**设计模式六大原则例子-- 依赖倒置原则（DIP）例子**

之前我们对设计模式的六大原则做了简单归纳，这篇博客是对依赖倒置原则进行的举例说明。

**依赖倒置原则的意义**

DIP是6大原则中最难以实现的原则，它是实现开闭原则的重要途径，DIP没有实现，就别想实现对扩展开放，对修改关闭。在项目中只要记住“面向接口编程”就基本上抓住了DIP的核心

**对各种概念进行一个描述：**

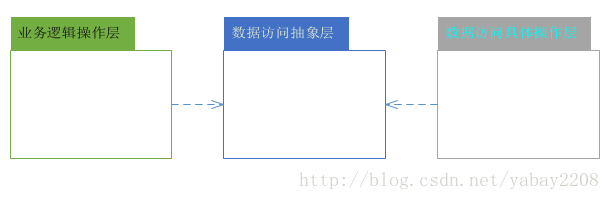
* **低层模块：**不可分割的原子逻辑，可能会根据业务逻辑经常变化。
* **高层模块：**低层模块的再组合，对低层模块的抽象。
* **抽象：** 接口或抽象类（是底层模块的抽象，特点：不能直接被实例化）
* **与接口或抽象类对应的实现类：**低层模块的具体实现（特点：可以直拉被实例化）

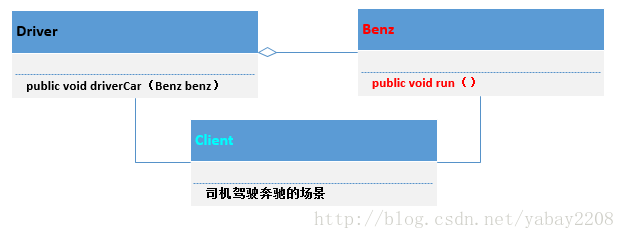
**我们先要写出低耦合高内聚的代码，在java中需要遵循如下原则：**   
**1. 模块间的依赖通过抽象类或接口发生，实现类之间的依赖关系也是通过抽象类或接口产生（实现类之间不应发生直接的依赖关系），降低系统的耦合性**   
**2. 接口或抽象不依赖于实现类，但实现类依赖接口或抽象类，实现类对系统需要的功能具体实现，提高类的内聚程度**

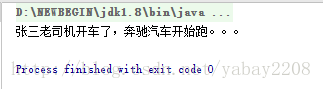
**依赖倒置原则的优点**

* **可以通过抽象使各个类或模块的实现彼此独立，不互相影响，实现模块间的松耦合（也是本质）**
* **可以规避一些非技术因素引起的问题（项目大时，需求变化的概率也越大，通过采用依赖倒置原则设计的接口或抽象类对实现类进行约束，可以减少需求变化引起的工作量剧增情况。同时，发生人员变动，只要文档完善，也可让维护人员轻松地扩展和维护）**
* **可以促进并行开发（如，两个类之间有依赖关系，只要制定出两者之间的接口（或抽象类）就可以独立开发了，规范已经定好了，而且项目之间的单元测试也可以独立地运行，而TDD开发模式更是DIP的最高级应用（特别适合项目人员整体水平较低时使用））**

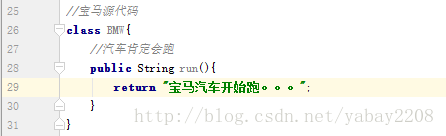
**依赖倒置原则的例子**

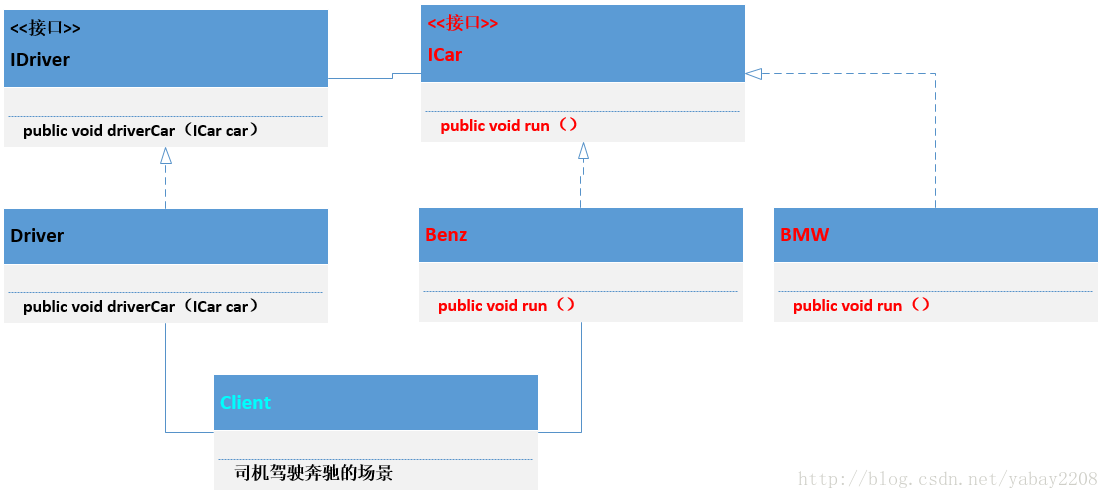
**1.数据访问抽象层：**   
“高层模块不应该依赖于低层模块，二者都应该依赖于抽象。”这一原则在分层架构模式中，得到了淋漓尽致地运用。   
例如，业务逻辑层（高层模块）的对象就不应该直接依赖于数据访问层（低层模块）的具体实现对象（**具体说的是：我们在编写业务代码的时候，不应该直接在某个业务代码处直接用jdbc操作数据库，而是业务层方法的参数应该是数据访问层的抽象**），**而应该通过数据访问层的抽象接口进行访问，如下图所示。如果高层模块直接依赖于低层模块，一旦低层模块发生变化，就会影响到高层模块。通过引入抽象，对于高层模块而言，低层模块的实现是可替换的。这实际上也是”开放封闭原则”的体现。**   
这一原则同时还体现了软件设计对”间接”的追求。下图中的数据访问抽象层就是在设计中引入的一层间接性：   


