

课程目标

内容定位

1. 开闭原则(OCP)
2. 依赖倒置原则(DIP)
3. 单一职责原则(SRP)
4. 接口隔离原则(ISP)
5. 迪米特法则(LoD)
6. 里氏替换原则(LSP)
7. 合成复用原则(CARP)
8. 设计原则总结

## 课程目标

- 1、通过对节课内容的学习，了解设计原则的重要性。
- 2、掌握七大设计原则的具体内容。

## 内容定位

学习设计原则，学习设计模式的基础。在实际开发过程中，并不是一定要求所有代码都遵循设计原则，我们要考虑人力、时间、成本、质量，不是刻意追求完美，要在适当的场景遵循设计原则，体现的是一种平衡取舍，帮助我们设计出更加优雅的代码结构。

### 1. 开闭原则(OCP)

开闭原则（Open-Closed Principle, OCP）是指一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。所谓的开闭，也正是对扩展和修改两个行为的一个原则。强调的是用抽象构建框架，用实现扩展细节。可以提高软件系统的可复用性及可维护性。开闭原则，是面向对象设计中最基础的设计原则。它指导我们如何建立稳定灵活的系统，例如：我们版本更新，我尽可能不修改源代码，但是可以增加新功能。

在现实生活中对于开闭原则也有体现。比如，很多互联网公司都实行弹性制作息时间，规定每天工作 8 小时。意思就是说，对于每天工作 8 小时这个规定是关闭的，但是你什么时候来，什么时候走是开放的。早来早走，晚来晚走。

实现开闭原则的核心思想就是面向抽象编程，接下来我们来看一段代码：

以咕泡学院的课程体系为例，首先创建一个课程接口 ICourse：

```
1 package cn.sitedev.ocp.course;
2
3 /**
4  * 课程接口
5  */
6 public interface ICourse {
7     Integer getId();
8
9     String getName();
10
11     Double getPrice();
12
13 }
```

整个课程生态有 Java 架构、大数据、人工智能、前端、软件测试等，我们来创建一个 Java 架构课程的类 JavaCourse：

```
1 package cn.sitedev.ocp.course;
2
3 public class JavaCourse implements ICourse {
4     private Integer id;
5     private String name;
6     private Double price;
7
8     public JavaCourse(Integer id, String name, Double price) {
9         this.id = id;
10        this.name = name;
11        this.price = price;
12    }
13
14    @Override
15    public Integer getId() {
16        return id;
17    }
18
19    @Override
20    public String getName() {
21        return name;
22    }
23
24    @Override
25    public Double getPrice() {
26        return price;
27    }
28 }
```

```

17     }
18
19     @Override
20     public String getName() {
21         return name;
22     }
23
24     @Override
25     public Double getPrice() {
26         return price;
27     }
28 }

```

现在我们要给 Java 架构课程做活动，价格优惠。如果修改 JavaCourse 中的 getPrice() 方法，则会存在一定的风险，可能影响其他地方的调用结果。我们如何在不修改原有代码前提下，实现价格优惠这个功能呢？现在，我们再写一个处理优惠逻辑的类，JavaDiscountCourse 类（思考一下为什么要叫 JavaDiscountCourse，而不叫 DiscountCourse）：

```

1 package cn.sitedev.ocp.course;
2
3 public class JavaDiscountCourse extends JavaCourse {
4     public JavaDiscountCourse(Integer id, String name, Double price) {
5         super(id, name, price);
6     }
7
8     public Double getOriginPrice() {
9         return super.getPrice();
10    }
11
12    public Double getPrice() {
13        return super.getPrice() * 0.61;
14    }
15 }
16 //////////////////////////////////////
17 package cn.sitedev.ocp.course;
18
19 public class OpenCloseTest {

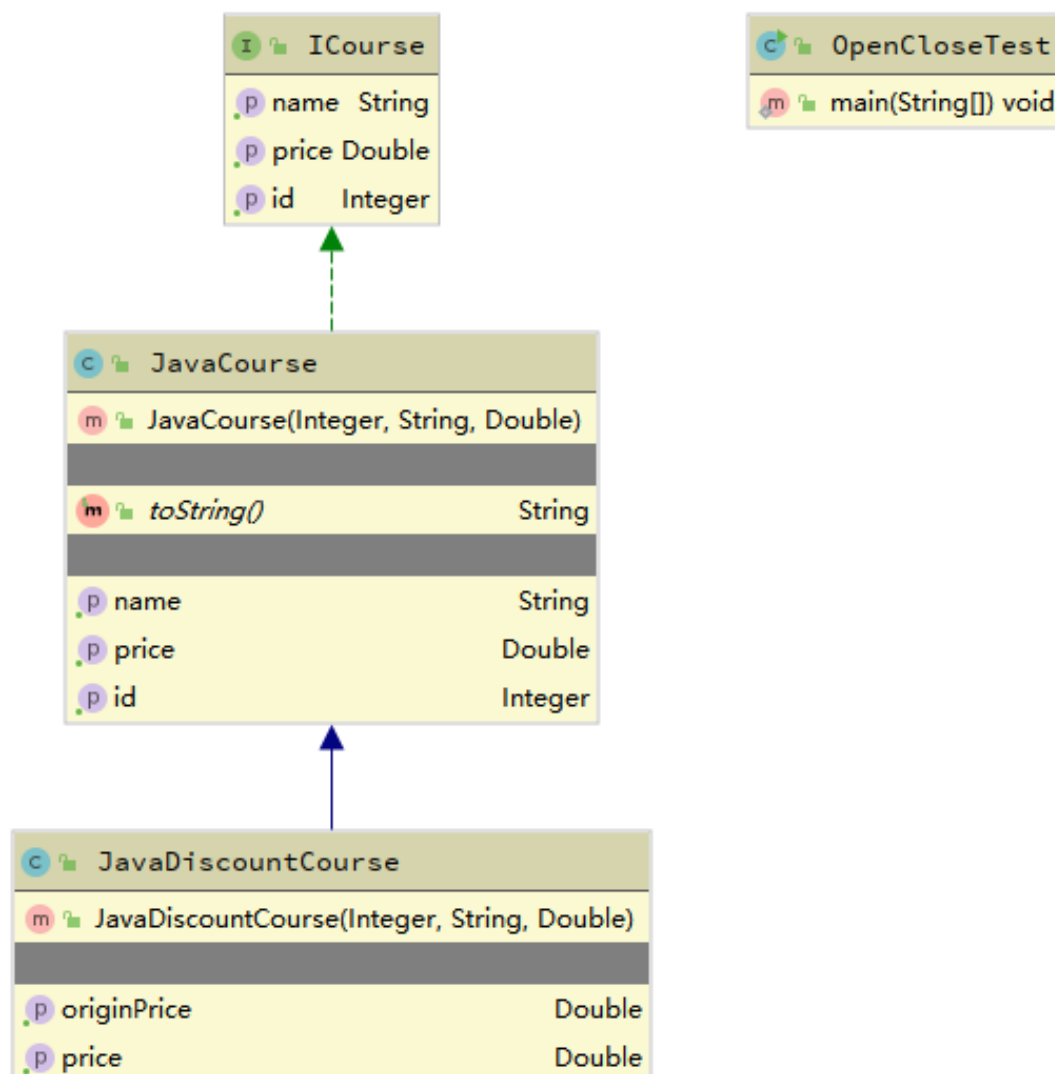
```

```

20     public static void main(String[] args) {
21         JavaCourse javaCourse = new JavaCourse(1, "Java架构师课程", 8200D);
22         System.out.println("打折前:" + javaCourse);
23         JavaDiscountCourse javaDiscountCourse = new JavaDiscountCourse(jav
24             javaCourse.getPrice());
25         System.out.println("打折后:" + javaDiscountCourse);
26     }
27 }

