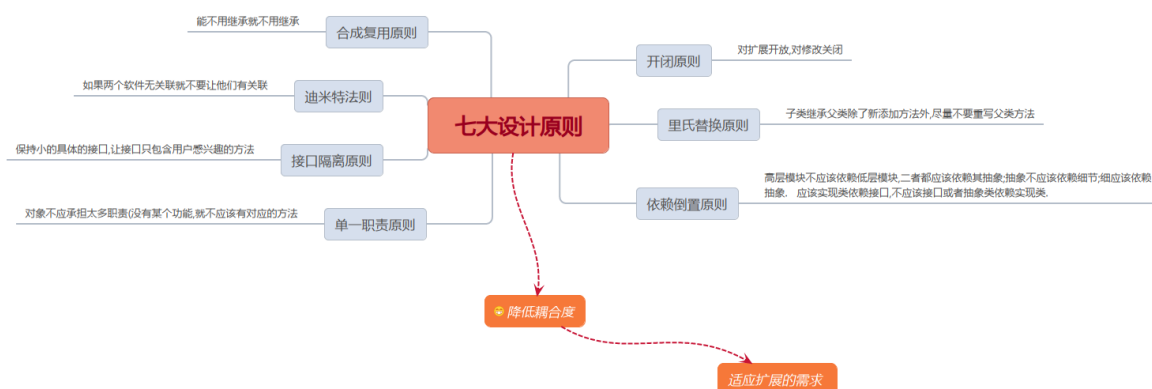


# 设计模式期末复习

## 类的基本知识

- 类（class）是指具有相同属性方法和关系的对象的抽象，它封装了数据和行为，是面向对象程序设计（OOP）的基础，具有**封装性**，**继承性**，和**多态性**三大特征。
- 接口（Interface）是一种特殊的类，它具有类的结构但不能被实例化，**只可以被子类实现**。它**包含抽象操作**，**但不包含属性**，它描述了类或组件对外可见的动作。
- 在软件系统中，类不是孤立存在的，类与类之间存在各种关系，根据类与类之间的耦合度从弱到强排列，UML中的类图有以下几种关系：**依赖<关联<聚合<组合<泛化=实现**。其中泛化和实现的耦合度相等。
- **聚合**是关联的一种，是强关联关系，是整体与部分的关系，是**has-a**的关系。**组合**关系也是关联的一种，也表示类之间的整体与部分的关系，但他是一种更强烈的聚合关系，是**contains-a**关系。**泛化**关系是对象之间耦合度最大的一种关系，表示一般与特使的关系，是父类与子类之间的关系，是一种继承关系，是**is-a**的关系。

## 设计原则



- **开闭原则**，修改关闭，拓展开放。
- 依赖倒置原则，实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合，其核心思想是**面向接口编程**，不要面向实现编程。
- 单一职责原则，核心是控制**类的粒度大小**，将对象解耦，提高其内聚性，如果遵循单一职责可以降低类的复杂度，类的逻辑简单；提高类的可读性，复杂度降低，可读性提高；提高了系统的可维护性；变更引起的风险降低。
- 接口隔离原则，要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。
- 迪米特法则，只与你的直接朋友交谈，**不跟“陌生人”说话**；如果两个软件实体**无需**直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。这也会造成系统的不同模块之间的通信效率降低，也会使系统的不同模块之间不容易协调。

