Java 设计模式

作者	日期	版本	备注
姜鹏	2018.11	V0.1	软件设计职责创建型设计模式；其余将在下一期培训

说明：培训的工程源代码请上 github 获取
https://github.com/alanjiang/java-design/tree/master/java-design

目录

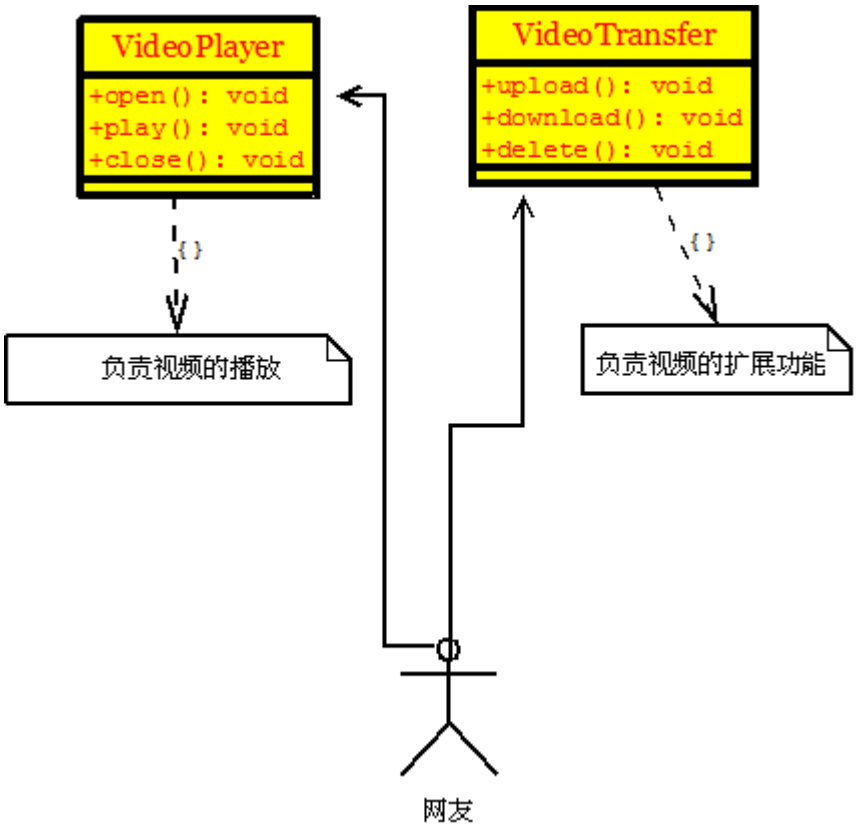
1 软件设计职责

1.1 单一职责

一个类应该专注于做一件事和仅有一个引起变化的原因。

优点：

- 有利于对象的稳定；职责越少、耦合度越弱、可扩展性越强。
- 提高可读性
- 提高代码可维护复用性；
- 降低因变更引起的风险



1.2 LSP 里氏替换原则(Liskov Substitution Principle)

其核心是子类继承父类。通用的行为特征高度抽象出来。

LSP 规范：

子类必须完全实现父类的方法；

子类可以有自己的个性；

覆盖实现父类方法时输入参数可以放大、缩小。

- 优点：

代码共享；

提高可重用性；

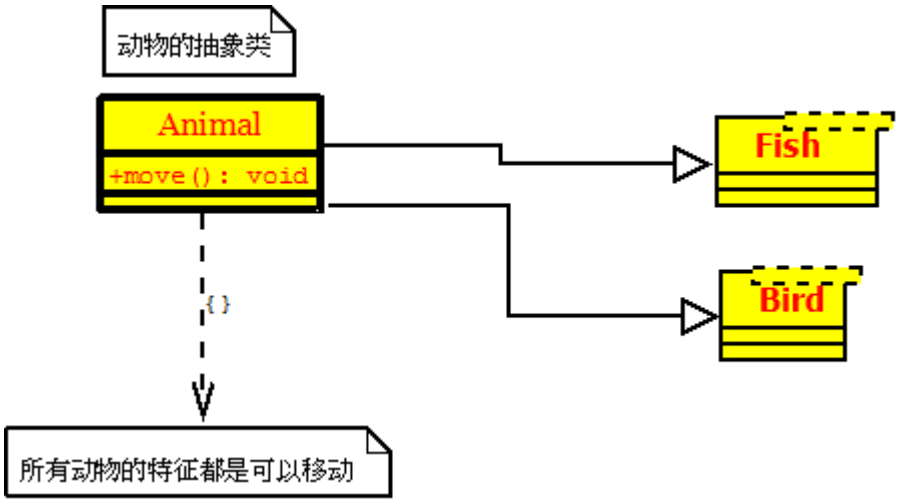
可扩展；

➢ 缺点：

继承是代码入侵式的；

子类受到父类的制约；

耦合性增强了。




```
public abstract class Animal {

     abstract void move();
}

public class Fish extends Animal {

    @Override
    void move() {

        System.out.println("----鱼儿游-----");

    }

}
```

```
public class Bird extends Animal {

    @Override
    void move() {
        System.out.println("----鸟儿飞-----");
    }

}
```

实例化鱼儿、鸟儿实例并调用 move()方法：

```
Animal fish=new Fish();
fish.move();

Animal bird=new Bird();
bird.move();
```

