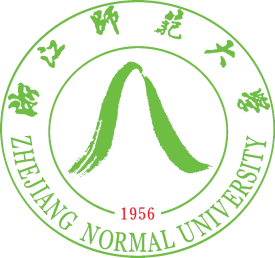
**浙江师范大学**

**数学与计算机科学学院**

**软件工程专业**

**（2017级）**

****

**《软件体系结构与设计模式》**

**课程报告**

|  |  |
| --- | --- |
| **主题：** | 设计模式 |
| **学号：** | 201732110127 |
| **姓名：** | 周旋 |
| **完成时间：** | 2019-11-25 |
| **分数：** |  |

**一、工厂模式**

工厂模式是现阶段在进行JavaWeb开发中我用地最舒服的一个模式。第一次接触到工厂模式的时候，我是在看网上的培训班视频(网上有非常多的培训班的免费视频，我看着视频写了我第一个spring框架项目)。

工厂模式的来源：为了实现低耦合高内聚，创建软件相关对象要用到工厂这一概念。举一个简单的例子，现如今大部分汽车都不会由同一家公司从头到尾生产。大多都是进行B2B的电子商务模式来”组装”出一辆汽车。一条龙的服务会给公司带来极大的负担，对软件设计来说也是如此。所以汽车生产商会向轮胎生产商购买轮胎。而换作软件开发中的时候，就是会向一个工厂类要一个该工厂生产的实际对象。我们把被创建的对象称为“产品”，把创建产品的对象称为“工厂”。

最初接触到这设计模式的时候，我最大的感触就是new 的使用减少了。new就是手动的创建一个对象。在工厂模式的使用下，我创建对象一般会调用某个工厂类。我只要给工厂类输入一些参数，那么工厂就会帮我创建我需要的对象。就像我要买一些轮胎，我只要告诉轮胎厂我需要什么规格的轮胎，轮胎厂就会自动的把我想要的轮胎送过来。我在使用SSM框架开发Web应用的时候，会在test类里使用Spring框架给我的BeanFactory这个类来创建工厂类，然后根据给出的工厂类创建我所需要的javaBean.

在实际开发中，工厂模式不是万能的通常适用于以下的情况：

1. 用户只关注产品的工厂名称，而不去关注工厂的产品的名称。

2. 创建对象的任务只是由多个具体工厂中的某一个完成。

3. 用户不关心产品怎么生产出来的，他只关心产品的品牌如何。

对于第二个条件，是很好容易理解的。当有多个具体的工厂时，我们需要有一个抽象工厂。我们对抽象工厂输入参数，抽象工厂自然会帮我们找到具体的工厂来生产产品。对于不使用抽象工厂的工厂模式一般分简单工厂模式和工厂方法模式。工厂方法模式是简单工厂模式的进化。