**广义的迪米特法则在类的设计上的体现：**

优先考虑将一个类设置成不变类。

尽量降低一个类的访问权限。

谨慎使用Serializable。

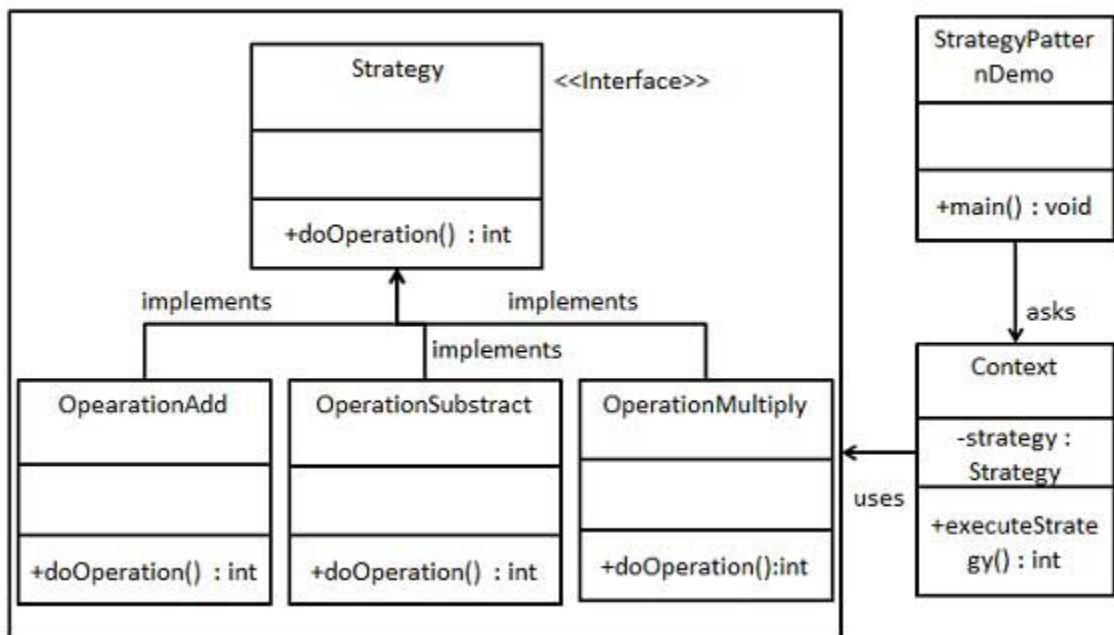
尽量降低成员的访问权限

- 合成复用原则，又叫组合/聚合复用原则，他要求在软件复用时，要尽量**先使用组合或者聚合等关联关系**实现，其次才考虑使用继承关系实现。
- 里氏替换原则，子类**可以扩展**父类的功能，但**不能改变**父类原有的功能。
- 设计模式，设计模式比库跟高级，告诉我们如何组织类和对象以解决某种问题
- 库和框架提供了我们某种特定的实现，让我们的代码可以轻易的引用，但这并不算设计模式，有时候库和框架也会用到设计模式。
- 模式并不是代码，而是针对设计问题的通用 解决方案

## 设计模式

### 1. 策略模式

- 该模式定义了一系列算法，并将每个算法**封装**起来，使他们可以相互替换，且算法的变化不会影响使用算法的用户。策略模式属于对象行为模式，他通过对对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派不同的对象对这些算法进行管理。
- **动静分离**，将动的代码开放出去，不写死在对象里面。