打印结果：

```
----鱼儿游-----
----鸟儿飞-----
```

里氏替换原则应用典范：

- 策略模式
- 组合模式
- 代理模式

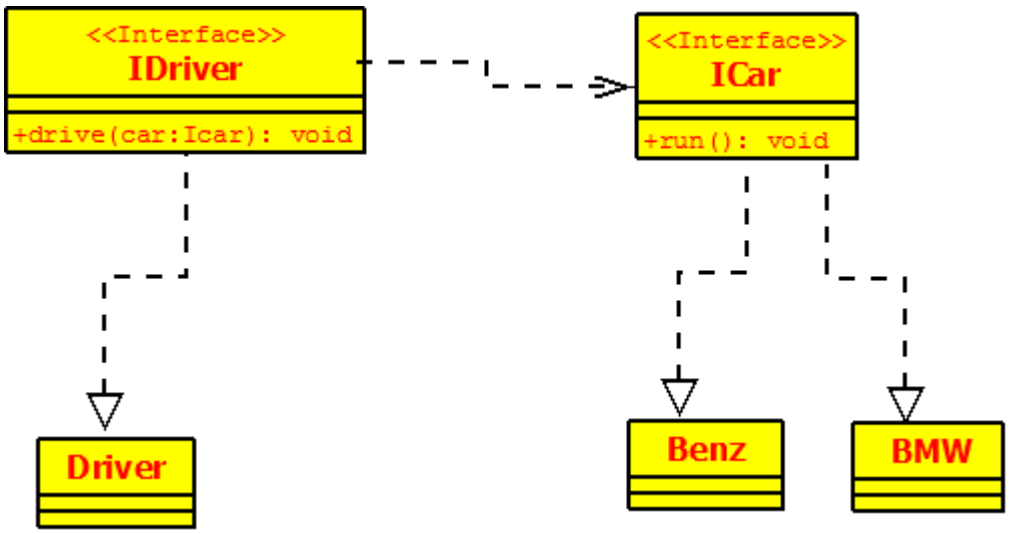
1.3 依赖倒置原则

提倡 OOD(Object-Oriented Design)面向接口编程。实现类之间不发生直接的依赖关系，其依赖关系通过接口或者抽象类产生。

- 优点：

减少类间的耦合；
提高系统的稳定性；
降低并行开发引起的风险；
提高代码可读和维护性。

司机和车的关系，司机只要会开车，就会开各种类型的车。司机不依赖于车的类型，司机和车的关系通过接口来实现依赖。



汽车和司机接口：

```
public interface ICar {

    void run();

    String getBrand();
}
```

```
public interface IDriver {

    void drive(ICar car);
}
```

汽车接口实现类，加一个车品牌以示不同的车。

```java
public class Car implements ICar {
    private String brand;

    public Car(String brand) {

        this.brand = brand;
    }

    public void run() {

        System.out.println(String.format("%s 车在跑....",brand));
    }

    public String getBrand() {
        // TODO Auto-generated method stub
        return brand;
    }

}
```

司机接口实现类新增一个姓名以区分不同司机。

```java
public class Driver implements IDriver {
    private String name;

    public Driver(String name) {

        this.name = name;
    }

    public void drive(ICar car) {

        System.out.println(String.format("--司机%s 开%s 车",name,car.getBrand()));
    }

}
```

现在让司机老王开奔驰车、老李来开宝马。

```
IDriver laowang=new Driver("老王");
ICar benz=new Car("Benz");
laowang.drive(benz);
benz.run();
//
IDriver laoli=new Driver("老李");

ICar bmw=new Car("宝马");

laoli.drive(bmw);
bmw.run();
```

