Chp17 OOAD 初步

OOAD，指的是面向对象的分析与设计。作为本书的最后一章，本章将介绍很多基本的面向对象的分析和设计内容,从根本上阐述面向对象设计思想的精髓。

首先，我们将介绍 UML 图以及类关系。

1 UML 图以及类关系

UML 是统一建模语言的缩写，使用 UML，能够用图形的方式比较清晰的表达出软件设计师的设计意图，也能够让程序员之间更加容易的交流软件的结构和设计。

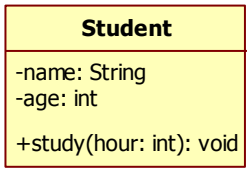
在 UML 中，共有 9 种基本的图形，包括：用例图，类图，对象图，时序图，协作图，组件图，部署图，活动图和状态转换图。其中，出现频率最高的当属类图。

类图，顾名思义，就是描述系统中类的组成，以及类与类之间的关系。

1.1 类图

1.1.1 UML 表示类

用类图表示一个类，则可以绘制如下：



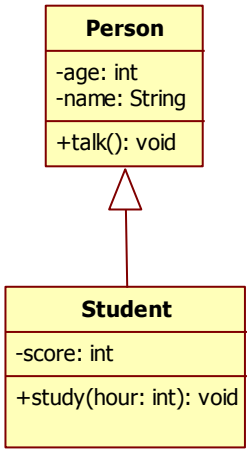
我们用一个矩形来表示一个类。把这个矩形用横线分为三个部分：第一部分用来写类名；第二部分用来写属性；第三部分用来写方法。

属性的写法：先写属性名，后写属性的类型。另外，对于访问权限修饰符，用“-”表示私有，用“+”表示公开。

方法的写法：先写方法名，然后写参数表，最后是返回值类型。

1.1.2 继承关系

两个有继承关系的类，用类图表示如下：

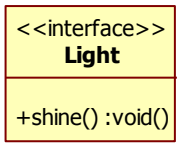


我们可以看到，子类 Student 继承 Person 类，则从子类 Student 出发，用一个空心的三角箭头指向父类。这就是用类图表示的继承关系。

1.1.3 接口

接口有两种画法。

接口的第一种画法是完整画法，如下：

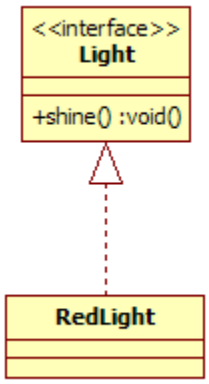


这种画法与类相似，也是用一个矩形来表示。所不同的是，在接口的名字上方，会有

< <interface> >

的字样，强调这是一个接口。

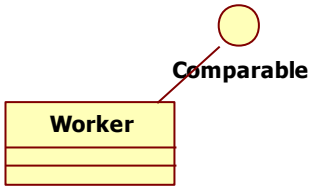
如果用一个类来实现这个接口，则 UML 图描述如下：



用一个空心的三角箭头指向被实现的接口，接口和实现类之间用虚线连接。

接口的第二种画法是简化的画法，如下图：

这种画法只需要画一个空心的圆形，然后写上接口的名字即可。对应于这种简化画法，一个类实现一个接口，也可以画成下面的图形：



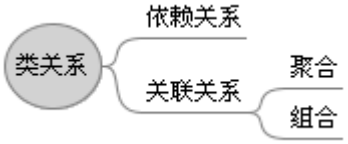
用一根实线连接接口和实现类，表示接口的实现。

1.2 类关系

接下来，我们要为大家介绍的是类与类之间的关系。类与类之间除了我们所熟知的继承关系之外，还可能存在以下的关系。

1.2.1 依赖、聚合与组合

类的关系，可以用下面的图来表示：



我们对这些关系依次来进行解释。

1.2.1.1 依赖关系

所谓依赖关系，又被称为 use-a 关系。这种关系的特点是：A 类和 B 类之间如存在依赖关系，则 A 类中有一个 B 类型的局部变量（当然，方法参数也可以认为是特殊的局部变量）。

怎么来理解这个关系呢？例如，有一个人要过河，也就说，有一个 Person 对象要调用 crossRiver 方法。而这个人要过河，需要一条小船载他过去，也就是说，他需要一个 Boat 对象。我们可以这样来理解：

```
class Person{
    public void crossRiver(Boat b){
        ...
    }
}
```

上面的代码，Boat 类是 Person 类的某一个函数的方法参数。表示如果一个人要过河的话，则他需要一条船。我们可以理解为，调用 Person 对象的 crossRiver 方法的话，需要用一下某一个船对象。而当这个方法调用完毕之后，作为局部变量的 Boat 引用即失效，Person 对象和 Boat 对象之间就毫无瓜葛，没有联系了。用英语表述的话，就是：

Person use a Boat to crossRiver.

