中延时加载的实现
  - mybatis中实现拦截器插件
  - AspectJ的实现
  - spring中AOP的实现
  - web service
  - RMI远程方法调用
  - ...

  - 实际上，随便选择一个技术框架都会用到代理模式！！