打印输出：

```
--司机老王 开 Benz 车
Benz 车在跑....
--司机老李 开 宝马 车
宝马 车在跑....
```

1.4 接口隔离原则 (Interface SegragationPriciple) ISP

客户端不应该依赖它不需要的接口；
类的依赖应该建立在最小的接口上。
尽量使用原子接口、原子类。

实例 1：
实体：老虎、人、鸟、中国人、英国人、百灵鸟
行为：吃、行走、喝水、飞翔、写字
请设计接口。

不合理的设计，见图 1.4.1。鸟、人、老虎类全部实现了 IAnimal 接口。鸟、人、老虎都依赖 IAnimal 接口，类的依赖不是建立在最小的接口上。

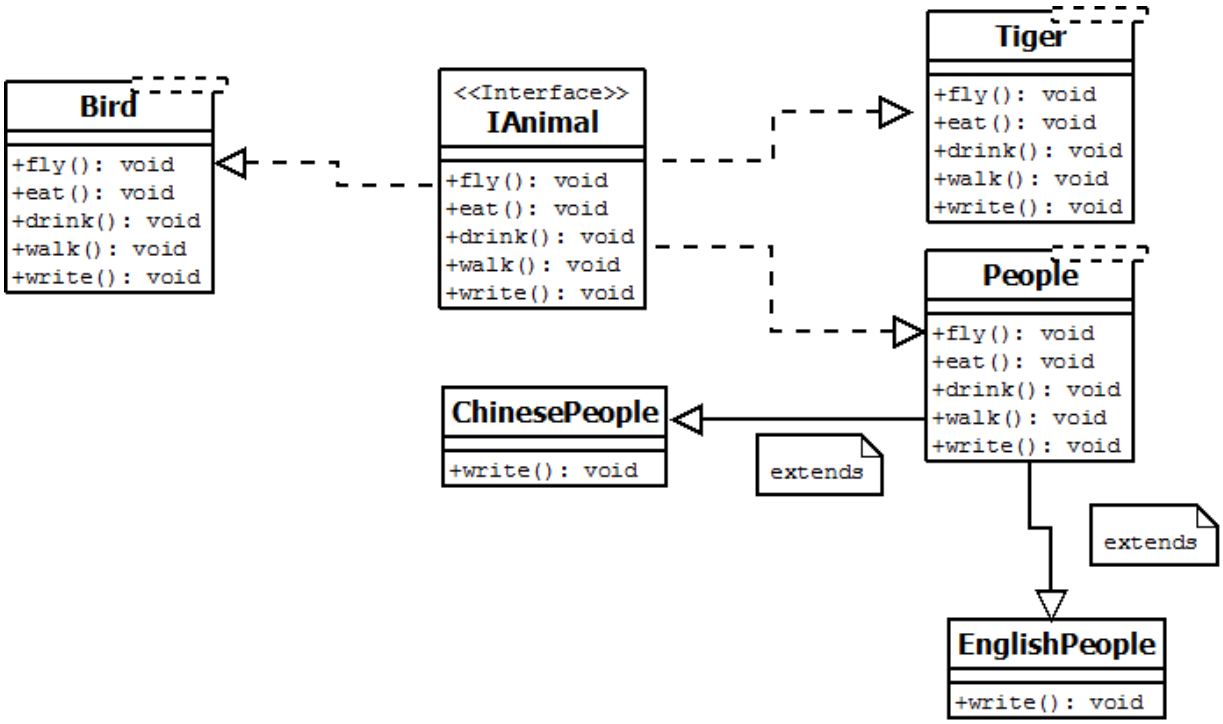


图 1.4.1 不满足接口隔离原则的设计

合理的设计见图 1.4.2。

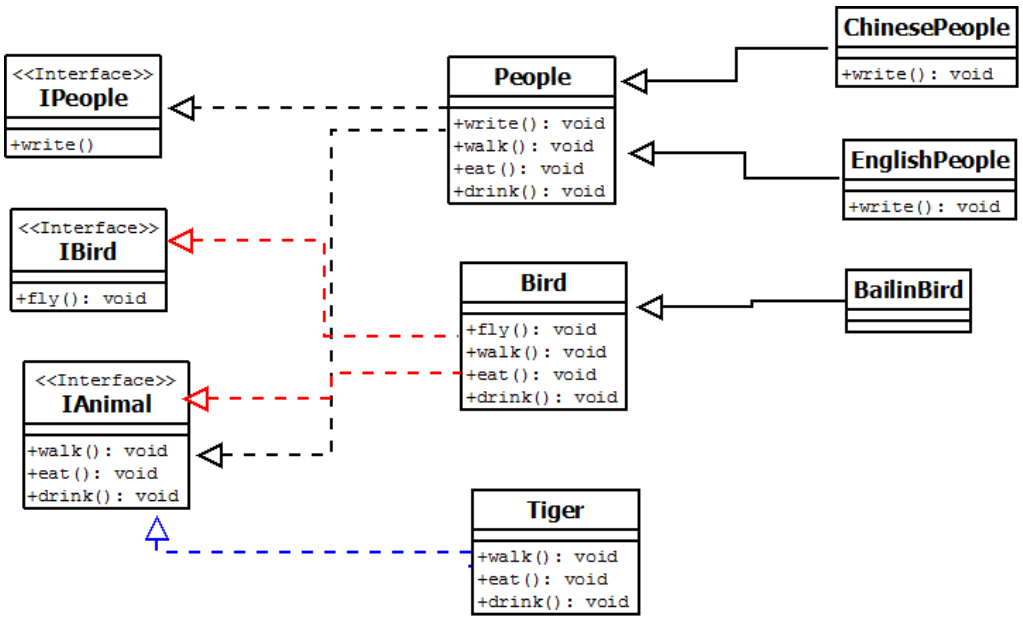


图 1.4.2 典型的接口隔离原则的设计

代码如下：

粒度最小的接口如下：

```
public interface IAnimal {
  void eat();
  void walk();
  void drink();
}
```

```
public interface IBird {

    void fly();
}
```

```
public interface IPeople {
    void write();
}
```

人类的实现类：

```
public class People implements IAnimal, IPeople {
    public void write() {
       System.out.println("---人写字---");
    }

    public void eat() {
        System.out.println("---人吃东西---");
    }

    public void walk() {
        System.out.println("---人行走---");
    }

    public void drink() {

        System.out.println("---人喝水东---");
    }

}
```

百灵鸟的实现类：

```
public class BailinBird implements IBird, IAnimal {

    public void eat() {
        System.out.println("---百灵鸟吃---");
    }

    public void walk() {
        System.out.println("---百灵鸟行走---");
    }

    public void drink() {
        System.out.println("---百灵鸟喝水---");
    }

    public void fly() {
        System.out.println("---百灵鸟飞翔---");

    }

}
```

老虎实现类：

```
public class Tiger implements IAnimal {

    public void eat() {
        System.out.println("---老虎吃---");

    }

    public void walk() {
        System.out.println("---老虎行走---");

    }

    public void drink() {
        System.out.println("---老虎喝水---");

    }

}
```

中国人和英国人写字有所不同：

```
public class ChinesePeople extends People {

    @Override
    public void write() {

        System.out.println("---中国人写汉字---");
    }

}
```

```
public class EnglishPeople extends People {

    @Override
    public void write() {

        System.out.println("---英国人国人写英文---");
    }

}
```

调用：

```
//只调用写字的动作，使用IPeople接口的实现
IPeople chinese_people=new ChinesePeople();
IPeople english_people=new EnglishPeople();
chinese_people.write();
english_people.write();
//只调用喝水、吃、行走的动作，使用IAnimal接口的实现
IAnimal chinese_animal=new ChinesePeople();
chinese_animal.drink();
chinese_animal.eat();
chinese_animal.walk();
```

结果：

```
---中国人写汉字---
---英国人国人写英文---
---人喝水---
---人吃东西---
---人行走---
```

1.5 LKP(Least Knowledge Principle) 最少知识原则(迪米特法则)

著名的火星登陆软件的指导设计原则。

原则：

- Only Talk to your immediate friends
- Don’t talk to strangers

一个类对自己需要耦合或者调用的类应该知道的最少。

核心思想：类之间的解耦、弱耦合。

该原则希望我们在设计中，不要让太多的类耦合在一起，免得修改系统中的一部分，会影响到其他部分。如果许多类之间相互依赖，那么这个系统就会变成一个易碎的系统，维护的成本也随之变得不可估量。