你看，这就说明 Person 类和 Boat 类是“use-a”关系，表明他们之间是依赖关系。

用类图来表示，如下：

Person 类依赖 Boat 类，则可以用一根虚线连接 Person 和 Boat 两个类，并在被依赖的一方画一个箭头。

1.2.1.2 关联关系

所谓 A 类和 B 类存在关联关系，指的是 A 类中有一个 B 类型的属性。由于属性表示对象“有”什么，所以关联关系也被称之为“has-a”关系。

当然，“有”也有所不同。例如，人有一双手，以及人有一辆自行车。虽然都是“有”，但是这两者还是有区别的。

人“有”自行车，在这种关系中，人和自行车是相对比较独立的对象。这个独立，指的是外部对象“人”和内部对象“自行车”的生命周期之间，没有必然的联系。有可能人还在，自行车没了；也有可能自行车还在，人没了。这种外部对象和内部对象在失掉关联关系之后，依然可以分别存在的情况，称之为“聚合”。用 UML 图表示如下：

在外部对象的一端，用一个空心的菱形表示聚合关系。

人“有”手，在这种关系中，人和手是密切不可分割的对象。外部对象一旦不存在，则内部对象也一定不存在了，换句话说，外部对象管理内部对象的生命周期。这种对象的关系称之为“组合”，用 UML 图表示如下：

在外部对象的一端，用一个实心的菱形表示组合关系。

1.2.2 关联关系

下面我们对关联关系作进一步的阐述。关联关系是有方向性和多重性的。

1.2.2.1 关联关系的方向性

我们来看如下代码：

```
class Person{
    Bike bike;
}
class Bike{}
```

很明显，在 Person 类中存在一个 Bike 类型的属性，因此我们可以认定，Person 类和 Bike 类之间存在关联关系，但是反过来，在 Bike 类中并不存在一个 Person 类型的属性，也就是说 Bike 类和 Person 类并不存在关联关系。也就是说，关联关系有方向性，A 类关联 B 类并不意味着 B 类也关联 A 类。

对于上述代码给出的关系，Person 类关联 Bike 类，但 Bike 类并不关联 Person 类，我们称之为“单向关联”。但是对于以下代码：

```
class Man{
    Woman wife;
}
class Woman{
    Man husband;
}
```

我们则可以称之为“双向关联”。因为在 Man 类关联 Woman 类的同时，Woman 类也关联了 Man 类。

对于单向关联，类图表示如下：

箭头由 Person 类指向 Bike 类，表示只有 Person 类关联 Bike 类，Bike 类没有关联 Person 类。

而对于双向关联，类图表示如下：

在 Man 类和 Womna 类之间只有一道实线，没有箭头，表示实线两端的类之间双向均存在关联关系。

1.2.2.2 关联关系的多重性

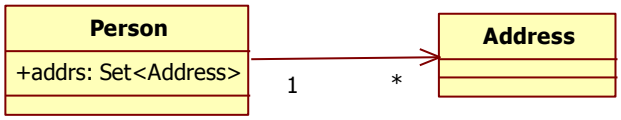
最简单的关联关系是一对一关联，也就是说，关联关系的两端往往是一个对象对应着一个对象。例如前面的例子：人和自行车（往往一个人只有一辆自行车）；丈夫和妻子(按照一夫一妻制，一个丈夫对象所关联的妻子对象也只能是一个)。这些都是一对一关联。

除了一对一关联之外，还有可能有一对多关联。例如，一个人（Person）可能有多个地址（Address）：公司地址、家庭地址、户口所在地等等。在设计代码时，Person 类中就必须采用数组或是集合来保存其所关联的所有 Address 对象。

一对多关联也有方向。例如，Person 类与 Adress 类，就是单向一对多关联。我们可以从一个 Person 对象中获得它关联的所有地址对象。示例代码如下：

```
class Address{ ... }
class Person{
    Set<Address> addrs;
}
```

一对多单向关联的 UML 图如下：



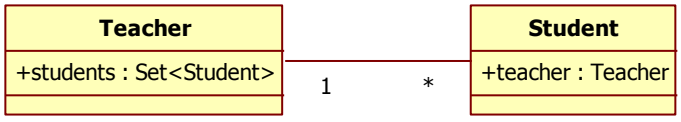
在单向关联的基础上，在“一”的一段用一个 1 来表示，而在“多”的一段用一个*来表示。

当然，也有一些其他的表示方式，例如，在“一”的一段，还可以使用“0..1”来表示，在“多”的一端，可以用“n”、“0..*”、“1..*”、“0..n”、“1..n”等方法来表示。

类似的，双向一对多关联也可以这么表示。例如老师和学生之间的关联：一个老师可以管理多个学生，而一个学生被一个老师管理。示例代码如下：

```
class Teacher{
    Set<Student> students;
}
class Student{
    Teacher teacher;
}
```

用 UML 图表示双向一对多关联，如下：



多对多关联，表示两个类中，都有一个属性，是另一个类的一个集合。因此，多对多关联没有单向，只有双向。例如，一个学生可以选多门课程，而每门课程都有多个学生选。学生和课程之间就形成了多对多的关联。示例代码如下：

```
class Course{
    Set<Student> students;
}
class Student{
    Set<Course> courses;
}
```

表示多对多关系的 UML 图如下：

2 常用设计模式

我们在介绍 I/O 框架的时候，曾经介绍过设计模式的概念。设计模式是值在设计面向对

象软件的过程中，用固定的套路去解决一些通用的问题。

在这一部分的内容中，我们将介绍单例模式、简单工厂模式。

2.1 单例模式

单例模式要解决的是这样一个问题：某一个类在整个程序中，只有唯一的一个对象。例如，在封建社会，“皇帝”这个类永远只能有一个唯一的对象。在实际代码中，我们也可能认为某个类在整个程序中只需要创建一个对象：例如唯一的一个数据库连接池对象，唯一的一个网络连接对象等等，而不希望程序在运行过程中创建很多新的连接而浪费资源影响性能。