```

回顾一下，简单一下类结构图：



## 2. 依赖倒置原则(DIP)

依赖倒置原则 ( Dependence Inversion Principle,DIP ) 是指设计代码结构时，高层模

块不应该依赖底层模块，二者都应该依赖其抽象。抽象不应该依赖细节；细节应该依赖抽象。通过依赖倒置，可以减少类与类之间的耦合性，提高系统的稳定性，提高代码的可读性和可维护性，并能够降低修改程序所造成的风险。接下来看一个案例，还是以课程为例，先来创建一个类 Tom：

```
1 package cn.sitedev.dip.course;
2
3 public class Tom {
4     public void studyJavaCourse() {
5         System.out.println("Tom 在学习Java课程");
6     }
7
8     public void studyPythonCourse() {
9         System.out.println("Tom 在学习Python课程");
10    }
11 }
```

来调用一下：

```
1 package cn.sitedev.dip.course;
2
3 public class DependenceInversionTest {
4     public static void main(String[] args) {
5         Tom tom = new Tom();
6         tom.studyJavaCourse();
7         tom.studyPythonCourse();
8     }
9 }
```

Tom 热爱学习，目前正在学习 Java 课程和 Python 课程。大家都知道，学习也是会上瘾的。随着学习兴趣的暴涨，现在 Tom 还想学习 AI 人工智能的课程。这个时候，业务扩展，我们的代码要从底层到高层（调用层）一次修改代码。在 Tom 类中增加 studyAICourse()的方法，在高层也要追加调用。

如此一来，系统发布以后，实际上是非常不稳定的，在修改代码的同时也会带来意想

不到的风险。接下来我们优化代码，创建一个课程的抽象 ICourse 接口：

```
1 package cn.sitedev.dip.course;
2
3 public interface ICourse {
4     void study();
5 }
```

然后写 JavaCourse 类：

```
1 package cn.sitedev.dip.course;
2
3 public class JavaCourse implements ICourse {
4     @Override
5     public void study() {
6         System.out.println("Tom在学习Java课程");
7     }
8 }
```

再实现 PythonCourse 类：

```
1 package cn.sitedev.dip.course;
2
3 public class PythonCourse implements ICourse {
4     @Override
5     public void study() {
6         System.out.println("Tom在学习Python课程");
7     }
8 }
```

修改 Tom 类：

```

1 package cn.sitedev.dip.course.improved.interfaceinjection;
2
3 import cn.sitedev.dip.course.improved.ICourse;
4
5 public class Tom {
6     public void study(ICourse course) {
7         course.study();
8     }
9 }

```

来看调用：

```

1 package cn.sitedev.dip.course.improved.interfaceinjection;
2
3 import cn.sitedev.dip.course.improved.JavaCourse;
4 import cn.sitedev.dip.course.improved.PythonCourse;
5
6 public class DependenceInversionTest {
7     public static void main(String[] args) {
8         Tom tom = new Tom();
9         tom.study(new JavaCourse());
10        tom.study(new PythonCourse());
11    }
12 }

```

我们这时候再看来代码，Tom 的兴趣无论怎么暴涨，对于新的课程，我只需要新建一个类，通过传参的方式告诉 Tom，而不需要修改底层代码。实际上这是一种大家非常熟悉的方式，叫依赖注入。注入的方式还有构造器方式和 setter 方式。我们来看构造器注入方式：

```

1 package cn.sitedev.dip.course.improved.constructorinjection;
2
3 import cn.sitedev.dip.course.improved.ICourse;
4
5 public class Tom {

```

```
6     private ICourse course;
7
8     public Tom(ICourse course) {
9         this.course = course;
10    }
11
12    public void study() {
13        course.study();
14    }
15 }
```

看调用代码：

```
1 package cn.sitedev.dip.course.improved.constructorinjection;
2
3 import cn.sitedev.dip.course.improved.JavaCourse;
4 import cn.sitedev.dip.course.improved.PythonCourse;
5
6 public class DependenceInversionTest {
7     public static void main(String[] args) {
8         Tom tom = new Tom(new JavaCourse());
9         tom.study();
10        tom = new Tom(new PythonCourse());
11        tom.study();
12    }
13 }
```

根据构造器方式注入，在调用时，每次都要创建实例。那么，如果 Tom 是全局单例，则我们就只能选择用 Setter 方式来注入，继续修改 Tom 类的代码：

```
1 package cn.sitedev.dip.course.improved.setterinjection;
2
3 import cn.sitedev.dip.course.improved.ICourse;
4
5 public class Tom {
6     private ICourse course;
```