如：发动或停止“汽车”，“汽车”不直接与“发动机”发生相互作用，可以通过“车钥匙”间接与“发动机”发生作用。

见图：

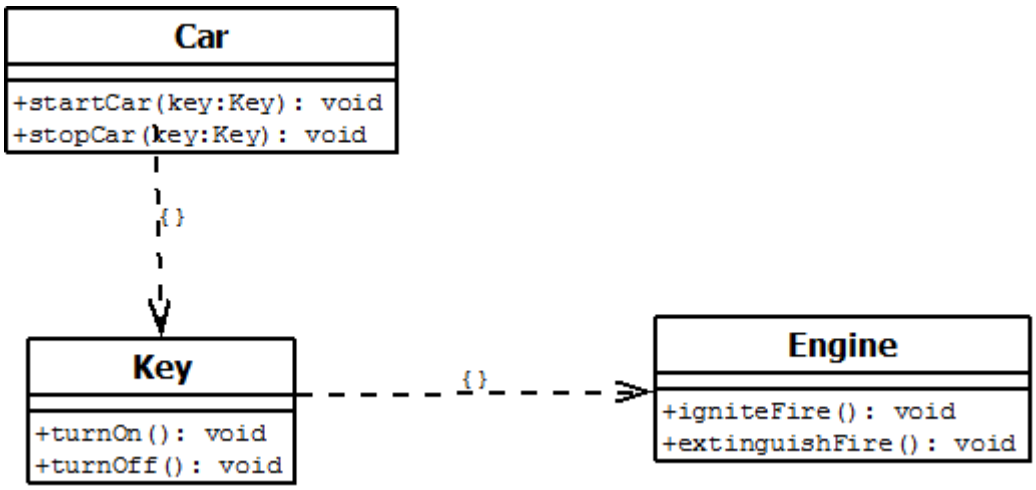


图 1.5.1 符合迪米特法则（LKP）的设计类图

代码如下：

汽车的启动和停止间接通过车钥匙的打开和关闭实现。

汽车对发动机工作细节一无所知。发动机类的变化不影响汽车类。

```
public class Car {

    public void startCar(Key key) {
        key.turnOn();
        System.out.println("车启动");
    }

    public void stopCar(Key key) {
        key.turnOff();
        System.out.println("车停止");
    }
}
```

车钥匙打开，发动机点火；车钥匙关闭，发动机熄火。

```
public class Key {

     Engine engine=new Engine();
     public void turnOn() {
         System.out.println("打开车钥匙");
         engine.igniteFire();
     }
     public void turnOff() {
         System.out.println("关闭车钥匙");
         engine.extinguishFire();
     }
}
```

发动机点火、熄火细节。

```java
public class Engine {
    /*点火火*/
    public void igniteFire() {
        System.out.println("发动机开始点火");
    }
    /*熄火*/
    public void extinguishFire() {
        System.out.println("发动机开始熄火");
    }
}
```

想在发动汽车、停止汽车。

```java
Car car=new Car();
Key key=new Key();
car.startCar(key);
car.stopCar(key);
```

输出结果：

```
打开车钥匙
发动机开始点火
车启动
关闭车钥匙
发动机开始熄火
车停止
```

依据迪米特法则|LKP 应用典范

- 外观模式
- 中介者模式

1.6 开闭原则 (Open-Closed Principle) OCP

软件实体对扩展开放，对修改关闭。

开闭原则对于已有的软件模块，特别是最重要的抽象层模块要求不能再修改，使得变化中的软件体统有一定的稳定性和延续性，便于维护；
软件系统面临新的需求时，系统的设计必须是稳定的，开闭原则可以通过扩展来提供新的行为、快速应对变化。

- 实例：

网上商城中商品光碟有按打折、原价销售、单价 150 以上减 5 元每件销售。请合理设计光碟实体类。

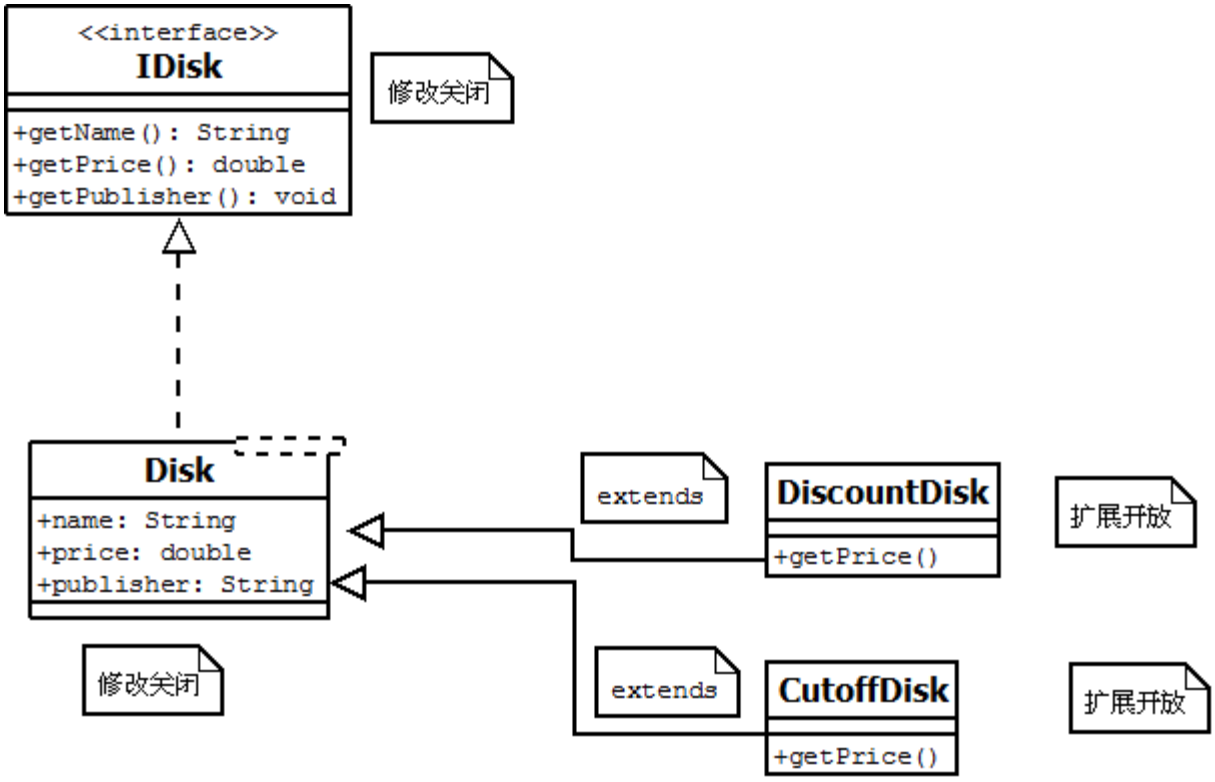


图 1.6.1 符合开闭原则的设计类图

首先定义光盘接口，确定光盘有名称、单价、出版商属性。

```
public interface IDisk {
    String getName();
    double getPrice();
    String publisher();
}
```