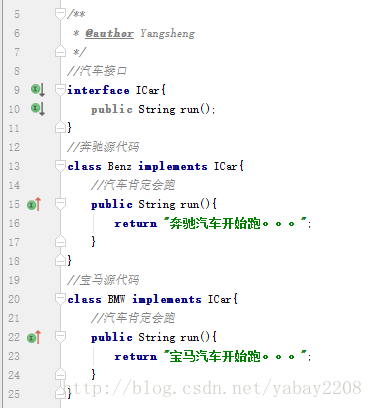
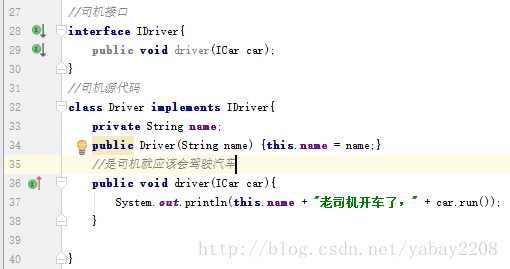
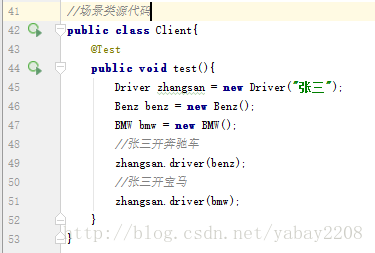
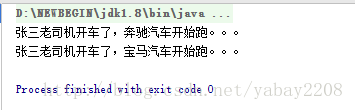
**2.汽车驾驶的一个例子**   
**我们第一次设计驾驶员驾驶奔驰汽车的时候，因为场景单一，很有可能就会陷入到面向实现编程（这里只是为了展现问题)，当然最小需求时，可以这样做：**   


**代码如下：**   
   
**场景运行结果如下：**   


**现在张三赚了一些钱，买了一辆宝马汽车：**



**宝马有了，却不能让张三开起来，这也太不合理了，我们的设计出了问题：司机类和奔驰类之间是紧耦合的关系，其导致的结果就是系统的可维护性大降低，可读性降低。//若需要改为开宝马，必须得更改司机类的public void drive(Benz benz)（低层模块，太不应该）现根据DIP原则，重新编写如下**   


**实现的代码：**   
**汽车：**   
   
**司机：**   
   
**场景：**   
   
**运行结果：**   


**依赖的三种写法，可选的具体实现细节：**

* **构造函数传递依赖对象**

public interface IDriver{

//是司机就应该会驾驶汽车

public void drive();

}

public class Driver implements IDriver{

private ICar car;

//构造函数注入

public Driver(ICar \_car){

this.car = \_car;

}

//司机的主要职责就是驾驶汽车

public void drive(){

this.car.run();

}

}

* **Setter方法传递依赖对象**

public interface IDriver {

//车辆型号

public void setCar(ICar car);

//是司机就应该会驾驶汽车

public void drive();

}

public class Driver implements IDriver{

private ICar car;

public void setCar(ICar car){

this.car = car;

}

//司机的主要职责就是驾驶汽车

public void drive(){

this.car.run();

}

}

**其实就是用接口声明依赖对象而不要用具体实现类来声明，维护后期扩展性**

public interface IDriver{

//是司机就应该会驾驶汽车

public void drive(ICar \_car);

}

public class Driver implements IDriver{

public Driver(){

}

//司机的主要职责就是驾驶汽车

public void drive(ICar \_car){

this.car.run();

}

}