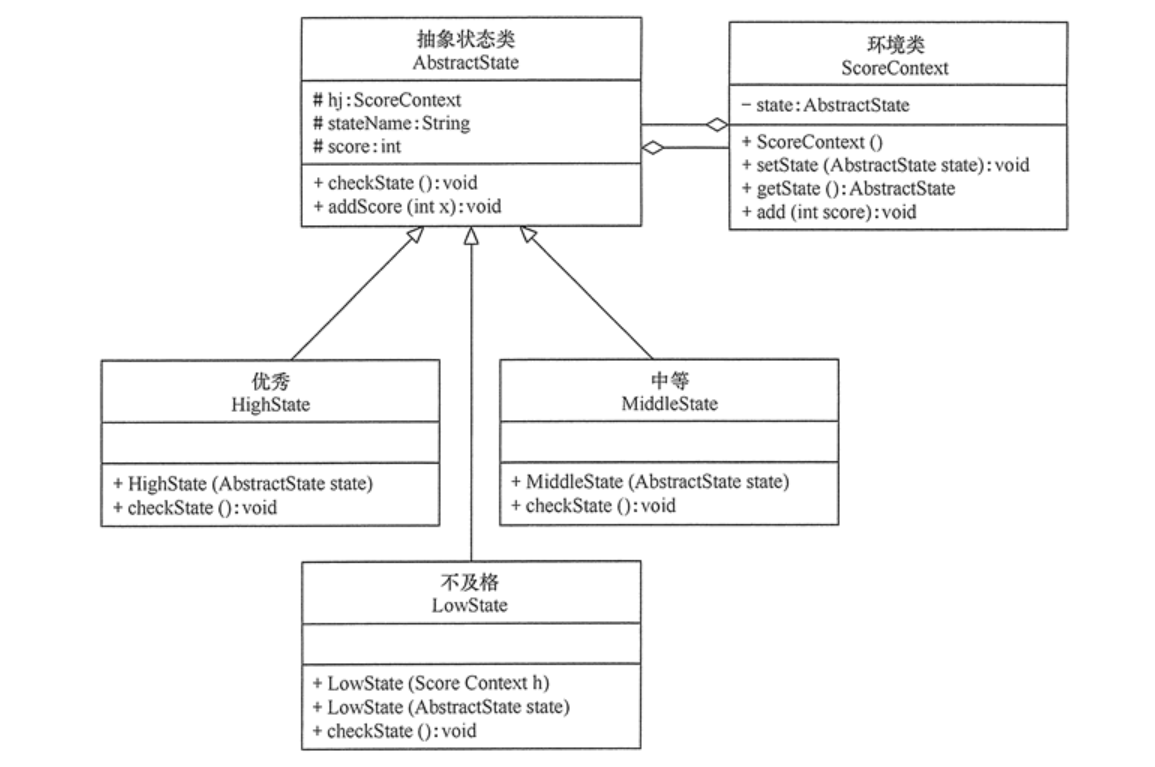
对于工厂方法模式来说，它是利用子类来创建对象。也就是继承。但在遇到该工厂要生产新的一件商品时，我就要专门再写一个类来继承父类工厂。在这里我要强调的是工厂生产新的商品与开头说的工厂专一生产一件产品是两码事。我这里指的是同一产品的不同牌子。对于抽象工厂模式来说，它要在抽象工厂类里添加一个新的接口。然后写一个新的类。在原来的工厂类（非抽象工厂）里添加一个新的方法。在比较了两种工厂模式下，我们可以发现这两种要生产一种新的产品，并没有多少简洁与复杂的区别。但是两种工厂模式创造对象的方式却不相同。工厂方法模式使用子类创造对象，抽象工厂模式使用接口的实现创造对象。

**二、状态模式**

这是非常容易理解的一个模式。在产品开发过程中，对象会随着环境的改变而改变他的状态。比如一个人的情绪。拿我个人来举例子吧。假如我哪天走在街上捡到一百块钱，拿着这一百块钱去消费了，我会高兴很长一段日子。但是当我在去消费的过程中被失主要回一百元的时候，我就会郁闷好一阵子：为什么我捡到钱之后不早点走开。在我从高兴和郁闷的情绪就是我状态的转换。状态的转换会导致行为的变化。而对于这种有状态的对象编程，传统的解决方案是用程序来一步步修改对象的内部属性，等修改完了，状态的转换就结束了。当对象的状态有很多种时，那么要用到修改状态的程序代码量会变得非常多。而且当添加新的状态时，我们就会去修改程序。这就违反了开闭原则（开闭原则：面对对象设计中最重要的原则之一。开：对于组件功能的扩展是开放的，允许对于其功能扩展。闭：对于原有代码的修改是关闭的，即不应该修改原有代码

）。那么怎么解决这个问题呢？

对有状态的对象，把复杂的判断逻辑提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。有利于程序的扩展。通过定义新的子类很容易增加新的状态和转换。不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。当然状态模式也会导致一些问题。它会增加类个数的增加。使用不当会出现程序结构和代码的混乱。



为了更好地解释状态模式，我从网上找了一张图。这是根据学生分数来评价学生优秀、中等、不及格。其中有三个状态类，优秀、中等、不及格。它们都是抽象状态类的子类。环境类里放的是学生分数。通过不同的分数，给学生判断不同的状态。

**三、观察者模式**

为了看懂这个模式，我又花了点时间把面向对象设计课里的7条原则看了一遍。如果说状态模式是状态之间的切换，那么观察者模式就是行为之间的切换。而且还是A对象因为B对象的行为而发生改变。用观察者模式的眼光再回到之前的例子，我会因为100块钱换来了一些零食的行为而感到高兴；也会因为失主把100块钱要回去而郁闷。我把高兴和郁闷改成了感到高兴和感到郁闷。把状态的切换改成行为的切换。