光盘实体实现光盘接口。

```java
public class Disk implements IDisk {
    private String name,publisher;
    private double price;
    public Disk(String name, String publisher, double price) {
        super();
        this.name = name;
        this.publisher = publisher;
        this.price = price;
    }
    public String getName() {
        // TODO Auto-generated method stub
        return this.name;
    }
    public double getPrice() {
        // TODO Auto-generated method stub
        return this.price;
    }
    public String publisher() {
        // TODO Auto-generated method stub
        return this.publisher;
    }
```

现在，光碟除部分按原价销售外、还有部分有按打折 8 折的和光碟单价 150 以上减 5 元的三大种类。

按照“开闭原则”中对修改关闭的原则，不可直接修改 IVideo 接口和 Video 类，依据“对扩展开放“原则，可以增加 Video 类的子类 DiscountDisk 和 CutoffDisk 来完成。见图：1.6.1。
打折光盘类 DiscountDisk，覆写父类 getPrice() 在单价的基础上乘以 0.8 实现打八折。

```java
public class DiscountDisk extends Disk {

    public DiscountDisk(String name, String publisher, double price) {
        super(name, publisher, price);
    }
    @Override
    public double getPrice() {
        return super.getPrice()*0.8;
    }

}
```

同理， 减价光盘类 CutoffDisk 覆写父类 getPrice() 对单价大于 150 元的减去 5 元。

```java
public class CutoffDisk extends Disk {
    public CutoffDisk(String name, String publisher, double price) {
        super(name, publisher, price);
    }
    @Override
    public double getPrice() {
        return (super.getPrice()>150)? super.getPrice()-5:super.getPrice();
    }
}
```

计算一下老刘买了 4 件光盘，总共节花了多少钱。

```java
List<IDisk> disks=new ArrayList<IDisk>();
IDisk disk1=new Disk("班得瑞","好莱坞",12);
IDisk disk2=new DiscountDisk("Beyond乐队","滚石唱片",76);
IDisk disk3=new CutoffDisk("许嵩专辑","北京唱片",94);
IDisk disk4=new CutoffDisk("王菲专辑","滚石唱片",218);
//老王将4件光碟加入购物车集合
disks.add(disk1);
disks.add(disk2);
disks.add(disk3);
disks.add(disk4);
//前台结账:
double sum=0d;
for(IDisk disk:disks) {
    System.out.println(">>>>"+disk.toString());
    sum+=disk.getPrice();
}
System.out.println(">>>总共花钱:"+sum);
```

输出：

```
>>>>Disk [name=班得瑞, publisher=好莱坞, price=12.0]
>>>>DiscountDisk [getPrice()=60.800000000000004, getName()=Beyond乐队, publisher()=滚石唱片]
>>>>CutoffDisk [getPrice()=94.0, getName()=许嵩专辑, publisher()=北京唱片]
>>>>CutoffDisk [getPrice()=213.0, getName()=王菲专辑, publisher()=滚石唱片]
>>>总共花钱:379.8
```

以上实例，在不修改接口和基类的情况下，经过灵活的扩展，轻松解决了各种打折、减价光碟的销售情况。值得读者细细品味。

2 常见设计模式

2.1 创建型模式

创建型模式共有 5 种：

- 单例模式
- 工厂方法
- 抽象工厂模式
- 建造者模式
- 原型模式

2.1.1 单例模式

使用场合：一个类只有一个实例存在，减少系统创建实例的开销，单线程操作的场合。单例不需要实现 Serializable, Cloneable 接口。反系列化和克隆均可实例化一个新的对象。

分为饿汉式单例和懒汉式单例。

- 饿汉式单例：类加载时进行对象实例化。

```
public class ErhanSingleton {
    private static ErhanSingleton instance=new ErhanSingleton();
    private ErhanSingleton() {
    }
    public static ErhanSingleton getInstance() {
        return instance;
    }
}
```