那么我们怎么能做到这一点呢。例如下面的代码：

```
public class Singleton{
    public Singleton(){}
}
```

由于 Singleton 类提供了一个公开的构造方法，我们当然可以创建出任意多个 Singleton 类的对象，因此，我们要把构造方法改为私有。这样，这个方法只能在 Singleton 类内部调用。由于创建对象必须调用构造方法，这也意味着，不能在 Singleton 类外部创建 Singleton 对象。

代码形式如下：

```
public class Singleton{
    private Singleton(){}
}
```

同时我们可以在 Singleton 类中增加一个静态方法，由这个静态方法负责返回一个 Singleton 对象。这个静态方法由于在 Singleton 内部，因此可以调用构造方法；并且，这个静态方法可以通过 Singleton 的类名，在外部直接调用。如下：

```
public class Singleton{
    private Singleton(){}
    public static Singleton getInstance(){
        return new Singleton();
    }
}
```

经过上面两步修改，我们可以通过 Singleton.getInstance()获取对象了，但是还是没法做到全局唯一，因为每次对该方法的调用都会新创建一个对象。为此，我们为 Singleton 类增加一个静态的 instance 属性，并且修改 getInstance 方法如下：

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

上面的代码中，当第一次调用 getInstance 方法时，由于静态属性 instance 为 null，会调用构造方法创建一个对象，并把这个对象返回。当后面再调用 getInstance 方法时，则会直

接返回 instance 对象。这样，在整个程序中，每次调用 getInstance 方法都能保证返回的是同一个对象，这就是基本的单例模式。

但是上面的单例模式的写法还有不完善的地方。注意 getInstance 方法：

```
01:public static Singleton getInstance(){
02:    if (instance == null)
03:        instance = new Singleton();
04:    return instance;
05:}
```

考虑多线程的情况。假设在一开始，就有两个线程 t1 和 t2 同时调用 getInstance 方法。当 t1 执行到 02 行的时候，进行判断，假设现在 instance 依然为 null，因此判断为 true，t1 线程创建对象。在 t1 线程创建对象的过程中，CPU 时间片到期，t1 进入了可运行状态，t2 进入运行状态。

此时，由于依然没有创建对象，因此 instance 依然为 null，所以 t2 线程进行判断之后，创建一个 Singleton 对象，并返回。

这样，t1 和 t2 线程总共创建了两个 Singleton 对象，破坏了单例模式的含义！

我们上面描述的多线程问题，出现的概率非常低，但是，如果在一个高并发的服务器上，一旦出现这种问题，可能就会造成非常大的影响。因此，我们必须要解决这个问题。

怎么解决这个问题呢？非常简单，我们只要做一个简单的修改即可：

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

在初始化 instance 属性的时候直接创建对象，这样，创建对象的过程在类加载的时候完成。这就解决了多线程的问题。

上面就是单例模式的介绍。在写单例模式的时候，有三个要点：1、私有的构造方法；2、静态的 instance 属性；3、静态的 getInstance()方法。掌握这三个要点，就能顺畅的写出单例模式的代码。

2.2 简单工厂模式

下面要介绍的是简单工厂模式。严格的说，简单工厂模式并不算是一种设计模式，因为它解决的问题非常基本，解决的方式也非常简单。事实上，我们可以把简单工厂模式当做常见的编程的写法。

这种模式主要解决的是创建对象的问题。我们之前创建对象，都使用 new 关键字。这种创建对象的方式当然可以，但假如是复杂的对象，可能就不那么简单。

例如，举一个生活中的例子来说。吃早餐，如果早餐我们打算吃一个煮鸡蛋，那可能早起 5 分钟，自己就可以创建一个煮鸡蛋对象出来。但如果早餐打算吃煎饼，如果自己创建的话，则需要和面、炸薄脆或者油条、准备面酱、切葱花香菜，最后，摊煎饼。很显然，这么多步骤都让自己来完成，是不现实的。为了解决这个问题，我们可以去煎饼摊买一个煎饼。在这个过程中，我们可以把煎饼摊当做是一个煎饼工厂，这个工厂有一个 createJianBing 方

法，能够被调用并返回一个煎饼对象。

示例代码如下：

```
public class JianBingFactory{
    //返回一个煎饼对象
    public static JianBing createJianBing(){
        return new JianBing();
    }
}
class JianBing{}
```

由工厂对象负责对象的创建，可以把创建一个复杂对象的代码从其他代码中分离出来，使得代码的功能更加单一，符合面向对象“各司其职”的原则。

3 三层体系结构介绍

接下来我们将介绍的是软件设计中非常重要也非常常用的内容：软件的三层体系结构。

3.1 需求的变化

首先，我们从代码开始。假设我们是在公司中工作的程序员，每天的工作就是完成老板提出的需求，写出相应的代码。好了，下面，我们开始一天的工作！老板的需求正源源不断的到来！

需求 1 ： 写一个程序，在屏幕上打印出 Hello World

这个代码非常简单，如下：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

需求 2：从文件中读取一行文本，在屏幕上打印

我们将前面的代码修改如下：

```
import java.io.*;
public class HelloWorld {
    public static void main(String[] args) {
        BufferedReader br = null;
        try{
            FileReader fr = new FileReader("hello.txt");
            br = new BufferedReader(fr);
            String line = br.readLine();
            System.out.println(line);
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (br != null){
                try{
```

```
                br.close();
            }catch(IOException e){}
        }
    }
}
```