- 类图：



鸭子模型

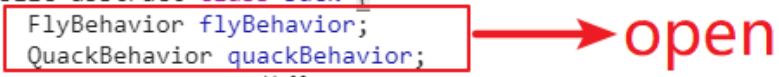
```

> public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public void swim(){}
    public abstract void display();
    public void fly(){
        flyBehavior.fly();
    }
    public void quack(){
        quackBehavior.quack();
    }

    public void setFlyBehavior(FlyBehavior flyBehavior) {
        this.flyBehavior = flyBehavior;
    }

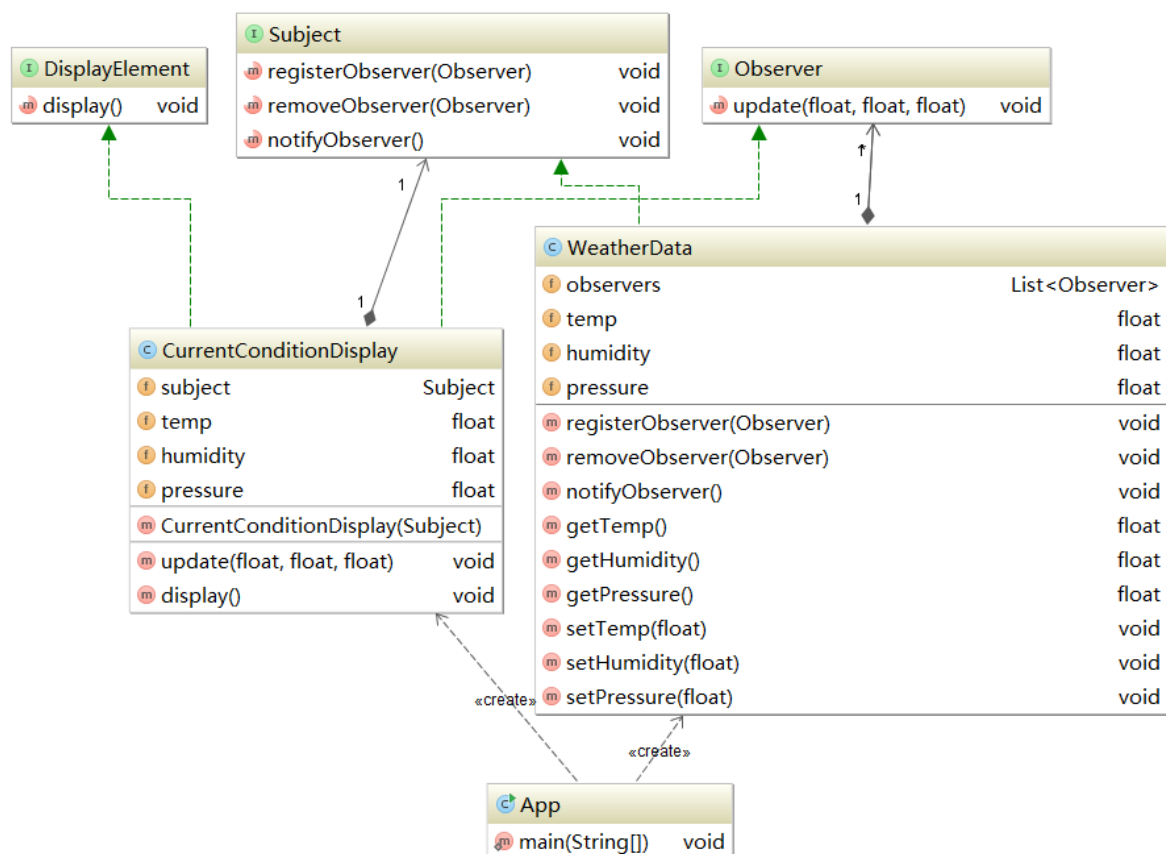
    public void setQuackBehavior(QuackBehavior quackBehavior) {
        this.quackBehavior = quackBehavior;
    }
}