- 懒汉式单例：第一次引用类时才进行对象实例化。

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

两种单例共同点：

私有构造函数，防止外部实例化对象。

2.1.2 工厂方法模式

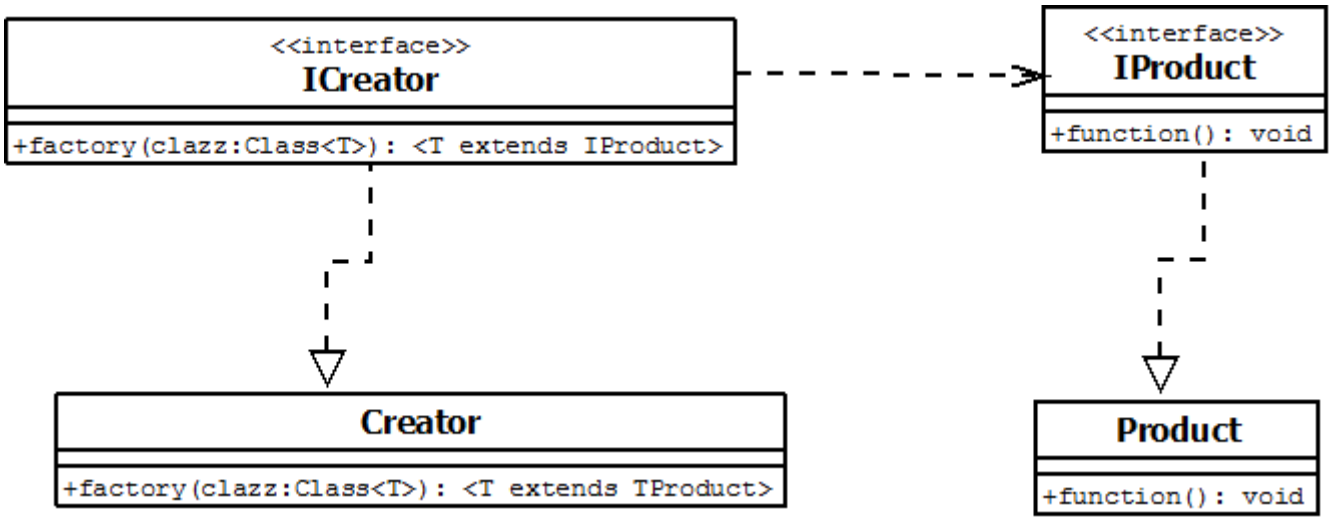


图 2.1.2.1 工厂方法模式设计类图

工厂方法模式中四个角色：

抽象工厂角色：创建对象的工厂类必须实现这个角色。
具体工厂角色：工厂方法创建抽象产品。
抽象产品角色：定义产品的功能。
具体产品：实现抽象工厂声明的接口，被创建的具体实例。

定义产品抽象接口：

```
public interface IProduct {

    void function();
}
```

具体产品的实现：

```
public class Product implements IProduct {

    @Override
    public void function() {
        System.out.println("---function goes---");
    }

}
```

定义抽象工厂：

抽象工厂工厂方法一般使用泛型，方便创建不同的具体产品。

```
public interface ICreator {

    <T extends IProduct> T factory(Class<T> clazz);
}
```

定义具体工厂，实现工厂方法：

```
public class Creator implements ICreator {

    @Override
    public <T extends IProduct> T factory(Class<T> clazz) {
        IProduct product=null;
        try {
            product=(IProduct) Class.forName(clazz.getName()).newInstance();
        } catch (InstantiationException | IllegalAccessException |ClassNotFoundException e) {
            e.printStackTrace();
        }
        return (T)product;

    }

}
```

注意上面的根据 class 名称实例化对象的用法。
Class.forName(clazz.getName()).newInstance();

下面使用抽象工厂方法创建实例：

```
ICreator creator=new Creator();
IProduct product=creator.factory(Product.class);
product.function();
```

运行结果：

```
---function goes---
```

当需要实例化多个具体产品时，抽象工厂非常创建实例非常灵活。

2.1.3 原型模式

原型设计模式设计三个角色：

客户角色： 负责创建对象。
抽象原型角色：JAVA 接口或者抽象类。
具体原型角色：被复制的对象，必须实现抽象原型接口。

优点：
内存二进制流的复制，比直接创建对象性能好。特别适合循环体内产生大量对象。

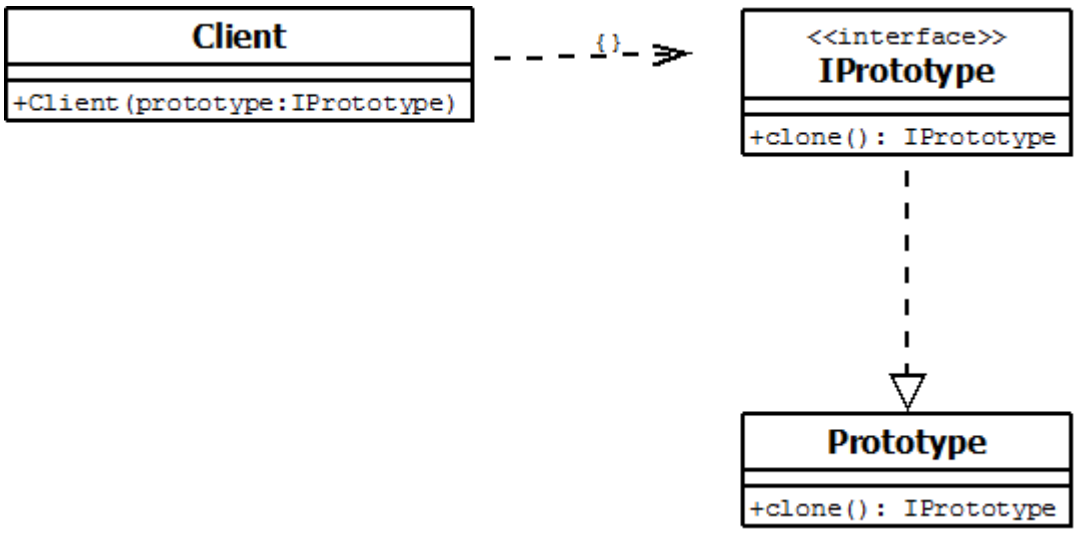


图 2.1.3.1 原型设计模式设计类图

应用实例： 发送 50000 条促销短信。

代码演示：

短信实体实现 Cloneable 接口：调用父类的 clone() 方法。如下：

```
public class ShortMessage implements Cloneable {
    private String title;
    private String mobile;//different
    private String content;
    private String clientName;//different
    private String companyName;
    public ShortMessage(String title, String companyName) {
     this.title = title;
     this.companyName = companyName;
}

    @Override
    protected Object clone() throws CloneNotSupportedException {

          return super.clone();
    }
```

下面模仿发给 50000 个顾客发促销短信。

由于 短信的标题和公司名称都是固定的。可以构造一个，其他的手机号、内容不同而已。下面比较了在循环体内克隆和不克隆的方式创建对象性能的差异。克隆整个过程 4423 毫秒、不可隆耗时 5539 毫秒。

```
long start=new Date().getTime();
ShortMessage message=new ShortMessage("2019双11促销", "天猫商城");
ShortMessage sendMessage=null;
for(int i=0;i<50000;i++) {
    //克隆浅复制对象，性能更高:
    //克隆
    //sendMessage=(ShortMessage)message.clone();//4423 ms
    //不克隆
    sendMessage=new ShortMessage("2019双11促销", "天猫商城");//5539 ms
    sendMessage.setClientName("A"+i);
    if(i<10) {
        sendMessage.setMobile("1"+i+"30568907"+i);
    }else if(i>10 && i<100) {
        sendMessage.setMobile("1"+i+"30568903");
    }
    else if(i>100 && i<1000) {
        sendMessage.setMobile("1"+i+"3056890");
    }
    else if(i>1000 && i<10000) {
        sendMessage.setMobile("1"+i+"305689");
    }else {
        sendMessage.setMobile("1"+i+"30568");
    }
    System.out.println(message.equals(sendMessage));
    sendMessage.setContent("尊敬的"+sendMessage.getClientName()+",双11所有商品低至7折，欢迎来抢货！");
    sendShortMessage(sendMessage);
}
```