我们可以看到，代码已经变得复杂了很多。

需求 3： 把读到的文本转成大写，然后输出。

```
import java.io.*;
public class HelloWorld {
    public static void main(String[] args) {
        BufferedReader br = null;
        try{
            FileReader fr = new FileReader("hello.txt");
            br = new BufferedReader(fr);
            String line = br.readLine();
            line = line.toUpperCase();
            System.out.println(line);
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (br != null){
                try{
                    br.close();
                }catch(IOException e){}
            }
        }
    }
}
```

需求 4： 不从文件中读取，改为从网络中读取。代码如下：

```
import java.io.*;
import java.net.*;
public class HelloWorld {
    public static void main(String[] args) {
        Socket s = null;
        try{
            s = new Socket("127.0.0.1", 9000);
            BufferedReader br = new BufferedReader(
                    new InputStreamReader(s.getInputStream()));
            String line = br.readLine();
            line = line.toUpperCase();
            System.out.println(line);
```

```
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (s != null){
                try{
                    s.close();
                }catch(IOException e){}
            }
        }
    }
}
```

注意，在进行网络编程的时候，需要修改的代码已经相当多了。

需求 5：把读到的字符串全都转为倒置输出。

```
import java.io.*;
import java.net.*;
public class HelloWorld {
    public static void main(String[] args) {
        Socket s = null;
        try{
            s = new Socket("127.0.0.1", 9000);
            BufferedReader br = new BufferedReader(
                    new InputStreamReader(s.getInputStream()));
            String line = br.readLine();
            StringBuffer sb = new StringBuffer(line);
            sb = sb.reverse();
            line = sb.toString();
            System.out.println(line);
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (s != null){
                try{
                    s.close();
                }catch(IOException e){}
            }
        }
    }
}
```

需求 6： 把从网络上读取再改成从文件中读取。

STOP！你会发现，你的代码越来越混乱，随着需求的改变和不断的调整，我们的代码也不断进行调整。在刚刚修改代码的过程中，你有没有这样的想法：为啥要在原来的代码上

修改呢？还不如推倒重新写一遍呢……

恭喜你，在整个软件行业中，并不是只有你有这样的想法，很多大项目，在维护到后面的时候，很多程序员都在抱怨，觉得前人留下的代码非常的混乱而难于理解，不希望继续在前人的基础上工作，从头开始做项目可能更加容易一些。现在我们应该能够理解他们了，面对杂乱无章的代码，确实令人抓狂。而代码为什么凌乱呢？并不是一开始就乱的，而是在一遍又一遍修改的过程中，逐渐被“改乱”的。

当一个程序无法应对新的需求变化的时候，当改动原有程序比重新写一个程序需要花更多时间的时候，这时，我们当然会选择重新写程序，那么原有的程序就走向了死亡。在软件行业中，有大量的软件，花了长的时间进行开发之后，没有多久就因为种种的原因而必须被淘汰。这造成了软件的短命，也造成了软件的成本居高不下，无法充分利用已有的资源。

我们首先来分析一下，造成软件短命的原因是什么。有人会说，需求的变化是造成软件短命的主要原因。如果一个软件的需求没有变化或者变化很少，则需要对这个软件的改动也非常少，这样这个软件就能够一直使用下去，而不用进行大的修正。

但是，一方面，由于需求是由客户提出的，对于程序员来说，无法控制需求的变化。另一方面，随着技术的进步和时代的发展，技术和商业一定会有着非常巨大的变化。这些变化，都会使得人们对软件的要求有变化。例如，最早，用电话线上网的时候，腾讯公司的 qq 软件只有文字聊天功能；之后，随着带宽的增加和需求的变化，qq 软件增加了传送文件等功能；再之后，随着 ADSL 的兴起，如今的 qq 软件有了语音以及视频聊天、qq 游戏、qq 空间等等非常丰富的功能。这就是一个典型的需求不断变化的例子。

因此我们发现，需求的变化是必然的，程序员只能去适应需求的变化。于是，人们延长软件寿命的工作重点放在了“避免修改代码”上。是啊，如果能够在不修改代码的情况下，满足新的需求，那么就能在最大程度上避免软件被“越改越乱”了。于是，软件行业提出了一个设计原则：“开闭原则”。

开闭原则，指的是：在软件设计的过程中，要求软件能够做到：对扩展开放，对修改关闭。程序员可以通过在原有代码基础上添加新代码的方式，来满足新的需求，而不是修改原有的代码。如果一个软件能够做到这一点，那么软件的功能可以自由扩展，从而应对需求的变化；而原有的部分保持成熟和稳定。这样就能更好的保持程序结构的清晰易读。

为了实现开闭原则，有一些更加具体的要求。例如，修改关闭，就意味着原有的代码，在新的扩展以后的系统中继续能够使用，也就是代码的“可重用性”；而扩展开放，就意味着新的代码能够很方便的扩展原有的系统，而不影响原有的代码，这也就是代码的“可扩展性”；为了能够达到开闭原则，就要求模块之间的联系应当尽可能的弱，这样才能够保证方便的扩展新功能而不影响其他功能。同时，我们应该能够根据不同的新需求，扩展相应的软件模块，这也就有了软件“各司其职”的要求，即：软件的不同模块在功能上应该有明确的职责划分。也就是说，为了实现开闭原则，我们的软件应该具备以下特点：