```



## 2. 观察者模式

- 指多个对象之间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖与它的对象都会得到通知并自动更新。
- 实现观察这模式时要注意具体的目标对象和具体观察者对象之间**不能直接调用**，否则使两者之间紧密耦合起来，违反了面向对象的设计原则。
- 抽象主题（Subject）角色：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
- 具体主题（Concrete Subject）角色：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**
- 抽象观察者（Observer）角色：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。
- 具体观察者（Concrete Observer）角色：**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。
- 观察者模式定义对象之间的一对多依赖，这样一来，当一个对象状态发生改变的时候，它所有的依赖者都会收到通知并自动更新。
- 主题（也就是可观察者）用一个共同的接口来更新观察者
-



报社和订阅者之间，在顶层设计的时候，可以让两个对象之间松耦合，但是他们依然可以交互，但是不太清楚彼此的细节，报社只知道按时将报纸投递到某个地址，具体是哪个人来接受这个报纸，具体什么时候接受却一无所知，但是这并不影响投递部的人的工作。

而对于订阅者而言同样，他们不需要知道这份报纸具体由谁来投递，以何种方式投递过来。

**松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的相互依赖降到最低。**

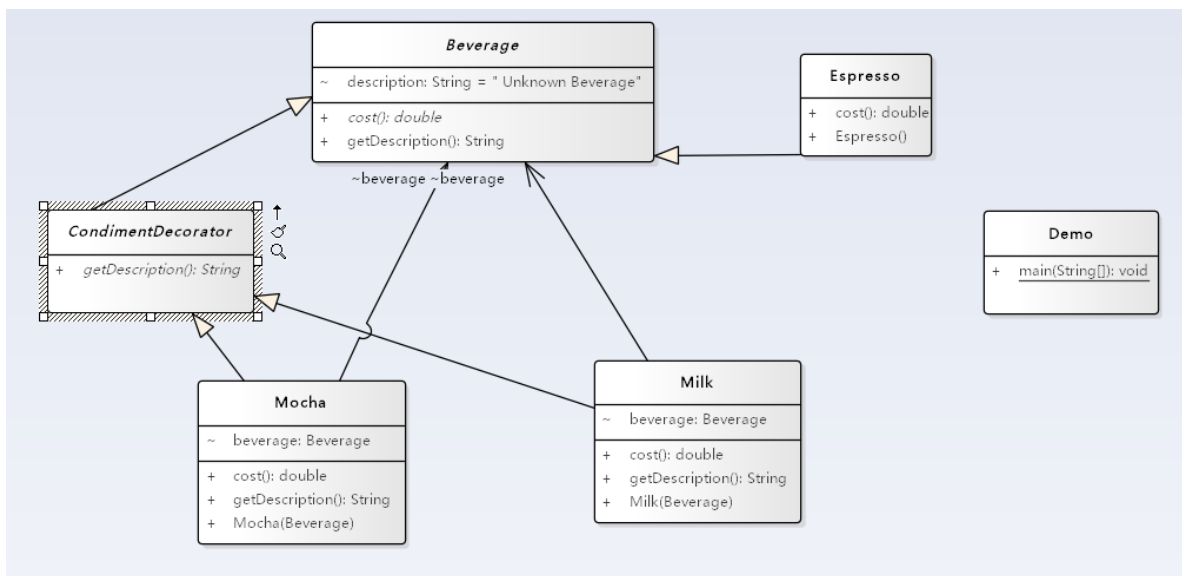
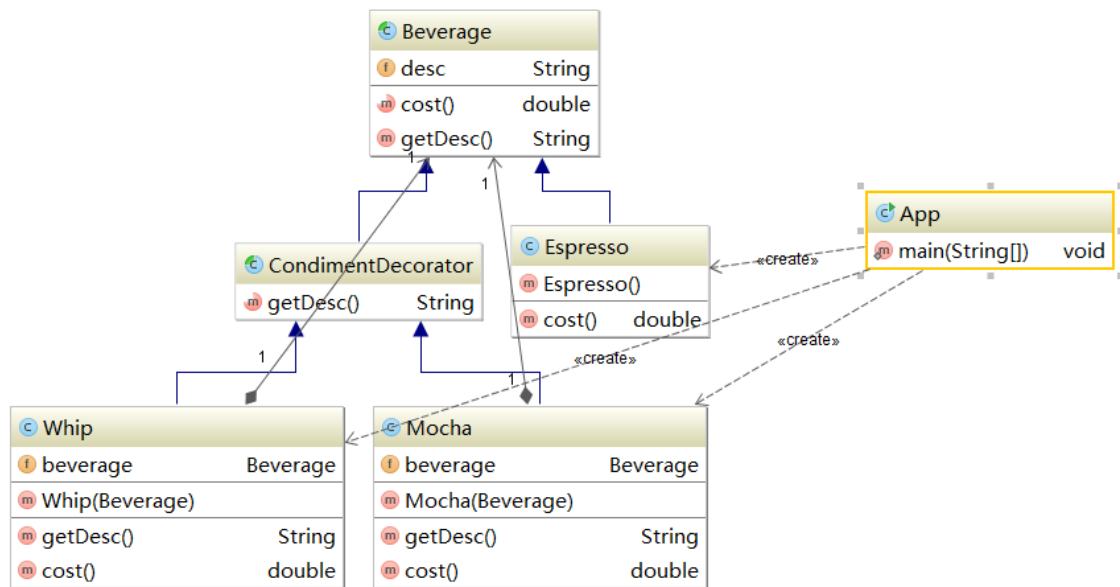
应用场景:

1. 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。
2. 一个对象的改变需要同时改变其他对象，而不知道具体有多少对象需要改变。
3. 一个对象必须通知其他对象，而有不能假定其他对象是谁。对象松耦合。

### 3. 装饰者模式

- 装饰者和被装饰者对象有相同的超类型
- 可以用一个或多个装饰者包装一个对象
- 装饰者可以在所委托被装饰者的行为之前或之后，加上自己的行为，已达到特定的目的。
- 装饰者模式动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更有弹性地替代方案