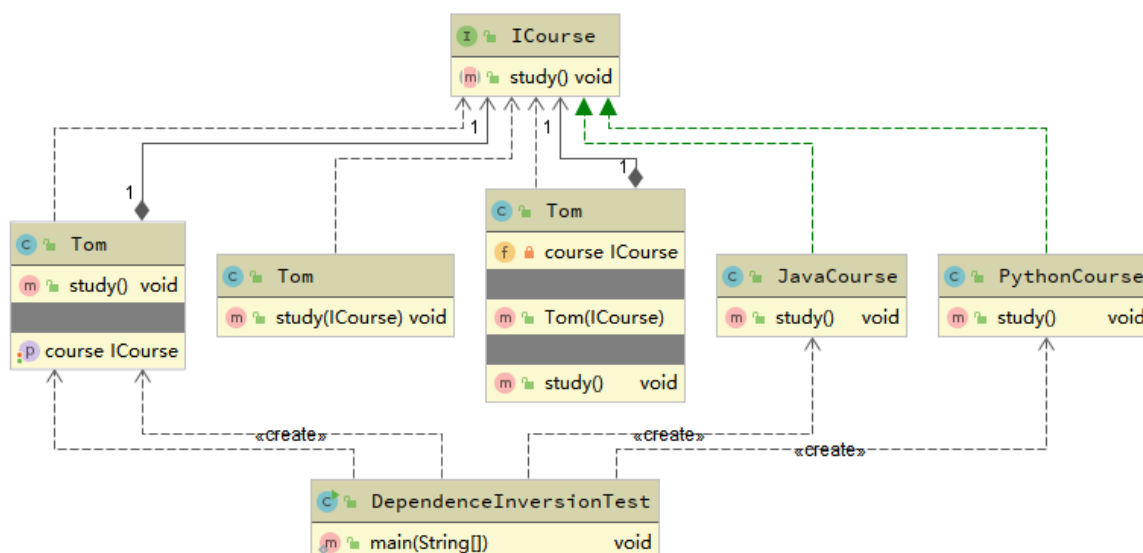


```
7
8     public ICourse getCourse() {
9         return course;
10    }
11
12    public void setCourse(ICourse course) {
13        this.course = course;
14    }
15
16    public void study() {
17        course.study();
18    }
19 }
```

看调用代码：

```
1 package cn.sitedev.dip.course.improved.setterinjection;
2
3
4 import cn.sitedev.dip.course.improved.JavaCourse;
5 import cn.sitedev.dip.course.improved.PythonCourse;
6
7 public class DependenceInversionTest {
8     public static void main(String[] args) {
9         Tom tom = new Tom();
10        tom.setCourse(new JavaCourse());
11        tom.study();
12
13
14        tom.setCourse(new PythonCourse());
15        tom.study();
16    }
17 }
```

现在我们再来看最终的类图：



大家要切记：以抽象为基准比以细节为基准搭建起来的架构要稳定得多，因此大家在拿到需求之后，要面向接口编程，先顶层再细节来设计代码结构。

### 3. 单一职责原则(SRP)

单一职责 ( Simple Responsibility Pinciple , SRP ) 是指不要存在多于一个导致类变更的原因。假设我们有一个 Class 负责两个职责，一旦发生需求变更，修改其中一个职责的逻辑代码，有可能会导致另一个职责的功能发生故障。这样一来，这个 Class 存在两个导致类变更的原因。如何解决这个问题呢？

我们就要给两个职责分别用两个 Class 来实现，进行解耦。后期需求变更维护互不影响。这样的设计，可以降低类的复杂度，提高类的可读性，提高系统的可维护性，降低变更引起的风险。总体来说就是一个 Class/Interface/Method 只负责一项职责。接下来，我们来看代码实例，还是用课程举例，我们的课程有直播课和录播课。直播课不能快进和快退，录播可以可以任意的反复观看，功能职责不一样。还是先创建一个 Course 类：

```

1 package cn.sitedev.srp.course;
2
3 public class Course {
4     public void study(String courseName) {
5         if ("直播课".equals(courseName)) {
6             System.out.println("不能快进");
7         } else {

```

```

8         System.out.println("可以任意来回播放");
9     }
10 }
11 }

```

看代码调用：

```

1 package cn.sitedev.srp.course;
2
3 public class SmpleResponsibilityTest {
4     public static void main(String[] args) {
5         Course course = new Course();
6         course.study("直播课");
7         course.study("录播课");
8     }
9 }

```

从上面代码来看，Course 类承担了两种处理逻辑。假如，现在要对课程进行加密，那么直播课和录播课的加密逻辑都不一样，必须要修改代码。而修改代码逻辑势必会相互影响容易造成不可控的风险。我们对职责进行分离解耦，来看代码，分别创建两个类 ReplayCourse 和 LiveCourse：

LiveCourse类:

```

1 package cn.sitedev.srp.course.improved;
2
3 public class LiveCourse {
4     public void study(String courseName) {
5         System.out.println(courseName + "不能快进看");
6     }
7 }

```

ReplayCourse 类:

```

1 package cn.sitedev.srp.course.improved;
2
3 public class ReplayCourse {
4     public void study(String courseName) {
5         System.out.println(courseName + "可以任意来回播放");
6     }
7 }

```

调用代码：

```

1 package cn.sitedev.srp.course.improved;
2
3 public class SmpleResponsibilityTest {
4     public static void main(String[] args) {
5         LiveCourse liveCourse = new LiveCourse();
6         liveCourse.study("直播课");
7
8         ReplayCourse replayCourse = new ReplayCourse();
9         replayCourse.study("录播课");
10    }
11 }

```

业务继续发展，课程要做权限。没有付费的学员可以获取课程基本信息，已经付费的学员可以获得视频流，即学习权限。那么对于控制课程层面上至少有两个职责。我们可以把展示职责和管理职责分离开来，都实现同一个抽象依赖。设计一个顶层接口,创建 ICourse 接口：

```

1 package cn.sitedev.srp.course.improved;
2
3 public interface ICourse {
4     /**
5      * 获得基本信息
6      *
7      * @return
8      */
9 }

```

```

9      String getCourseName();
10
11      /**
12       * 获得视频流
13       *
14       * @return
15       */
16      byte[] getCourseVideo();
17
18      /**
19       * 学习课程
20       */
21      void studyCourse();
22
23      /**
24       * 退款
25       */
26      void refundCourse();
27 }

```

我们可以把这个接口拆成两个接口，创建一个接口 `ICourseInfo` 和 `ICourseManager`：

`ICourseInfo` 接口：

```

1 package cn.sitedev.srp.course.improved;
2
3 public interface ICourseInfo {
4     String getCourseName();
5
6     byte[] getCourseVideo();
7 }