- ✓ 可重用性
- ✓ 可扩展性
- ✓ 弱耦合性
- ✓ 各司其职

细心的读者可以看出，这些正是面向对象编程思想的特点和要求。由此可见，面向对象

的思想不是凭空产生的，而是软件行业为了应对需求的变化，为了能够更好的实现开闭原则，在编程思想领域的重大进步。

举个例子，我们都知道中国古代的四大发明，分别是造纸术，指南针，火药和活字印刷术。这其中，造纸术、指南针和火药都是从无到有的发明，唯有活字印刷术比较特别，北宋时期的毕昇只是将原有的印刷术加以改进，发明了活字印刷术。这难道不奇怪吗？我们往往认为，技术的发明者要比技术的改良者更值得纪念。就像我们记住了灯泡的发明者是爱迪生，却淡忘了节能环保型灯泡的发明者。可是针对印刷术，谁又能说清印刷术的发明者是谁呢？我们记住的只是那个改良者—毕昇。

这不难理解，传统的印刷术，印刷工人要在一整块木板上刻下所有的文字。一个错字就可能使得整个版作废。而活字印刷术高明之处在于，将每个字做成独立的“个体”，由多个“个体”组成词语，句子。这样，当文字发生改变的时候，只需要替换或增加有改动的文字即可，使得印刷工作符合了“开闭原则”。具体的说，每个字是独立的个体，这符合“各司其职”的要求；做好的字可以反复使用，这符合“可重用性”的要求；字与字之间彼此独立，互不影响，这符合“弱耦合性”的要求；整个版面可以在不影响其他字的情况下，随意添加新的文字，这又符合了“可扩展性”的要求。总之我们可以戏称，活字印刷术位列四大发明，体现了开闭原则的价值，闪烁着面向对象的光芒。

而为了更好的使用面向对象思想，为了使得我们的程序更加符合开闭原则，下面我们将为大家介绍非常典型的软件职责划分的方法：软件的三层体系结构。

3.2 三层体系结构介绍

在介绍三层体系结构之前，我们先分析一下之前提出的那些需求。我们可以把所有的需求分成三大类。

第一类：数据从哪儿来。之前提出的需求中，有的需求数据是从文件中读取，而有些情况，数据是从网络中读取。这一类需求的变化，是数据来源的变化，也可以认为是访问数据的方式的变化。

第二类，数据怎么处理。之前提出的需求中，数据获得之后，有些需求要求把数据全部转为了大写，有些需求要求把数据都转为倒置。这一类需求的变化，是对数据处理的变化，也可以认为是处理数据方式的变化。

第三类，数据怎么显示。在我们这些需求中，数据的显示比较简单，通过输出语句直接输出数据。但是我们可以想象，在以后的编程实践中，数据的现实会有各种各样的方式，例如通过图形界面显示，通过网页显示等等。

本着各司其职的思想，我们把软件设计成三个层次。这三个层次分别对应于三类需求。

首先是数据访问层。数据访问层是用来和数据打交道，具体的说，负责数据的增加，删除，修改和查询（当然，在我们的例子中，只涉及数据的查询）。而数据访问层的对象，被称之为数据访问对象（Data Access Object），简称 DAO。因此，数据访问层也被称为 DAO 层。DAO 层对应着第一类需求：数据从哪儿来。

其次是业务逻辑层。业务逻辑层，是专门用来处理数据的，这一层的对象被称之为业务对象（Business Object），简称 BO。而数据访问层也被称为 biz 层。

需要注意的是，业务逻辑层处理的数据，往往是从 DAO 层来的。也可以认为，DAO 负责获取数据，然后把数据传递给 biz 层，让 biz 层对数据进行处理。

用户提交的请求数据需要被接收，当数据处理完之后，结果数据需要显示给用户。而负责接收用户请求，并显示数据的是显示层，也被称为 view 层。而 view 层中的负责与用户交

互的对象称之为 View Object，简称 VO。

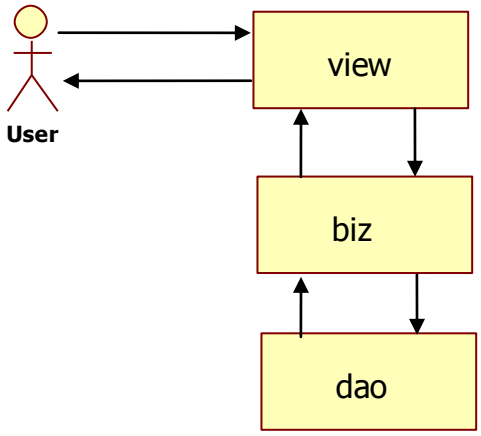
把软件分成三个层次之后，典型的情况如下：

view 层与用户交互的过程中，接受用户的一个指令。如果 view 层用图形界面显示数据，则这个指令有可能是图形界面上的一次点击；如果 view 层用网页显示数据，则这个指令有可能是网页上发送的一个 http 请求。

当 view 层获得一个用户的指令之后，会把这个指令交给 biz 层。在 view 层中，VO 会调用 biz 层中的方法，把用户跟 view 层交互时输入的一些数据，传递给 biz 层。然后，等 biz 层把数据处理完成之后，再把处理完成的数据返回给 view 层，让 view 层显示结果。

而 biz 层如果要处理数据，可能要先获得数据。为了获得数据，biz 层需要调用 dao 层的方法。当 dao 层把数据获取之后，dao 对象会将结果返回给 biz 层，biz 层才能根据数据进行下一步的处理。

