设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。
使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。
项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

一、设计模式的分类

创建型模式，工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

二、Java的设计模式

1、工厂方法模式（Factory Method）

工厂方法模式分为三种：普通工厂模式　多个工厂方法模式　静态工厂方法模式

1.1、普通工厂模式，就是建立一个工厂类，对实现了同一接口的产品类进行实例的创建

例子:

```
//发送短信和邮件的接口
public interface Sender {
    public void Send();
}

//发送邮件的实现类
public class MailSender implements Sender {
    public void Send() {
        System.out.println("发送邮件!");
    }
}
//发送短信的实现类
public class SmsSender implements Sender {
    public void Send() {
        System.out.println("发送短信!");
    }
}

//创建工厂类
```

```
public class SendFactory {
    //工厂方法
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}


//测试类
public class FactoryTest {

    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produce("sms");
        sender.Send();
    }
}
```

1.2、多个工厂方法模式 是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
//将上面的代码做下修改，改动下SendFactory类就行
//这个就不用根据用户传的字符串类创建对象了
public class SendFactory {

    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}

//测试类
public class FactoryTest {

    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.Send();
```

```
    }
}
```

1.3、静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {

    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}

//测试类
public class FactoryTest {

    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.Send();
    }
}
```

2、抽象工厂模式（Abstract Factory）

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

例子:

```
//发送短信和邮件的接口
public interface Sender {
    public void Send();
}

//发送邮件的实现类
public class MailSender implements Sender {
    public void Send() {
        System.out.println("发送邮件!");
    }
}
//发送短信的实现类
public class SmsSender implements Sender {
```

```
    public void Send() {
        System.out.println("发送短信!");
    }
}

//给工厂类一个接口
public interface Provider {
    public Sender produce();
}
//两个工厂的实现类
public class SendMailFactory implements Provider {
    public Sender produce(){
        return new MailSender();
    }
}
public class SendSmsFactory implements Provider{
    public Sender produce() {
        return new SmsSender();
    }
}

//测试类
public class Test {

    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.Send();
    }
}
```

注:这个模式的好处就是，如果你现在想增加一个功能：发送及时信息，则只需做一个实现类实现Sender接口，同时做一个工厂类，实现Provider接口就可以了，无需去改动现成的代码。这样做，拓展性较好

3、单例模式（Singleton）

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。

例子：

```
//简单的单例类  饿汉模式
```

```
public class Singleton {

    /* 持有私有静态实例，防止被引用*/
    private static Singleton instance = new Singleton();

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 静态工程方法，返回Singleton实例 */
    public static Singleton getInstance() {
        return instance;
    }

    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
    private Object readResolve() {
        return instance;
    }
}
```

这个类是可以实现单例模式的，但是存在不少问题,比如在类中不管用户是否要使用该类的对象,就先创建好了一个实例放在内存中。

```
//简单的单例类 懒汉模式
public class Singleton {

    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 */
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 静态工程方法，创建实例 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
    private Object readResolve() {
        return instance;
    }
}
```

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

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

但是，synchronized作为修饰符在方法上使用,在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

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

解决了之前提到的问题，将synchronized关键字加在了方法内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创建对象的时候才需要加锁，性能有一定的提升。

4、建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性。建造者模式主要用于“分步骤构建一个复杂的对象”，在这其中“分步骤”是一个稳定的算法，而复杂对象的各个部分则经常变化。因此， 建造者模式主要用来解决“对象部分”的需求变化。 这样可以对对象构造的过程进行更加精细的控制。