定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。其中有目标与观察者。目标指的是改变的对象的行为。观察者指的是与目标对象依赖的对象，会因目标行为的变化而改变自身的行为的对象。

优点如下：1. 降低目标与观察者的耦合度。

2. 双方建立触发机制，无需用户手动去修改行为。

缺点： 1. 目标与观察者之间的依赖关系没有完全解除，只能减弱耦合

2. 当观察者过多时，发布通知会时间过长。

观察者模式的主要角色如下。

1. 抽象主题：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

2. 具体主题：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

3. 抽象观察者：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

4. 具体观察者：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

这几个角色一般是定死的。我在做项目的过程中没有什么对观察者模式使用的印象。不过Spring等框架一定会使用这个模式。具体观察者继承抽象观察者，具体主题继承抽象主题。直接打交道的是两个抽象的对象。这个模式有应用场景：对象存在一对多关系，一个对象的改变会影响其他对象。

**四： 享元模式**

内存是非常珍贵的资源，在实际的开发中必须考虑到。在学生阶段里做的项目，绝大多数都是玩具软件，不会考虑到性能。享元模式是我这次第一次接触。享元模式中存在两种状态：内部状态和外部状态。这个模式运用共享技术来有效的支持大量细粒度对象的复用。它通过共享已经存在的相同或相似对象来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源利用率。

上面我说明了享元模式的用处和定义。但是有几个名词我想仔细解释一下。因为这几个也是我之前遇到过，但没怎么去了解。

**细粒度和粗粒度**：（这两个名词我觉得我想靠文本可能讲不清楚）细粒度与粗粒度相对立。网上看见一句话“粒度似乎是根据项目模块划分的细致程度区分，一个项目模块分得越多，每个模块越小，负责的工作越细，就说明粒度越细，否则为粗粒度。”但是这句话是没有完全被认同的，这要靠意会。有个简单的例子：一个user类，其中有email类。一个user会有多个email。第一种表示email的方法是用list来保存email，第二种是再设计一个email类，user的email属性由email对象组成。在这个例子中，相对于第一种方法，第二种方法显然是细粒度的。再举个例子吧。我在《教务系统》分析报告里画的UML图中其实也有体现粗细粒度的时序图。在学生选课的时序图中，通过中间部件，一步步传输请求，最终请求到达数据库，数据库再返回数据。这是细粒度的体现。为什么说这是细粒度？再看看我的学生报名的时序图：在学生报名的图里，学生报名的请求我没有画到数据库操作的步骤，反而画了个报名器，由报名器来实现数据库操作。所以这个是粗粒度的。但是还是有问题。我的两个例子都是对比之后才能确定粗细粒度，怎么样才能普遍得确认粗细粒度？通过查找，我得出来结论。粗粒度一般指类级别，不考虑某个特例。对所有对象一视同仁。细粒度表示实例对象，即考虑具体对象得实例。

**内部状态**：不会随着环境的改变而改变的可共享部分。

**外部状态：**随环境改变而改变的不可以共享部分。

理解了上述的三个名词，那么理解这个模式就会简单许多。享元模式的组成部分有抽象享元角色、具体享元角色、非享元角色、享元工厂角色。

抽象享元角色：作为所有具体享元类的基类，它为具体享元规范需要实现的公共接口，非享元的外部状态以参数的形式通过方法传入。

具体享元角色：实现抽象享元角色中所规定的接口。

非享元角色：是不可以共享的外部状态。

享元工厂角色：负责创建和管理享元角色。当用户请求一个享元对象时，享元工厂检查系统中是否存在符合要求的享元对象，如果存在则直接提供，否则创建一个新的享元对象。

举个实际例子吧。拿围棋来说。围棋分两种：黑棋与白棋。那在这个模式下，角色就很好确认了。黑棋与白棋就是两个具体享元角色。在棋盘上的位置就是非享元角色。抽象享元角色就是棋子。对于围棋来说，使用享元模式毫无疑问就会节省大量的内存空间消耗。所以当系统中会出现大量相同或相似的对象时，使用享元模式是一个非常漂亮的举措。