因此，view、biz、dao 三个层次之间，是从上到下依次调用方法的关系。示意图如下：



我们以一个现实生活中的例子，来说明三层结构的概念。例如，一家汽车 4S 店，这种店为客户提供多种服务，例如购车、保养、修车等等。比如，客户的车坏了，需要修车。这个时候，当客户来到 4S 店时，与客户交互的往往是前台的接待员。这些接待员能够跟用户清晰、友好的沟通，获得客户的指令。我们可以把这些接待员当做就是 view 层中的 VO。

当客户告诉接待员，“我需要修车”，这就相当于客户发送了一个指令给了 VO。接待员知道用户需要修车之后，自己不会替用户完成修车这个过程。修车的指令，被接待员发送给了真正的汽车修理工。汽车哪部分有毛病，哪部分需要检修，这些数据都由接待员告诉修理工，由修理工真正来完成“修车”这个业务。在这个关系中，修理工就相当于 biz 层中的 BO，而上述流程就相当于 view 层的 VO 调用 biz 层的 BO 的方法。

当修理工修车时，有可能需要进行零部件的更换。例如，可能需要更换发动机，为此，修理工必须要获得一个新的发动机。往往备用的零配件是存放在仓库中，而仓库显然不能让每个人都随意进行访问，往往公司会安排一个专人做仓库管理员。仓库管理员的职责，就是负责向仓库存放物资，以及从仓库中取出物资。如果我们把零配件当做数据，那么仓库管理员的工作就是存取数据，扮演的就是 DAO 的角色。而修理工修车时，会根据需要向管理员要零配件，可以认为这就是 biz 层的 BO 对象在调用 dao 层的 DAO 对象的方法。

当仓库管理员找到相应的零配件时，会把这些物品交给修理工；而当修理工修车完毕之后，会通知前台的接待员；接待员最后，会把修好的车以及其他的一些信息（例如修车花了多少钱……）显示给客户。类比 Java 代码就是：dao 层返回到 biz 层，biz 层返回给 view 层，

view 层把运算的结果显示给客户。

上面我们介绍了软件的三层体系结构。那么，把软件设计成三层结构有什么好处呢？好处在于：当需求发生改变时，我们可以把改变局限在某个层次中，而不影响其他层次。例如，如果仓库的地点以及放置物品的位置发生改变的话，我们不需要对前台接待员和汽车修理工做过多的说明，只要让仓库保管员能够清楚应该怎么工作就可以了。同样的，如果某一个层次的需求发生变化，则我们只需要针对那个特定的层次，修改相应的代码，而不用改变其他层次的代码。

为了让层次与层次之间，实现弱耦合性，我们使用接口来定义三个不同的层次。

3.3 三层结构的 HelloWorld 程序

下面我们以 Hello World 程序为例，来看一下应当如何应用三层体系结构。

首先，应当定义三层的接口。先是 dao 层的接口。

```
package dao;

public interface Dao {
    String getData();
}
```

dao 层中所有的 DAO 对象都应该实现 Dao 接口。

然后，是 biz 层的接口。需要注意的是，因为 biz 层需要调用 dao 层的方法，因此，在 biz 对象中，需要维护一个 Dao 对象的引用。为此，所有 Biz 层接口的实现类都应当有一个 Dao 类型的属性，并且提供一个 setDao 方法。因此，在 Biz 接口中，我们定义了 setDao 方法。

```
package biz;
import dao.Dao;

public interface Biz {
    void setDao(Dao dao);
    String dealData();
}
```

biz 层中所有的 BO 对象都应该实现 Biz 接口。

最后，是 view 层的接口。与之前的情况类似，view 层也应当有一个 Biz 类型的引用，用以调用 biz 层的方法。因此，在 View 接口中，我们定义了 setBiz 方法。

```
package view;
import biz.Biz;

public interface View {
    void setBiz(Biz biz);
    void showData();
}
```

view 层中所有的 VO 对象都应该实现 View 接口。

定义完了三个接口之后，接下来应该给出的是接口的实现类。例如，我们首先给出 Dao

接口的实现类。假设我们希望从当前目录下的“test.txt”文件中获取数据，则可以给出一个实现类：FileDaoImpl 实现 Dao 接口，代码如下：

```
package dao;
import java.io.*;
public class FileDaoImpl implements Dao {

    public String getData() {
        String data = null;
        BufferedReader br = null;
        try{
            br = new BufferedReader(new FileReader("test.txt"));
            data = br.readLine();
        }catch(IOException e){
            e.printStackTrace();
        }
        finally{
            if (br != null){
                try{
                    br.close();
                }catch(IOException e){e.printStackTrace();}
            }
        }
        return data;
    }

}
```

下面是 Biz 接口的实现。假设我们要实现把所有数据都转成大写的逻辑，则可以给出一个 UpperCaseBizImpl 的实现类。代码如下：

```
package biz;
import dao.Dao;

public class UpperCaseBizImpl implements Biz {
    private Dao dao;

    public String dealData() {
        String data = dao.getData();
        if (data != null){
            data = data.toUpperCase();
        }
        return data;
    }

    public void setDao(Dao dao) {
```

```
        this.dao = dao;
    }

}
```