例子:

```
//CPU接口
public interface CPU {

}
//Inter的cup
class IntelCPU implements CPU{

}
//AMD的cpu
class AMDCPU implements CPU{
```

```
}

//内存接口
public interface Memory {

}
//金士顿内存
class KingstonMemory implements Memory{

}
//三星内存
class SamsungMemory implements Memory{

}

//主板内存
public interface Mainboard {

}
//华硕主板
class AsusMainboard implements Mainboard{

}
//技嘉主板
class GaMainboard implements Mainboard{

}

//计算机
public class Computer {
    private CPU cpu;
    private Memory memory;
    private Mainboard mainboard;
    get/set
}

//计算机的builder的接口
public interface ComputerBuilder {
    public void buildCPU();
    public void buildMemory();
    public void buildMainboard();
    public Computer getComputer();
}


//联想电脑的builder
public class LenoveComputerBuilder implements ComputerBuilder {
   private Computer lenoveComputer;
   public LenoveComputerBuilder(){
       lenoveComputer = new Computer();
```

```
    }
    public void buildCPU() {
        lenoveComputer.setCpu(new IntelCPU());
    }
    public void buildMemory() {
        lenoveComputer.setMemory(new KingstonMemory());
    }
    public void buildMainboard() {
        lenoveComputer.setMainboard(new AsusMainboard());
    }
    public Computer getComputer() {
        return lenoveComputer;
    }
}

//惠普电脑的builder
public class HPComputerBuilder implements ComputerBuilder {
    private Computer HPComputer;

    public HPComputerBuilder(){
        HPComputer = new Computer();
    }
    public void buildCPU() {
        HPComputer.setCpu(new AMDCPU());
    }
    public void buildMemory() {
        HPComputer.setMemory(new SamsungMemory());
    }
    public void buildMainboard() {
        HPComputer.setMainboard(new GaMainboard());
    }
    public Computer getComputer() {
        return HPComputer;
    }
}

//Director类(导演)
//指导如何具体的创造电脑
public class Director {
    private ComputerBuilder builder;
    public Director(ComputerBuilder builder) {
        this.builder = builder;
    }
    //用户自定义的自造顺序 具体指导各种builder如何创建电脑
    public void construct() {

            builder.buildCPU();
            builder.buildMemory();
            builder.buildMainboard();
    }
}
```

```
//测试类
public class Test {
    public static void main(String[] args) {
        Computer lenoveComputer = null;
        ComputerBuilder lenoveComputerBuilder = new LenoveComputerBuilder();
        Director director = new Director(lenoveComputerBuilder);
        director.construct();
        lenoveComputer = lenoveComputerBuilder.getComputer();
        System.out.println(lenoveComputer);
    }
}
```

从这点看出，建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建适合对象的多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定。

例如一个Person类是由头、身体、脚三个对象组成，那么我们在建造者模式中就要先分别创造出这三个部分然后再把他们组装成一个Person对象。

5、原型模式（Prototype）

原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。在Java中，复制对象是通过clone()实现的，先创建一个原型类：

```
public class Prototype implements Cloneable {

    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
}
```

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，说明这个方法实现并不是使用java语言。

在这里先认识俩个概念 : 浅复制 深复制

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

```
public class Prototype implements Cloneable, Serializable {

    private static final long serialVersionUID = 1L;
```

```
    private String string;
    //这个是在下面声明的一个类
    private SerializableObject obj;

    /* 浅复制 */
    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }

    /* 深复制 */
    public Object deepClone() throws IOException, ClassNotFoundException {

        /* 写入当前对象的二进制流 */
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);

        /* 读出二进制流产生的新对象 */
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

    public SerializableObject getObj() {
        return obj;
    }

    public void setObj(SerializableObject obj) {
        this.obj = obj;
    }

}

class SerializableObject implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

上面是五种是创建型模式,接下来看一下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式

6、适配器模式（Adapter）

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式：

核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口是Targetable，通过Adapter类，将Source的功能扩展到Targetable里

例子：

```
public class Source {
        public void method1() {
                System.out.println("this is original method!");
        }
}
public interface Targetable {

        /* 与原类中的方法相同 */
        public void method1();

        /* 新的方法 */
        public void method2();
}

//类Source和接口Targetable因为不兼容，导致不能在一起工作
//适配器Adapter则可以在不改变源代码的基础上解决这个问题
//这样Targetable接口的实现类Adapter的对象即使Targetable类型,也能访问到Source中的方法
public class Adapter extends Source implements Targetable {
        public void method2() {
            System.out.println("this is the targetable method!");
        }
}

//测试类  这样Targetable接口的实现类就具有了Source类的功能。
public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```

对象的适配器模式

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题

例子:

```
//只需要修改Adapter类的源码即可：
public class Wrapper implements Targetable {

        private Source source;

        public Wrapper(Source source){
                this.source = source;
        }
        public void method2() {
                System.out.println("this is the targetable method!");
        }
        public void method1() {
                source.method1();
        }
}

//测试类  输出与第一种情况一样，只是使用的适配方法不同而已。
public class AdapterTest {

    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

我们这GUI这个章节应该是见过不少的监听器接口的适配器类:XxxxAdapter

例子:

```
public interface Sourceable {
        public void method1();
        public void method2();
}
//抽象类
public abstract class Wrapper implements Sourceable{
        public void method1(){}
        public void method2(){}
}
```

之后在我们写的子类中需要什么方法去重写什么方法就可以了,就不需要把接口中的所有方法都实现了

三种情况适配器模式的总结:

类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

7、装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是【动态】的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例

这里的动态指的是用户可以根据自己的需求把之前定好的功能任意组合。

JDK中的IO流部分就是典型的使用了装饰模式,回忆一下BufferedReader对象的是如何创建的

例子:

```
//功能接口
public interface Action {
	public void go();
}
//被装饰的类  就是需要我们装饰的目标
public class Person implements Action{
	public void go() {
		System.out.println("我在走路");
	}
}

//抽象的装饰类
public abstract class Decorator implements Action{
	private Action action;
	public Decorator(Action action) {
		this.action = action;
	}
	public void go() {
		this.action.go();
	}
}

//具体的装饰类 可以添加一个听音乐的功能
public class ListenDecorator extends Decorator{
	public ListenDecorator(Action action) {
		super(action);
```

```
        }
        public void go() {
            listen();//可以在go方法【前】添加一个听音乐的功能
            super.go();
        }
        public void listen(){
            System.out.println("我在听音乐");
        }

}

//具体的装饰类 可以添加一个休息的功能
public class RelaxDecorator extends Decorator{
        public RelaxDecorator(Action action) {
            super(action);
        }
        public void go() {
            super.go();
            relax();//可以在go方法【后】添加一个休息的功能
        }
        public void relax(){
            System.out.println("我在休息");
        }
}

//测试类
public class Test {
        /*用户可以根据需求 任意给go方法添加听音乐或者休息的功能*/
        //Action a = new Person();
        //Action a = new ListenDecorator(new Person());
        //Action a = new RelaxDecorator(new Person());
        //Action a = new RelaxDecorator(new ListenDecorator(new Person()));
        Action a = new ListenDecorator(new RelaxDecorator(new Person()));
        a.go();
}
```

装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。

缺点：产生过多相似的对象，不易排错！

8、代理模式（Proxy）

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。

例子：

```
//公共接口
public interface Sourceable {
    public void method();
}
//目标类/被代理类
public class Source implements Sourceable {
    public void method() {
        System.out.println("the original method!");
    }
}

//代理类
public class Proxy implements Sourceable {

    private Source source;
    public Proxy(Source source){
        this.source = source;
    }
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}

//测试类
public class ProxyTest {
    public static void main(String[] args) {
        Source target = new Source();
        Sourceable proxy = new Proxy(target);
        proxy.method();
    }
}
```

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

注意：装饰模式和代理模式在很多情况下,大部分代码都是类似的，但是这俩种设计的意图是不一样的,装饰模式是增强被包装对象的功能,代理模式是控制被代理对象的行为

例如一块代码，如果被描述为使用了装饰模式,那么我们就知道设计的意图是增加被包装对象的功能,如果被描述为使用了代理模式,那么我们就知道设计的意图是控制被代理对象的行为，虽然这俩种情况下他们的代码结构基本相同.

装饰器模式 ： 能动态的新增或组合对象的行为。

代理模式 ： 为目标对象提供一种代理以便控制对这个对象的访问.

装饰模式是“新增行为”，而代理模式是“控制访问”。

1.装饰模式：对被装饰的对象增加额外的行为

如：杯子生产线，杯子必须可以装水，在生产线上可以给杯子涂颜色，加杯盖，但要保证杯子可以装水。

2.代理模式：对被代理的对象提供访问控制。

如：客户网上商城订购商品，网上商城是厂家的代理，网上商城可以帮客户完成订购商品的任务，但是商城可以对商品进行控制，不交钱不给商品，人不在不给商品，也可以赠送你额外的礼品，代金券。

对代理模式的一些重要扩展

用户tom---买--->商品

由于各种原因导致不是很方便购买,所以就找代购

用户tom---找--->代购者zs---买--->商品

那么在代理模式中,用户tom就是目标对象,代购者zs就是代理对象

创建目标对象的类叫目标类或者被代理类

创建代理对象的类加代理类

代理类可分为两种：

静态代理类：

由程序员创建或由特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。

动态代理类：在程序运行时，运用反射机制动态创建而成。

与静态代理类对照的是动态代理类，动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包下面的Proxy类和InvocationHandler 接口提供了生成动态代理类的能力。

静态代理: staticProxy

例子:

//公共接口