深克隆和浅克隆：

实例代码见：

com.java.lessongs.design.prototype.deepclone
com.java.lessongs.design.prototype.hollowclone

2.2 结构模式

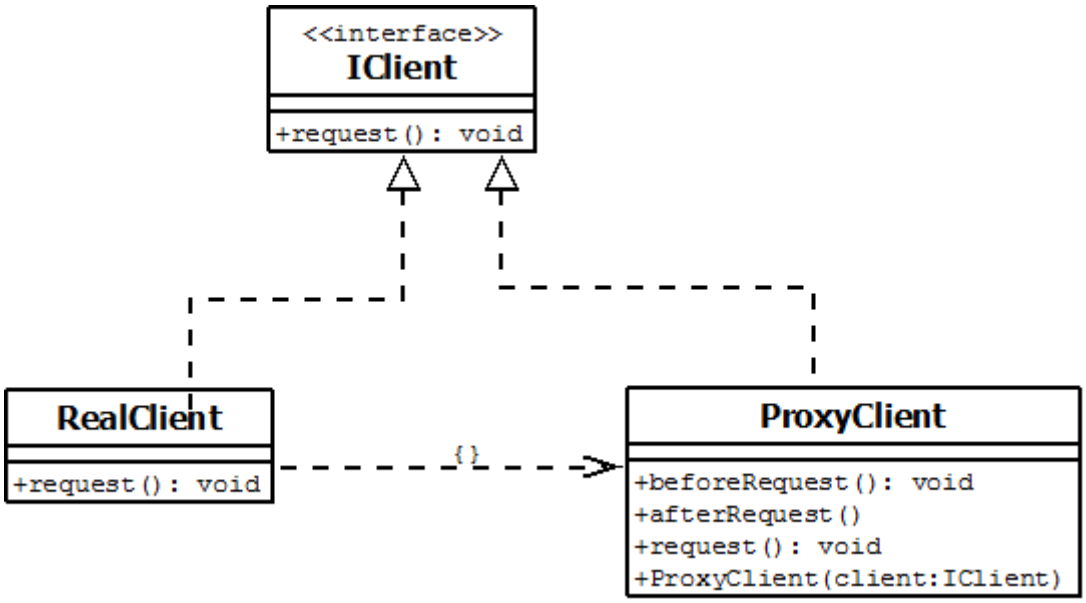
2.2.1 代理模式

代理模式提供 3 个角色：

抽象角色：真实角色和代理角色的共同接口。
代理角色：委托类、代理类。真实角色处理完毕前后，做一些预处理或善后处理。
真实角色：业务逻辑的具体执行者。

代理客户端和真实客户端均实现了代理接口。下图是代理模式的原理类图。
在代理角色中，一般会构造注入或者依赖注入真实角色。代理角色实现接口方法时，会调用真实角色的接口方法，并且在调用真实角色的接口方法前后调用自身的方法。

如：Spring AOP 采用了代理模式。



2.2.1.1 代理模式设计类图

代码演示：

抽象角色接口：

```
public interface IClient {

    void resquest();
}
```

真实角色：

```
public class RealClient implements IClient {

    @Override
    public void resquest() {

        System.out.println(">>>>真实客户端调用<<<<<");

    }

}
```

代理角色：通常代理角色中引用真实角色。本实例采用构造注入的方式。

```
public class ProxyClient implements IClient {
   private IClient realClient;
    public ProxyClient(IClient realClient) {
        this.realClient = realClient;
    }
    public void beforeRequest() {
        System.out.println(">>>>代理客户端调用真实客户端前<<<<<");
    }

    @Override
    public void resquest() {
        beforeRequest();
        realClient.resquest();
        afterRequest();
    }

    public void afterRequest() {
        System.out.println(">>>>代理客户端调用真实客户端前<<<<<");
    }
}
```

调用：

```
IClient realClient=new RealClient();

IClient proxyClient=new ProxyClient(realClient);

proxyClient.resquest();
```

输出结果：

```
>>>>代理客户端调用真实客户端前<<<<<
>>>>真实客户端调用<<<<<
>>>>代理客户端调用真实客户端前<<<<<
```

使用代理模式的优势：

- **中介隔离作用：**

一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

- **增加功能的同时遵循开闭原则：**

代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。
例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要修改已经封装好的委托类。