```

`ICourseManager` 接口：

```

1 package cn.sitedev.srp.course.improved;
2

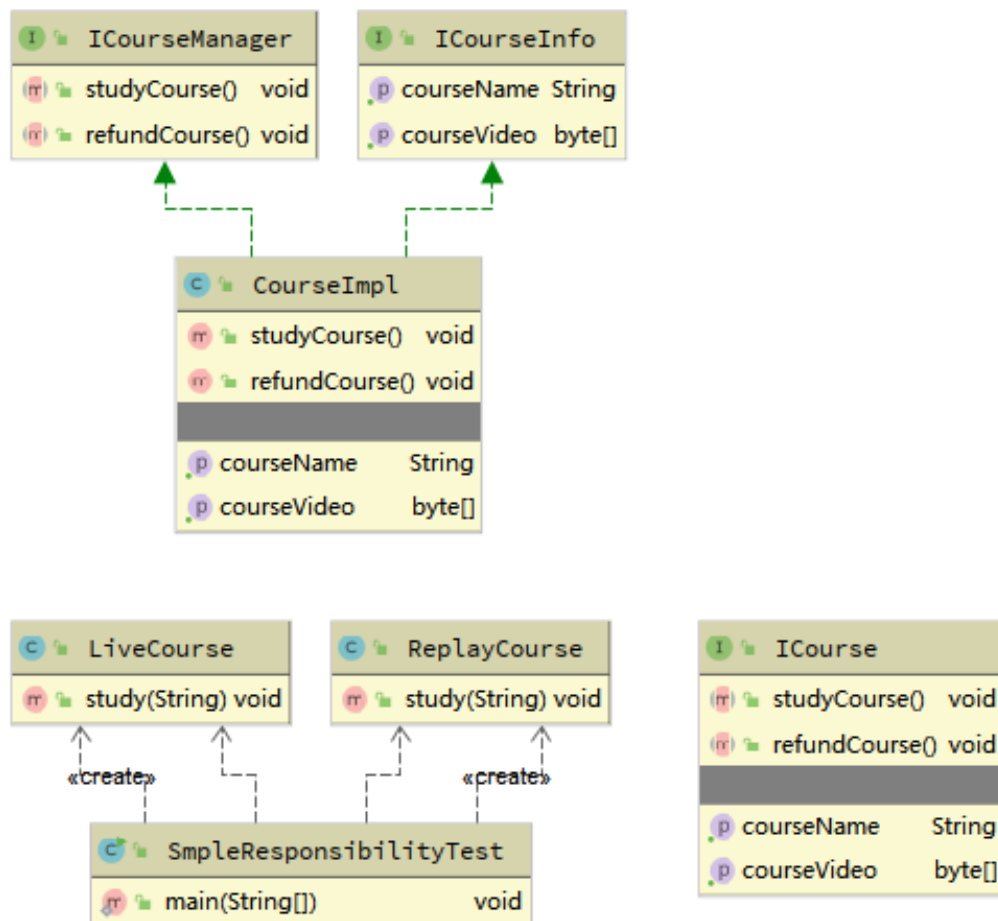
```

```

3 public interface ICourseManager {
4     void studyCourse();
5
6     void refundCourse();
7 }

```

来看一下类图：



下面我们来看一下方法层面的单一职责设计。有时候，我们为了偷懒，通常会把一个方法写成下面这样：

```

private void modifyUserInfo(String userName,String address){
    userName = "Tom";
    address = "Changsha";
}

```

还可能写成这样：

```
private void modifyUserInfo(String userName,String... fileds){
    userName = "Tom";
    //    address = "Changsha";
}
private void modifyUserInfo(String userName,String address,boolean bool){
    if(bool){

    }else{

    }

    userName = "Tom";
    address = "Changsha";
}
```

显然，上面的 modifyUserInfo()方法中都承担了多个职责，既可以修改 userName,也可以修改address，甚至更多，明显不符合单一职责。那么我们做如下修改，把这个方法拆成两个：

```
private void modifyUserName(String userName){
    userName = "Tom";
}
private void modifyAddress(String address){
    address = "Changsha";
}
```

这修改之后，开发起来简单，维护起来也容易。但是，我们在实际开发中会项目依赖，组合，聚合这些关系，还有还有项目的规模，周期，技术人员的水平，对进度的把控，很多类都不符合单一职责。但是，我们在编写代码的过程，尽可能地让接口和方法保持单一职责，对我们项目后期的维护是有很大帮助的。

## 4. 接口隔离原则(ISP)

接口隔离原则（Interface Segregation Principle, ISP）是指用多个专门的接口，而不使用单一的总接口，客户端不应该依赖它不需要的接口。这个原则指导我们在设计接口时应当注意以下几点：

- 1、一个类对一类的依赖应该建立在最小的接口之上。
- 2、建立单一接口，不要建立庞大臃肿的接口。
- 3、尽量细化接口，接口中的方法尽量少（不是越少越好，一定要适度）。

接口隔离原则符合我们常说的高内聚低耦合的设计思想，从而使得类具有很好的可读性、可扩展性和可维护性。我们在设计接口的时候，要多花时间去思考，要考虑业务模型，包括以后有可能发生变更的地方还要做一些预判。所以，对于抽象，对业务模型的理解是非常重要的。下面我们来看一段代码，写一个动物行为的抽象：

IAnimal 接口：

```
1 package cn.sitedev.isp;
2
3 public interface IAnimal {
4     void eat();
5
6     void fly();
7
8     void swim();
9 }
```

Bird 类实现：

```
1 package cn.sitedev.isp;
2
3 public class Bird implements IAnimal {
4     @Override
5     public void eat() {
6
7     }
8
9     @Override
10    public void fly() {
11
12    }
13
14    @Override
15    public void swim() {
16
17    }
18 }
```

Dog 类实现：

```
1 package cn.sitedev.isp;
2
```



```

3 public class Dog implements IAnimal {
4     @Override
5     public void eat() {
6
7     }
8
9     @Override
10    public void fly() {
11
12    }
13
14    @Override
15    public void swim() {
16
17    }
18 }

```

可以看出，Bird 的 swim()方法可能只能空着，Dog 的 fly()方法显然不可能的。这时候，我们针对不同动物行为来设计不同的接口，分别设计 IEatAnimal，IFlyAnimal 和 ISwimAnimal 接口，来看代码：

IEatAnimal 接口：

```

1 package cn.sitedev.isp.improved;
2
3 public interface IEatAnimal {
4     void eat();
5 }

```

IFlyAnimal 接口：

```

1 package cn.sitedev.isp.improved;
2
3 public interface IFlyAnimal {
4     void fly();
5 }