```
public interface HelloService {
	void sayHello();
}
//委托类
public class HelloServiceImpl implements HelloService{
	public void sayHello() {
		System.out.println("hello world");
	}
}
//代理类
public class HelloServiceProxy implements HelloService{
	private HelloService target;
	public HelloServiceProxy(HelloService target) {
		this.target = target;
	}
	public void sayHello() {
		System.out.println("log:sayHello马上要执行了...");
		target.sayHello();
	}

}
//测试类
public class Test {
	public static void main(String[] args) {
		//目标对象
		HelloService target = new HelloServiceImpl();
		//代理对象
		HelloService proxy = new HelloServiceProxy(target);
		proxy.sayHello();
	}

}
```

JDK的动态代理： dynamicProxy

例子:

```
//Student类
public class Student {
	private long id;
	private String name;
	private int age;
	get/set
}
//日志类
public class StudentLogger {
	public void log(String msg){
		System.out.println("log: "+msg);
	}
}
//Service接口 处理学生的相关业务
```

```
public interface IStudentService {
	void save(Student s);
	void delete(long id);
	Student find(long id);
}
//接口的一个简单实现
public class StudentServiceImpl implements IStudentService {
	public void delete(long id) {
		System.out.println("student is deleted...");
	}
	public Student find(long id) {
		System.out.println("student is found...");
		return null;
	}
	public void save(Student s) {
		System.out.println("student is saved...");
	}
}

//InvocationHandler接口的实现类
//JDK动态代理中必须用到的接口实现
public class MyHandler implements InvocationHandler{
	private Object target;
	private StudentLogger logger = new StudentLogger();

	public MyHandler(Object target, StudentLogger logger) {
		this.target = target;
		this.logger = logger;
	}

	public MyHandler(Object target) {
		this.target = target;
	}

	//参数1 proxy  将来给目标对象所动态产生的代理对象
	//参数2 method 将来你所调用的目标对象中的方法的镜像
	//参数3 args        将来你所调用方法的时候所传的参数
	public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
		String msg = method.getName()+"方法被调用了...";
		logger.log(msg);
		Object o = method.invoke(target, args);
		return o;
	}
}

//测试类
public class DProxyTest {
	public static void main(String[] args) {
		IStudentService target = new StudentServiceImpl();
```

```
                ClassLoader loader = target.getClass().getClassLoader();
                Class<?>[] interfaces = target.getClass().getInterfaces();
                InvocationHandler h = new MyHandler(target);

                //参数1 loader                  目标对象的类加载器
                //参数2 interfaces      目标对象所实现的接口
                //参数3 h                       InvocationHandler接口的实现类对象
                IStudentService proxy =
(IStudentService)Proxy.newProxyInstance(loader, interfaces, h);

                proxy.delete(1);
                proxy.save(null);
                proxy.find(1);

                System.out.println(proxy.toString());
                System.out.println(proxy.getClass());
                System.out.println(target.getClass());
        }

    }
```

9、外观模式（Facade）

外观模式也可以叫做门面模式

为子系统或者模块中的一组接口提供一个一致的访问方式，此模式定义了一个高层接口，这个接口使得各个子系统/模块中的功能更加容易使用。

实际应用中，我们在对付一些老旧的代码或者即便不是老旧code，但涉及多个子系统时，除了重写全部代码，我们还可能采用这样一种策略：重新进行类的设计，将原来分散在源码中的类/结构及方法重新组合，形成新的、统一的接口，供上层应用使用,同时也隐藏了子系统或者子模块中功能实现的复杂性

例子:

```
//模块A中的类
public class ServiceA {
        public void start(){
                System.out.println("模块A中的start方法");
        }
}
//模块B中的类
public class ServiceB {
        public void run(){
                System.out.println("模块B中的run方法");
        }
}

//模块C中的类
public class ServiceC {
        public void end(){
```

```
                System.out.println("模块C中的end方法");
        }
}

//外观类/门面类
public class Facade {
        private ServiceA a;
        private ServiceB b;
        private ServiceC c;
        public Facade() {
                a = new ServiceA();
                b = new ServiceB();
                c = new ServiceC();
        }

        public void start(){
                a.start();
        }
        public void run(){
                b.run();
        }
        public void end(){
                c.end();
        }

        public void service(){
                a.start();
                b.run();
                c.end();
        }