2.2.2 装饰模式

装饰模式有以下几个角色：

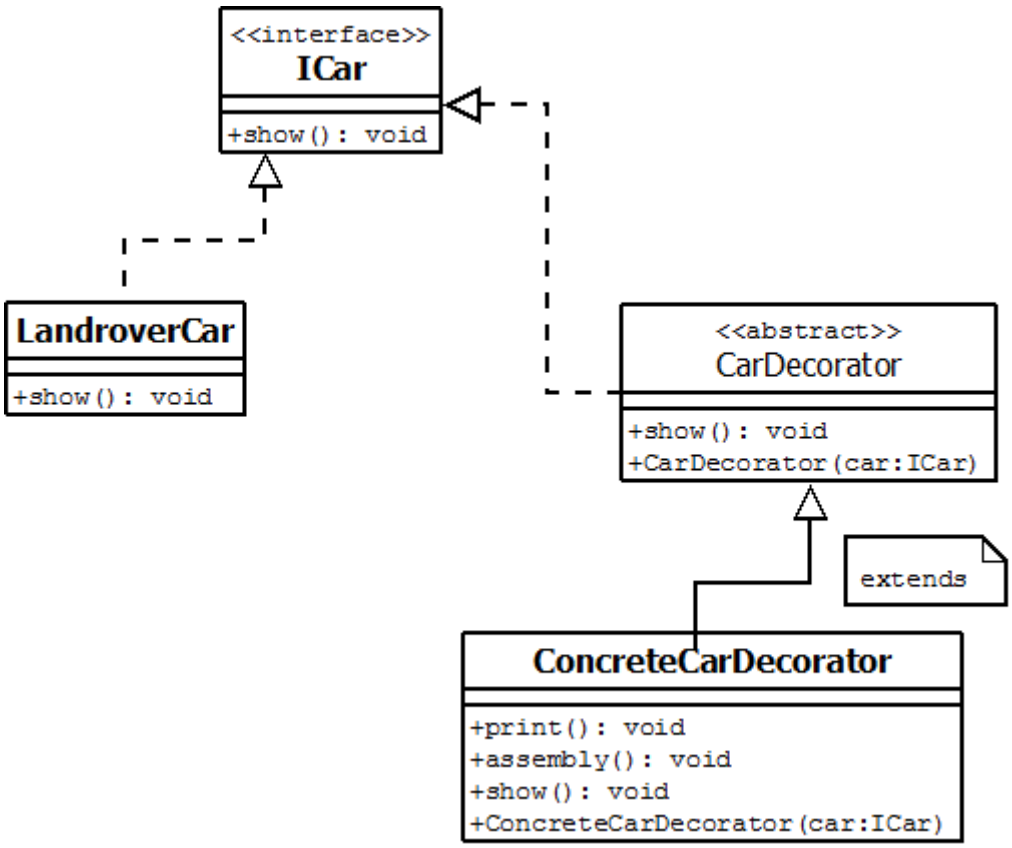
抽象构件：规范要装饰的对象接口。ICar
具体构件：要装饰的对象接口的实现。需要装饰的原始类。LandroverCar
装饰角色：抽象类，实现抽象构件的接口,持有抽象接口对象的引用。CarDecorator
具体装饰角色：负责对构件对象进行装饰； ConcreteCarDecorator

优点：

- 装饰是集成的一个替代方案。不管装饰多少层，返回的始终是抽象构件接口类。
- 动态扩展实现类的功能。

缺点：
多层装饰比较复杂。

下面对路虎车进行装饰，默认路虎车是黑色的，现对齐重新喷漆并完成组装。

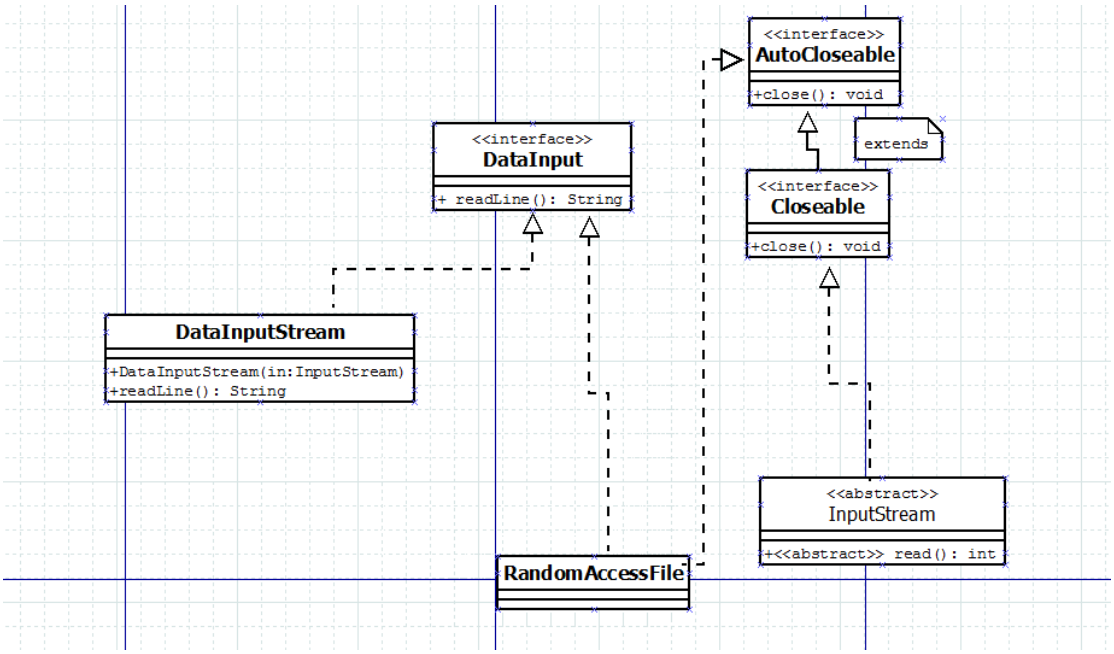


调用演示：

```
ICar landover=new LandroverCar();
CarDecorator decorator=new ConcreteCarDecorator(landover);
decorator.show();
```

DataInput, InputStream 类使用了部分这种设计模式：

看下面的类图：



调用演示：

```
InputStream is = new ByteArrayInputStream("ABC".getBytes());
OutputStream os = new ByteArrayOutputStream();
DataOutput dos = new DataOutputStream(os);


DataOutput out=new RandomAccessFile("F:/a.txt","rw");
out.writeChars("Hello World");
System.out.println(">>>write done<<<");

FileInputStream fis=new FileInputStream("F:/a.txt");

DataInput input=new RandomAccessFile("F:/a.txt","rw");
System.out.println(">>>read <<<"+input.readLine());
```

2.2.3 适配器模式

2.2.4 组合模式

2.2.5 桥梁模式

2.2.6 外观模式

2.2.7 享元模式

2.3 行为模式

2.3.1 模板方法模式

2.3.2 命令模式

2.3.3 策略模式

2.3.4 迭代器模式

2.3.5 中介者模式

2.3.6 备忘录模式

2.3.7 状态模式

2.4 混合设计模式

命令链模式

工厂策略模式

观察中介者模式