需要注意的是，我们为 Biz 的实现类增加了 Dao 类型的属性，原因是在 dealData 方法中，我们使用了 Dao 接口中定义的方法。

最后，View 接口的实现比较简单。我们给出 TextViewImpl 的实现代码：

```
package view;
import biz.Biz;

public class TextViewImpl implements View {
    private Biz biz;
    public void setBiz(Biz biz) {
        this.biz = biz;
    }

    public void showData() {
        String data = biz.dealData();
        System.out.println(data);
    }

}
```

当把三个接口的实现类完成之后，接下来，就可以写主方法，设置类之间的关联关系。代码如下：

```
package test;

import dao.*;
import biz.*;
import view.*;

public class TestMain {
    public static void main(String[] args) {
        //创建对象并调用 set 方法进行组装
        Dao dao = new FileDaoImpl();
        Biz biz = new UpperCaseBizImpl();
        biz.setDao(dao);
        View view = new TextViewImpl();
        view.setBiz(biz);

        view.showData();
    }
}
```

完成三层结构之后，再有新的需求到来时，我们就可以更好的应对。例如，现在有新的需求，需要把所有的字符串转为小写。

由于这个需求是“数据如何处理”，属于 biz 层的需求，我们无需改动原有的 biz 层对象，而可以为 Biz 接口扩展出一个新的实现类 LowerCaseBizImpl，代码如下：

```
package biz;
import dao.Dao;
public class LowerCaseBizImpl implements Biz {
    private Dao dao;

    public String dealData() {
        String data = dao.getData();
        if (data != null){
            data = data.toLowerCase();
        }
        return data;
    }

    public void setDao(Dao dao) {
        this.dao = dao;
    }

}
```

我们只是实现 Biz 接口，很轻松的就完成了对代码的扩展，而没有对原有代码进行任何的改动。这就达到了开闭原则中，“扩展开放”的要求。

之后，我们需要修改的代码只有这样一条：

```
    public static void main(String[] args) {
        //创建对象并调用 set 方法进行组装
        Dao dao = new FileDaoImpl();
        Biz biz = new LowerCaseBizImpl();
        biz.setDao(dao);
        View view = new TextViewImpl();
        view.setBiz(biz);

        view.showData();
    }
```

这样，与“修改关闭”的要求，也已经非常接近了。

3.4 简单工厂模式的应用

然而，接近了“修改关闭”，但是依然会在需求变化的时候，修改原有的代码。能不能完全不修改代码呢？

首先，修改代码的原因，与创建对象相关。为此，我们可以利用简单工厂模式，把所有

床架对象的过程，都挪到一个简单工厂中。工厂代码如下：

```
package factory;
import dao.*;
import biz.*;
import view.*;
public class SimpleFactory {
    public SimpleFactory(){}

    public Dao createDao(){
        return new FileDaoImpl();
    }

    public Biz createBiz(){
        return new UpperCaseBizImpl();
    }

    public View createView(){
        return new TextViewImpl();
    }
}
```

这样，TestMain 程序就可以改成：

```
package test;

import dao.*;
import biz.*;
import view.*;
import factory.SimpleFactory;

public class TestMain {
    public static void main(String[] args) {
        SimpleFactory factory = new SimpleFactory();
        Dao dao = factory.createDao();
        Biz biz = factory.createBiz();
        biz.setDao(dao);
        View view = factory.createView();
        view.setBiz(biz);

        view.showData();
    }

}
```

这样，当实现类改变的时候，主方法不需要改变，因为创建对象的代码都使用 factory

来完成了。

但是，这样如果需求改变的话，还是要该 factory 的代码。有没有办法不改代码，就能够在某个层次用一个实现类替换另一个实现类呢？

我们知道，利用反射可以灵活的创建对象。使用 Class.forName(String className)，通过一个字符串获得类对象。然后，通过类对象，可以创建一个相应类型的对象。简单的说，可以通过给定一个字符串，就获得一个该类型的对象。

而字符串，既可以写在代码中，同样可以从别的途径获得。例如，可以从一个配置文件中获得。因此，修改配置文件，就可以让 Class.forName 获得不同的字符串，从而创建出不同类型的对象来。换句话说，我们可以通过配置文件+反射的方式，来完成对象的创建。这样，当我们需要修改实现类的时候，只需要修改配置文件，而完全不需要修改代码。

首先，我们写一个方法，这个方法通过字符串来创建一个对象。private Object

```
private Object createObject(String name){
        Object result = null;
        try {
            Class c = Class.forName(name);
            result = c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
```

然后，我们在 SimpleFactory 的构造方法中，读某一个配置文件，然后把配置文件的信息保存起来。

我们创建一个配置文件 conf.props，并把配置文件设计成下面这种格式：

```
view=view.TextViewImpl
biz=biz.UpperCaseBizImpl
dao=dao.FileDaoImpl
```

用“=”分开两个部分，左边的部分是 view、biz、dao，分别表示三层；而右面表示的是该层我们采用的实现类的名字。注意，给出名字的时候，给出的是“包名 + 类名”的全限定名。我们可以看到，这样的配置文件非常类似于“键值对”的结构，等号左边为键，等号右边为值。并且，键和值都是字符串。

在 Java 中，解析这种“键值对”形式的配置文件，有一个非常方便的类：java.util.Properties。这个类是 Hashtable 的子类，他是一个特殊的 Map。特殊的地方有两种：