```

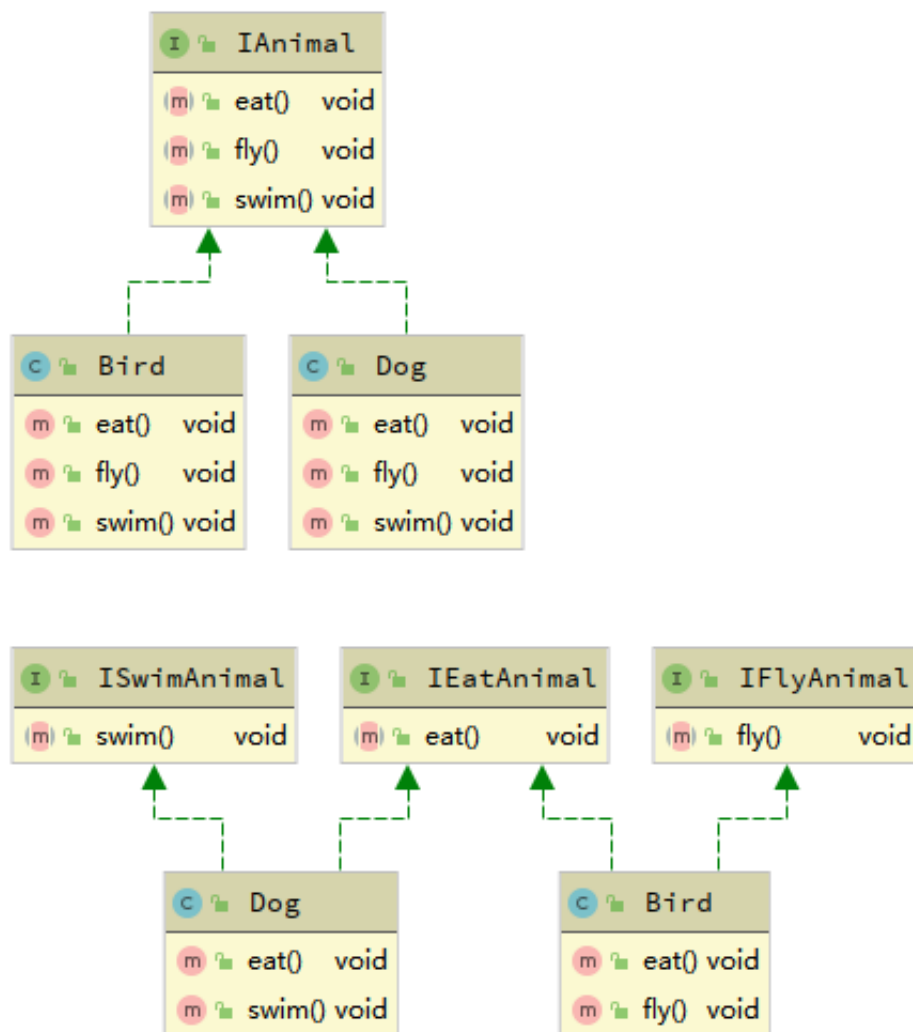
ISwimAnimal 接口：

```
1 package cn.sitedev.isp.improved;
2
3 public interface ISwimAnimal {
4     void swim();
5 }
```

Dog 只实现 IEatAnimal 和 ISwimAnimal 接口：

```
1 package cn.sitedev.isp.improved;
2
3 public class Dog implements IEatAnimal, ISwimAnimal {
4     @Override
5     public void eat() {
6
7     }
8
9     @Override
10    public void swim() {
11
12    }
13 }
```

来看下两种类图的对比，还是非常清晰明了的：



## 5. 迪米特法则(LoD)

迪米特原则 ( Law of Demeter LoD ) 是指一个对象应该对其他对象保持最少的了解，又叫最少知道原则 ( Least Knowledge Principle,LKP )，尽量降低类与类之间的耦合。迪米特原则主要强调只和朋友交流，不和陌生人说话。出现在成员变量、方法的输入、输出参数中的类都可以称之为成员朋友类，而出现在方法体内部的类不属于朋友类。

现在来设计一个权限系统，TeamLeader需要查看目前发布到线上的课程数量。这时候，TeamLeader要找到员工 Employee 去进行统计，Employee 再把统计结果告诉 TeamLeader。接下来我们还是来看代码：

Course 类：

```
1 package cn.sitedev.lod;
```

```
2
3 public class Course {
4 }
```

Employee 类：

```
1 package cn.sitedev.lod;
2
3 import java.util.List;
4
5 public class Employee {
6     public void checkNumberOfCourses(List<Course> courseList) {
7         System.out.println("目前已发布的课程数量是：" + courseList.size());
8     }
9 }
```

TeamLeader 类：

```
1 package cn.sitedev.lod;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class TeamLeader {
7     public void commandCheckNumber(Employee employee) {
8         List<Course> courseList = new ArrayList<>();
9         for (int i = 0; i < 20; i++) {
10             courseList.add(new Course());
11         }
12         employee.checkNumberOfCourses(courseList);
13     }
14 }
```

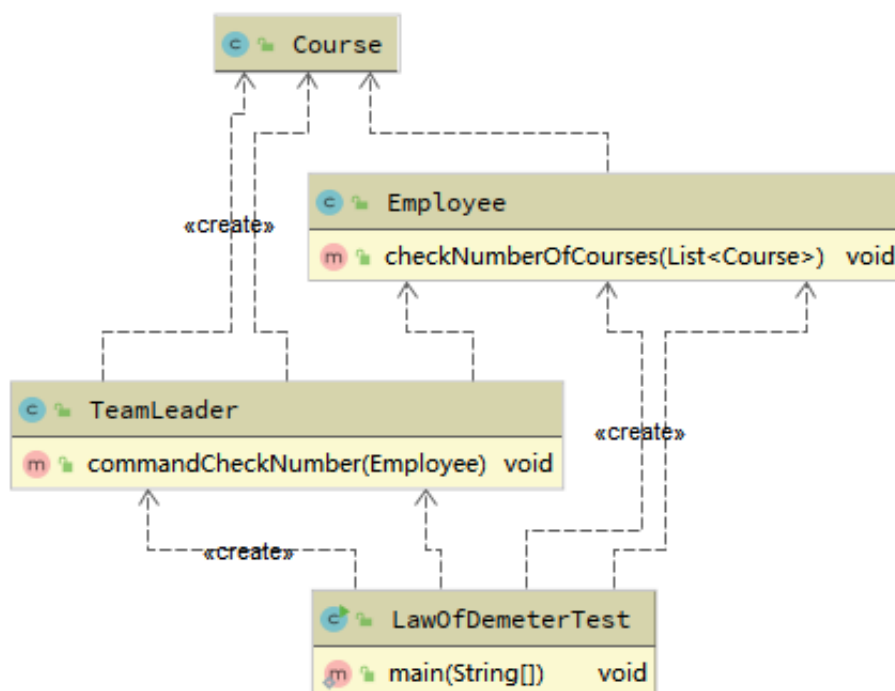
测试代码：

```

1 package cn.sitedev.lod;
2
3 public class LawOfDemeterTest {
4     public static void main(String[] args) {
5         TeamLeader teamLeader = new TeamLeader();
6         Employee employee = new Employee();
7         teamLeader.commandCheckNumber(employee);
8     }
9 }