}

//测试类
public class Test {

   public static void main(String[] args) {
        Facade f = new Facade();
        f.start();
        f.run();
        f.end();

        f.service();
   }
}
```

Facade是我们的外观类/门面类,用户可以通过这个类使用到系统中不同模块中的不同方法，同时也对用户隐藏了系统中对这些功能的实现细节。给用户提供了一个统一的访问方式。

10、桥接模式（Bridge）

桥接模式（也叫桥梁模式）就是将抽象部分和实现部分分离，使它们都可以独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。

例子：

```
//公共的驱动接口
public interface Driver {
    public void getConnection();
}
//第一个实现类 mysql驱动类
public class MysqlDriver implements Driver{
    public void getConnection() {
        System.out.println("mysql 数据库连接");
    }
}
//第二个实现类 oracle驱动类
public class OracleDriver implements Driver {
    public void getConnection() {
        System.out.println("oracle数据库连接");
    }
}
//抽象的管理器  Bridge
public abstract class Manager {
            private Driver driver;
            public  void getConnection(){
                driver.getConnection();
            }
            public void setDriver(Driver driver) {
                this.driver = driver;
            }
}
//具体的驱动管理器 Bridge
public class DriverManager extends Manager {
    public DriverManager(Driver driver){
        setDriver(driver);
    }
    public void getConnection() {
        super.getConnection();
    }
}
```

//测试类 注意我们的抽象和具体实现是分开的,无论他们如何变化都不会影响到我们bridge中的功能执行

//JDBC中,我们使用的就是一系列javaAPI提供的接口,而且数据公司商则给我们提供接口的实现

```
public class Test {
    public static void main(String[] args) {

        DriverManager manager = new DriverManager(new MysqlDriver());
        manager.getConnection();

        manager = new DriverManager(new OracleDriver());
        manager.getConnection();
    }
}
```

11、组合模式（Composite）

组合模式有时又叫部分-整体模式，在处理类似树形结构的问题时比较方便

例子:

```
//节点类
public class TreeNode {

    private String name;
    private TreeNode parent;
    private Vector<TreeNode> children = new Vector<TreeNode>();

    public TreeNode(String name){
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public TreeNode getParent() {
        return parent;
    }

    public void setParent(TreeNode parent) {
        this.parent = parent;
    }

    //添加孩子节点
    public void add(TreeNode node){
        children.add(node);
    }
```

```
    //删除孩子节点
    public void remove(TreeNode node){
        children.remove(node);
    }

    //取得孩子节点
    public Enumeration<TreeNode> getChildren(){
        return children.elements();
    }
}
//表示一个树状结构
public class Tree {

    TreeNode root = null;

    public Tree(String name) {
        root = new TreeNode(name);
    }
}

//测试类
public class Test{

        public static void main(String[] args) {
                Tree tree = new Tree("A");
                TreeNode nodeB = new TreeNode("B");
                TreeNode nodeC = new TreeNode("C");

                nodeB.add(nodeC);
                tree.root.add(nodeB);
                System.out.println("build the tree finished!");
        }
}
```

12、享元模式（Flyweight）

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

例子：

```
//数据库连接池
public class ConnectionPool {

    private Vector<Connection> pool;

    /*公有属性*/
    private String url = "jdbc:mysql://localhost:3306/test";
```

```
    private String username = "root";
    private String password = "root";
    private String driverClassName = "com.mysql.jdbc.Driver";

    private int poolSize = 100;
    //这里对instance可以使用一个单例模式
    private static ConnectionPool instance = null;
    Connection conn = null;

    /*构造方法，做一些初始化工作*/
    private ConnectionPool() {
        pool = new Vector<Connection>(poolSize);

        for (int i = 0; i < poolSize; i++) {
            try {
                Class.forName(driverClassName);
                conn = DriverManager.getConnection(url, username, password);
                pool.add(conn);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    /* 把连接对象返回到连接池 */
    public synchronized void release() {
        pool.add(conn);
    }

    /* 返回连接池中的一个数据库连接 */
    public synchronized Connection getConnection() {
        if (pool.size() > 0) {
            Connection conn = pool.get(0);
            pool.remove(conn);
            return conn;
        } else {
            return null;
        }
    }
}
```