1） 这个类的键和值都是字符串。因此，这个类提供了一个方法：getProperty。这个方法接受一个字符串类型的参数，表示“键”；返回值也是字符串，对应的是值。

2） 我们可以通过这个类的 load 方法，来读入配置文件。load 方法可以接受一个 InputStream 参数，如果要读文件的话，那创建一个 FileInputStream 作为 load 方法参数即可。load 方法会自动解析输入流，例如我们上面的 conf.props，就会被自动解析出三个键值对放入 Properies 中，键分别为“view”、“biz”、“dao”，对应的值为“view.TextViewImpl”、“biz.UpperCaseBizImpl”、“dao.FileDaoImpl”。

为此，我们可以为 SimpleFactory 增加一个 Properties 的属性，并且在构造方法中，利用 load 方法，读入 conf.props。完整的 SimpleFactory 代码如下：

```
package factory;
import dao.*;
import biz.*;
import view.*;
import java.util.Properties;
import java.io.*;
public class SimpleFactory {
    private Properties props;
    public SimpleFactory(){
        props = new Properties();
        InputStream is = null;
        try{
            is = new FileInputStream("conf.props");
            props.load(is);
        }catch(IOException e){
            e.printStackTrace();
        }
        finally{
            if (is!=null){
                try{
                    is.close();
                }
                catch(Exception e){
                    e.printStackTrace();
                }
            }
        }

    }

    public Dao createDao(){
        String className = props.getProperty("dao");
        Dao dao = (Dao) createObject(className);
        return dao;
    }

    public Biz createBiz(){
        String className = props.getProperty("biz");
        Biz biz = (Biz) createObject(className);
        return biz;
    }
```

```
    public View createView(){
        String className = props.getProperty("view");
        View view = (View) createObject(className);
        return view;
    }

    private Object createObject(String name){
        Object result = null;
        try {
            Class c = Class.forName(name);
            result = c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

配置 conf,props 文件如下：

```
view=view.TextViewImpl
biz=biz.UpperCaseBizImpl
dao=dao.FileDaoImpl
```

test.txt 文件内容如下：

```
Hello World
```

当上述简单工厂完成之后，运行结果如下：

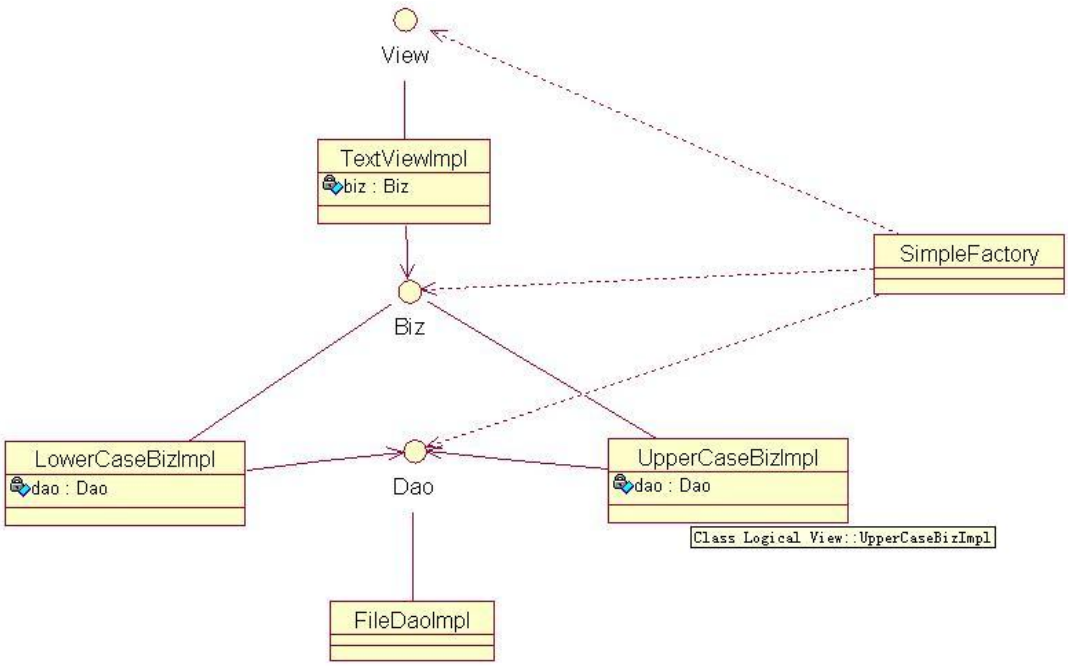
```
    HELLO WORLD
```

而当有新需求，希望把所有字符改成小写，只需要修改配置文件：

biz=biz.LowerCaseBizImpl

即可。

整个应用的类图如下：



读者也可以在该代码上，尝试着完成其他的一些需求。例如：文字从网络中获取（改变的是数据的获取方式，需要为 Dao 接口添加一个实现类，替换掉 FileDaoImpl）；将文字倒置处理（改变的是数据的处理方式，需要为 Biz 接口添加一个实现类，替换 LowerCaseBizImpl）。

也许读者会迷惑，写了这么多的接口和类，最终只是完成了最初级的 HelloWorld 输出的功能，是不是有点小题大做了？诚然，这个程序的功能并不复杂，但这种程序结构，使得在需求变化的时候，我们总是能够有针对性的扩展出某个接口的实现类，而不需要改动任何原有代码，从而保证了开闭原则的要求，也使得我们的代码能够“健康长寿”。我们可以认为，这才是一个真正贯彻了面向对象思想，发挥了面向对象优势的 HelloWorld！