```

写到这里，其实功能已经都已经实现，代码看上去也没什么问题。根据迪米特原则，TeamLeader只想要结果，不需要跟 Course 产生直接的交流。而 Employee 统计需要引用 Course 对象。TeamLeader和 Course 并不是朋友，从下面的类图就可以看出来：



下面来对代码进行改造：

```

1 package cn.sitedev.lod.improved;
2
3 import cn.sitedev.lod.Course;
4

```

```

5 import java.util.ArrayList;
6 import java.util.List;
7
8 public class Employee {
9     public void checkNumberOfCourses() {
10         List<Course> courseList = new ArrayList<>();
11         for (int i = 0; i < 20; i++) {
12             courseList.add(new Course());
13         }
14         System.out.println("目前已发布的课程数量是:" + courseList.size());
15     }
16 }

```

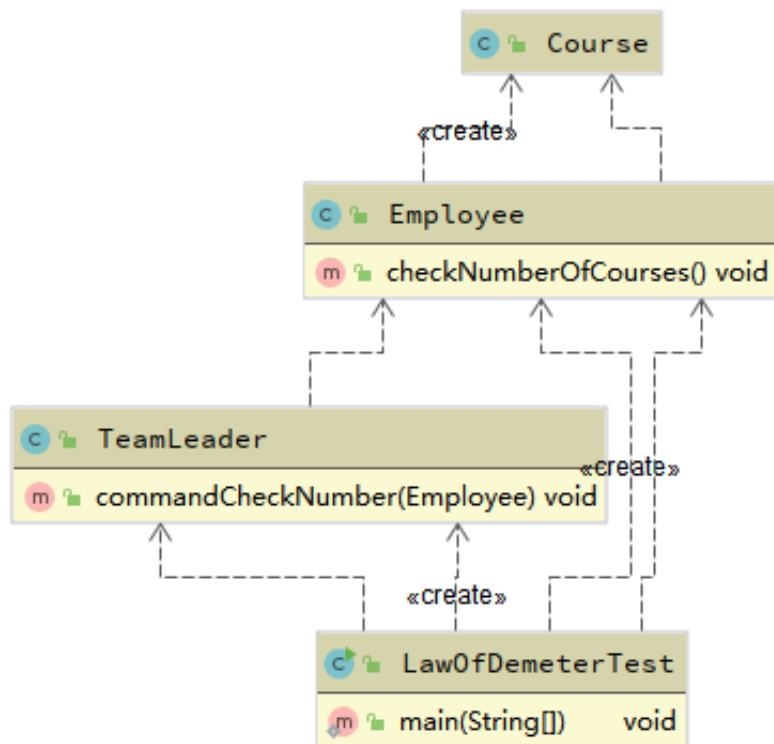
TeamLeader 类：

```

1 package cn.sitedev.lod.improved;
2
3 public class TeamLeader {
4     public void commandCheckNumber(Employee employee) {
5         employee.checkNumberOfCourses();
6     }
7 }

```

再来看下面的类图，Course 和 TeamLeader 已经没有关联了。



## 6. 里氏替换原则(LSP)

里氏替换原则 ( Liskov Substitution Principle,LSP ) 是指如果对每一个类型为 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 都替换成 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。

定义看上去还是比较抽象, 我们重新理解一下, 可以理解为一个软件实体如果适用一个父类的话, 那一定是适用于其子类, 所有引用父类的地方必须能透明地使用其子类的对象, 子类对象能够替换父类对象, 而程序逻辑不变。根据这个理解, 我们总结一下:

引申含义: 子类可以扩展父类的功能, 但不能改变父类原有的功能。

- 1、子类可以实现父类的抽象方法, 但不能覆盖父类的非抽象方法。
- 2、子类中可以增加自己特有的方法。
- 3、当子类的方法重载父类的方法时, 方法的前置条件 ( 即方法的输入/入参 ) 要比父类方法的输入参数更宽松。
- 4、当子类的方法实现父类的方法时 ( 重写/重载或实现抽象方法 ), 方法的后置条件 ( 即方法的输出/返回值 ) 要比父类更严格或相等。

在前面讲开闭原则的时候埋下了一个伏笔, 我们记得在获取折后时重写覆盖了父类的 getPrice()方法, 增加了一个获取源码的方法 getOriginPrice(), 显然就违背了里氏替换原则。我们修改一下代码, 不应该覆盖 getPrice()方法, 增加 getDiscountPrice()方

法：

```
1 package cn.sitedev.lsp.course;
2
3 /**
4  * 课程接口
5  */
6 public interface ICourse {
7     Integer getId();
8
9     String getName();
10
11     Double getPrice();
12
13 }
14 //////////////////////////////////////
15 package cn.sitedev.lsp.course;
16
17 import lombok.ToString;
18
19 @ToString
20 public class JavaCourse implements ICourse {
21     private Integer id;
22     private String name;
23     private Double price;
24
25     public JavaCourse(Integer id, String name, Double price) {
26         this.id = id;
27         this.name = name;
28         this.price = price;
29     }
30
31     @Override
32     public Integer getId() {
33         return id;
34     }
35
36     @Override
37     public String getName() {
```



```

38         return name;
39     }
40
41     @Override
42     public Double getPrice() {
43         return price;
44     }
45 }
46 ///////////////////////////////////////////////////
47 package cn.sitedev.lsp.course;
48
49 public class JavaDiscountCourse extends JavaCourse {
50     public JavaDiscountCourse(Integer id, String name, Double price) {
51         super(id, name, price);
52     }
53
54
55     public Double getDiscountPrice() {
56         return super.getPrice() * 0.61;
57     }
58 }

```

使用里氏替换原则有以下优点：

- 1、约束继承泛滥，开闭原则的一种体现。
- 2、加强程序的健壮性，同时变更时也可以做到非常好的兼容性，提高程序的维护性、扩展性。降低需求变更时引入的风险。

现在来描述一个经典的业务场景，用正方形、矩形和四边形的关系说明里氏替换原则，我们都知道正方形是一个特殊的长方形，那么就可以创建一个长方形父类 Rectangle 类：

```

1 package cn.sitedev.lsp.graph;
2
3 import lombok.Data;
4
5 @Data
6 public class Rectangle {

```

```
7     private long height;
8     private long width;
9 }
```

创建正方形 Square 类继承长方形：

```
1 package cn.sitedev.lsp.graph;
2
3 import lombok.Data;
4
5 @Data
6 public class Square extends Rectangle {
7     private long length;
8
9     @Override
10    public long getHeight() {
11        return getLength();
12    }
13
14    @Override
15    public void setHeight(long height) {
16        setLength(height);
17    }
18
19    @Override
20    public long getWidth() {
21        return getLength();
22    }
23
24    @Override
25    public void setWidth(long width) {
26        setLength(width);
27    }
28 }
```

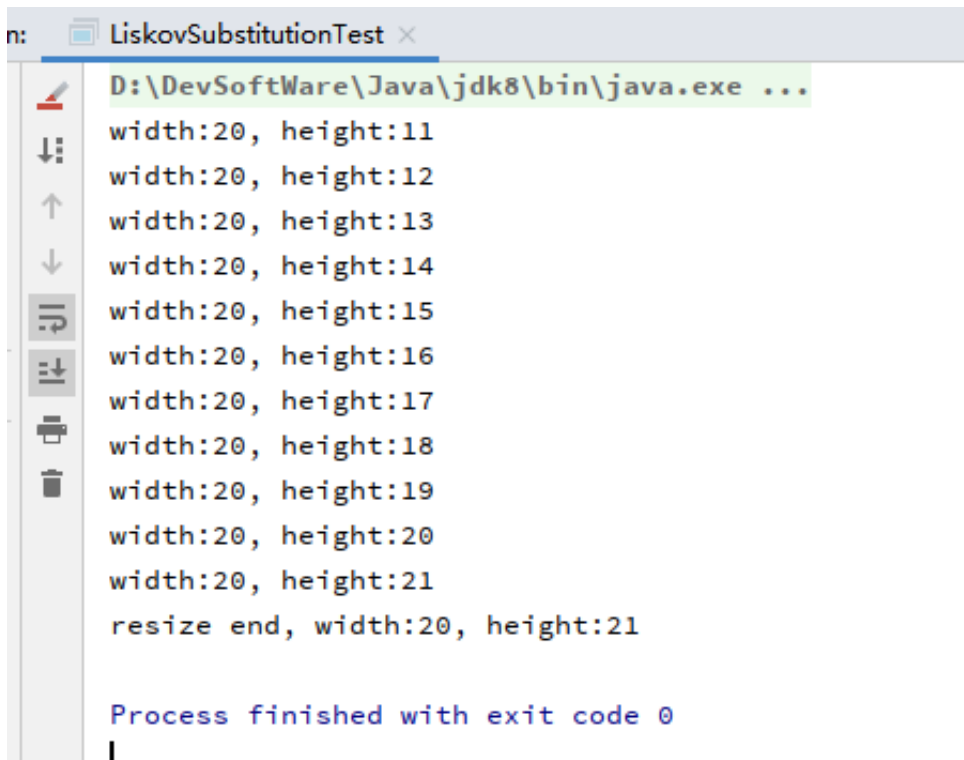
在测试类中创建 `resize()` 方法，根据逻辑长方形的宽应该大于等于高，我们让高一直自增，直到高等于宽变成正方形：

```

1 package cn.sitedev.lsp.graph;
2
3 public class LiskovSubstitutionTest {
4     public static void main(String[] args) {
5         Rectangle rectangle = new Rectangle();
6         rectangle.setWidth(20);
7         rectangle.setHeight(10);
8         resize(rectangle);
9     }
10
11     public static void resize(Rectangle rectangle) {
12         while (rectangle.getWidth() >= rectangle.getHeight()) {
13             rectangle.setHeight(rectangle.getHeight() + 1);
14             System.out.println("width:" + rectangle.getWidth() + ", height:" + rectangle.getHeight());
15         }
16         System.out.println("resize end, width:" + rectangle.getWidth() + ", height:" + rectangle.getHeight());
17     }
18 }

```

运行结果：



```

n: LiskovSubstitutionTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
width:20, height:11
width:20, height:12
width:20, height:13
width:20, height:14
width:20, height:15
width:20, height:16
width:20, height:17
width:20, height:18
width:20, height:19
width:20, height:20
width:20, height:21
resize end, width:20, height:21

Process finished with exit code 0
|

```

发现高比宽还大了，在长方形中是一种非常正常的情况。现在我们再来看下面的代

码，把长方形Rectangle 替换成它的子类正方形 Square，修改测试代码：

```
1 package cn.sitedev.lsp.graph;
2
3 public class LiskovSubstitutionTest {
4     public static void main(String[] args) {
5         Rectangle rectangle = new Rectangle();
6         rectangle.setWidth(20);
7         rectangle.setHeight(10);
8         resize(rectangle);
9         System.out.println("=====");
10        Square square = new Square();
11        square.setLength(10);
12        resize(square);
13    }
14
15    public static void resize(Rectangle rectangle) {
16        while (rectangle.getWidth() >= rectangle.getHeight()) {
17            rectangle.setHeight(rectangle.getHeight() + 1);
18            System.out.println("width:" + rectangle.getWidth() + ", height");
19        }
20        System.out.println("resize end, width:" + rectangle.getWidth() + "
21    }
22 }
```

这时候我们运行的时候就出现了死循环，违背了里氏替换原则，将父类替换为子类后，程序运行结果没有达到预期。因此，我们的代码设计是存在一定风险的。里氏替换原则只存在父类与子类之间，约束继承泛滥。我们再来创建一个基于长方形与正方形共同的抽象四边形 Quadrangle 接口：

```
1 package cn.sitedev.lsp.graph.improved;
2
3 public interface QuadRangle {
4     long getWidth();
5
6     long getHeight();
7 }
```

修改长方形 Rectangle 类：

```
1 package cn.sitedev.lsp.graph.improved;
2
3 public class Rectangle implements QuadRangle {
4     private long height;
5     private long width;
6
7     public void setWidth(long width) {
8         this.width = width;
9     }
10
11     @Override
12     public long getWidth() {
13         return width;
14     }
15
16     public void setHeight(long height) {
17         this.height = height;
18     }
19
20     @Override
21     public long getHeight() {
22         return height;
23     }
24
25 }
```

修改正方形类 Square 类：

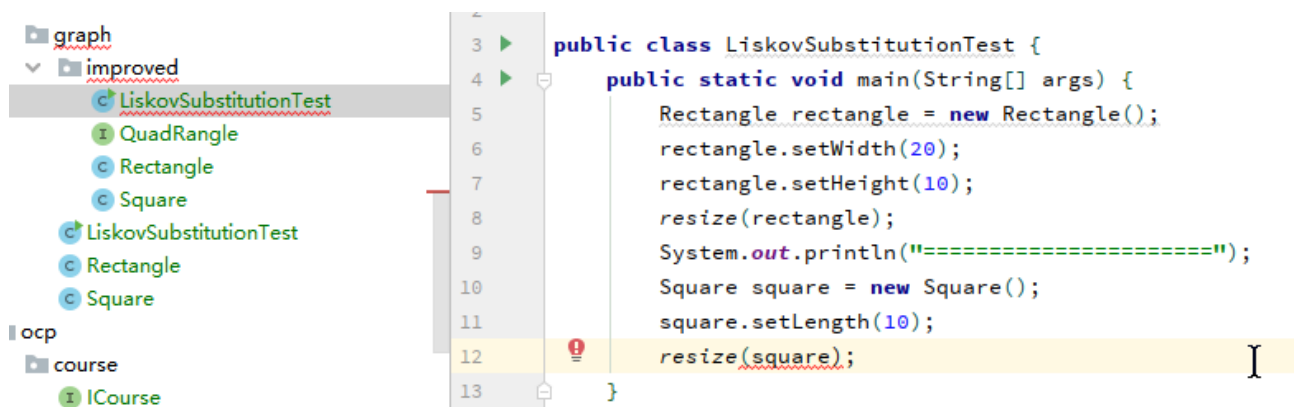
```
1 package cn.sitedev.lsp.graph.improved;
2
3
4 public class Square implements QuadRangle {
5     private long length;
```

```

6
7     public long getLength() {
8         return length;
9     }
10
11    public void setLength(long length) {
12        this.length = length;
13    }
14
15    @Override
16    public long getWidth() {
17        return length;
18    }
19
20    @Override
21    public long getHeight() {
22        return length;
23    }
24 }

```

此时，如果我们把 `resize()` 方法的参数换成四边形 `Quadrangle` 类，方法内部就会报错。



因为正方形 `Square` 已经没有了 `setWidth()` 和 `setHeight()` 方法了。因此，为了约束继承泛滥，`resize()` 的方法参数只能用 `Rectangle` 长方形。当然，我们在后面的设计模式课程中还会继续深入讲解。

## 7. 合成复用原则(CARP)

合成复用原则 ( Composite/Aggregate Reuse Principle, CARP ) 是指尽量使用对象组

合(has-a)/聚合(contanis-a)，而不是继承关系达到软件复用的目的。可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少。

继承我们叫做白箱复用，相当于把所有的实现细节暴露给子类。组合/聚合也称之为黑箱复用，对类以外的对象是无法获取到实现细节的。要根据具体的业务场景来做代码设计，其实也都需要遵循 OOP模型。还是以数据库操作为例，先来创建 DBConnection 类：

```
1 package cn.sitedev.carp.db;
2
3 public class DBConnection {
4     public String getConnection() {
5         return "mysql 数据库连接";
6     }
7 }
```

创建 ProductDao 类：

```
1 package cn.sitedev.carp.db;
2
3 public class ProductDao {
4     private DBConnection dbConnection;
5
6     public void setDbConnection(DBConnection dbConnection) {
7         this.dbConnection = dbConnection;
8     }
9
10    public void addProduct() {
11        String conn = dbConnection.getConnection();
12        System.out.println("使用" + conn + "增加产品");
13    }
14 }
```

这就是一种非常典型的合成复用原则应用场景。但是，目前的设计来说，DBConnection 还不是一种抽象，不便于系统扩展。目前的系统支持 MySQL 数据库

连接，假设业务发生变化，数据库操作层要支持 Oracle 数据库。当然，我们可以在 DBConnection 中增加对 Oracle 数据库支持的方法。但是违背了开闭原则。其实，我们可以不必修改 Dao 的代码，将 DBConnection 修改为 abstract，来看代码：

```
1 package cn.sitedev.carp.db.improved;
2
3 public abstract class DBConnection {
4     public abstract String getConnection();
5 }
6 }
```

然后，将 MySQL 的逻辑抽离：

```
1 package cn.sitedev.carp.db.improved;
2
3 public class MySQLConnection extends DBConnection {
4     @Override
5     public String getConnection() {
6         return "MySQL数据库连接";
7     }
8 }
```

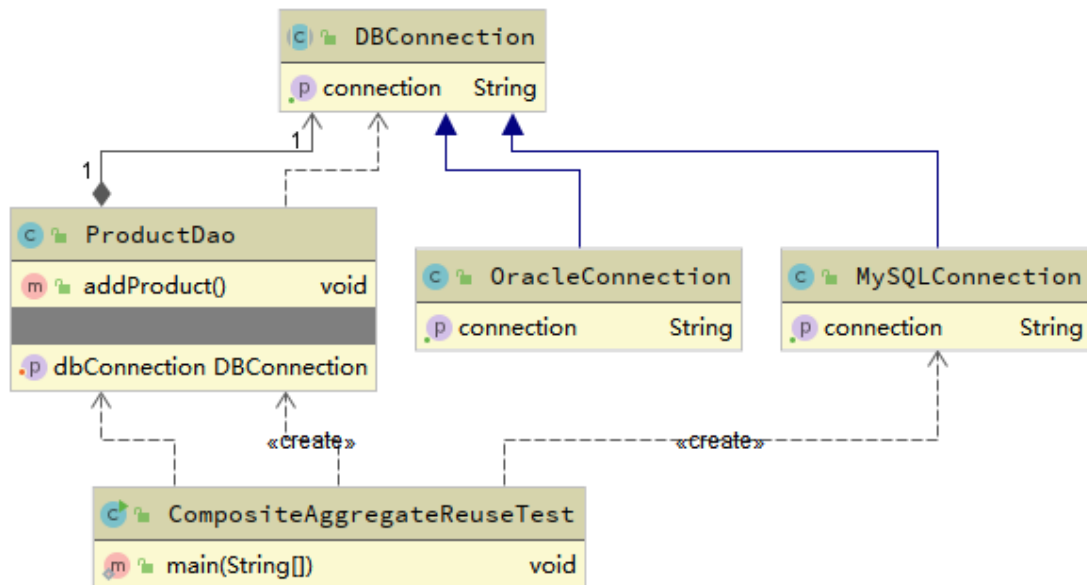
再创建 Oracle 支持的逻辑：

```
1 package cn.sitedev.carp.db.improved;
2
3 public class OracleConnection extends DBConnection {
4     @Override
5     public String getConnection() {
6         return "Oracle数据库连接";
7     }
8 }
```

...



具体选择交给应用层，来看一下类图：



## 8. 设计原则总结

学习设计原则，学习设计模式的基础。在实际开发过程中，并不是一定要求所有代码都遵循设计原则，我们要考虑人力、时间、成本、质量，不是刻意追求完美，要在适当的场景遵循设计原则，体现的是一种平衡取舍，帮助我们设计出更加优雅的代码结构。