装饰者模式可以实现和继承类似的功能，比继承更灵活。



步骤 1 创建一个接口：

步骤 2 创建实现接口的实体类。

m步骤 3 创建实现了 *Shape* 接口的抽象装饰类。

步骤 4 创建扩展了 *ShapeDecorator* 类的实体装饰类。

步骤 5 使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

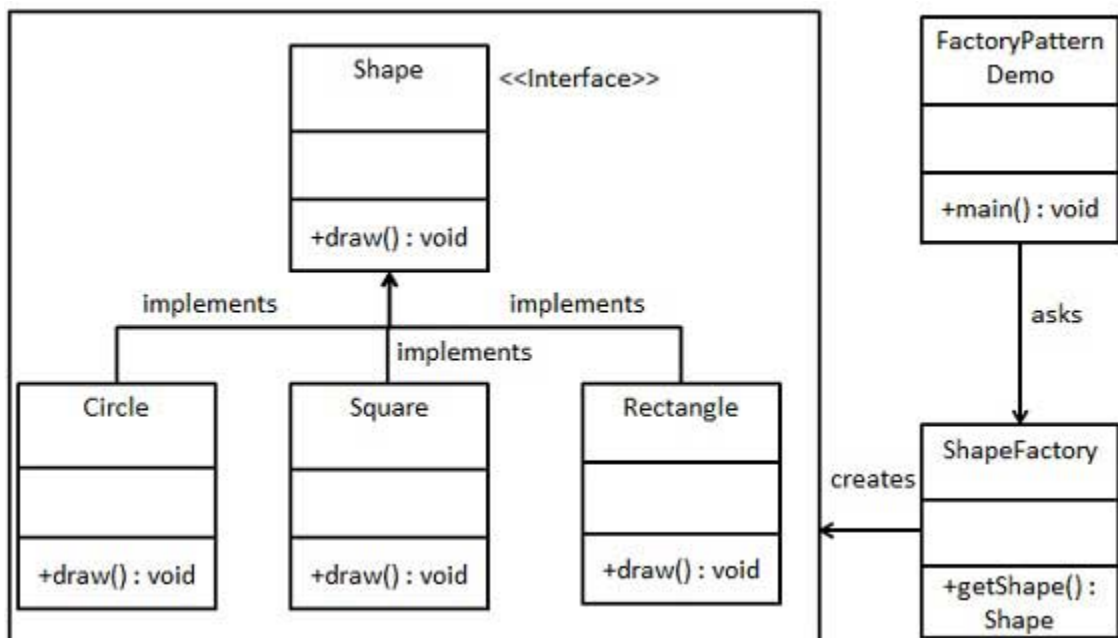
步骤 6 执行程序，输出结果：

## 4. 工厂模式

- 定义：定义了一个创建对象的接口，但由于子类决定要实例化的类是哪一个，工厂方法让类把实例化推迟到了子类。

- 创建者类：抽象，定义了抽象的工厂方法让子类去实现次类产品的制造
- 产品类：抽象，工厂生产的产品。

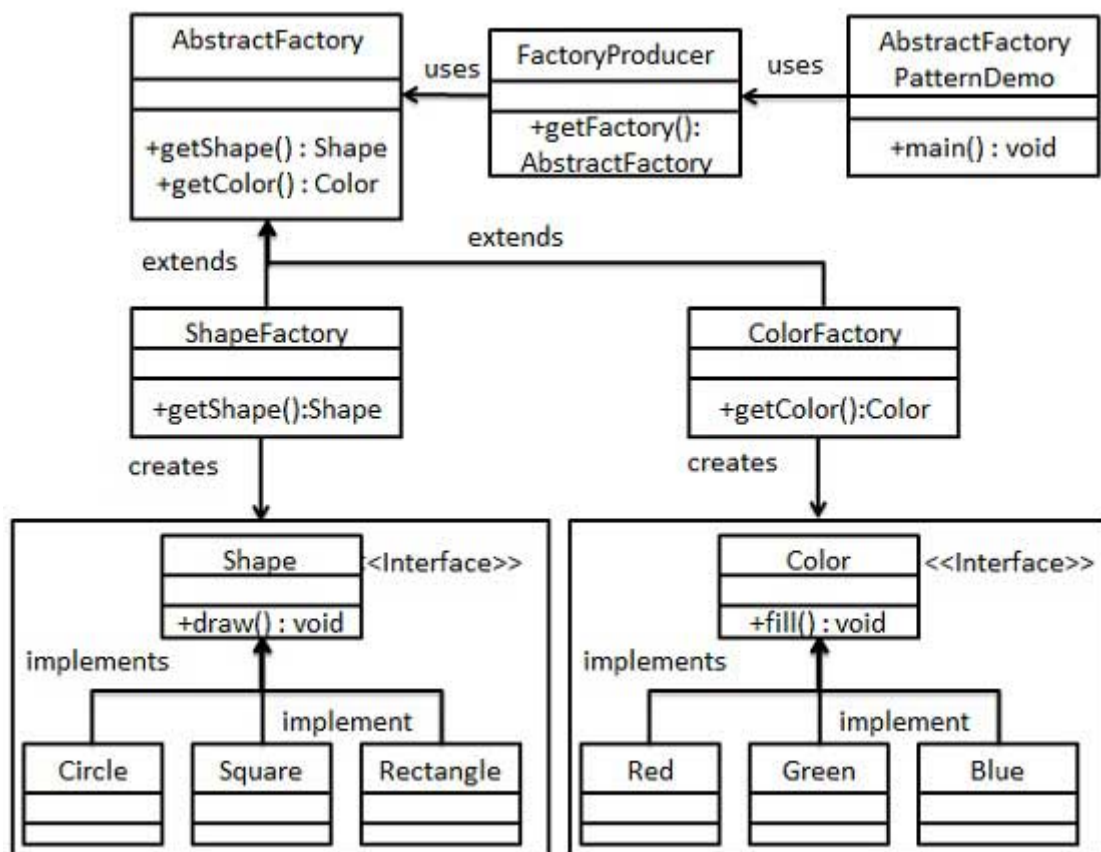
简单工厂类图：



工厂模式：

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类

类图：

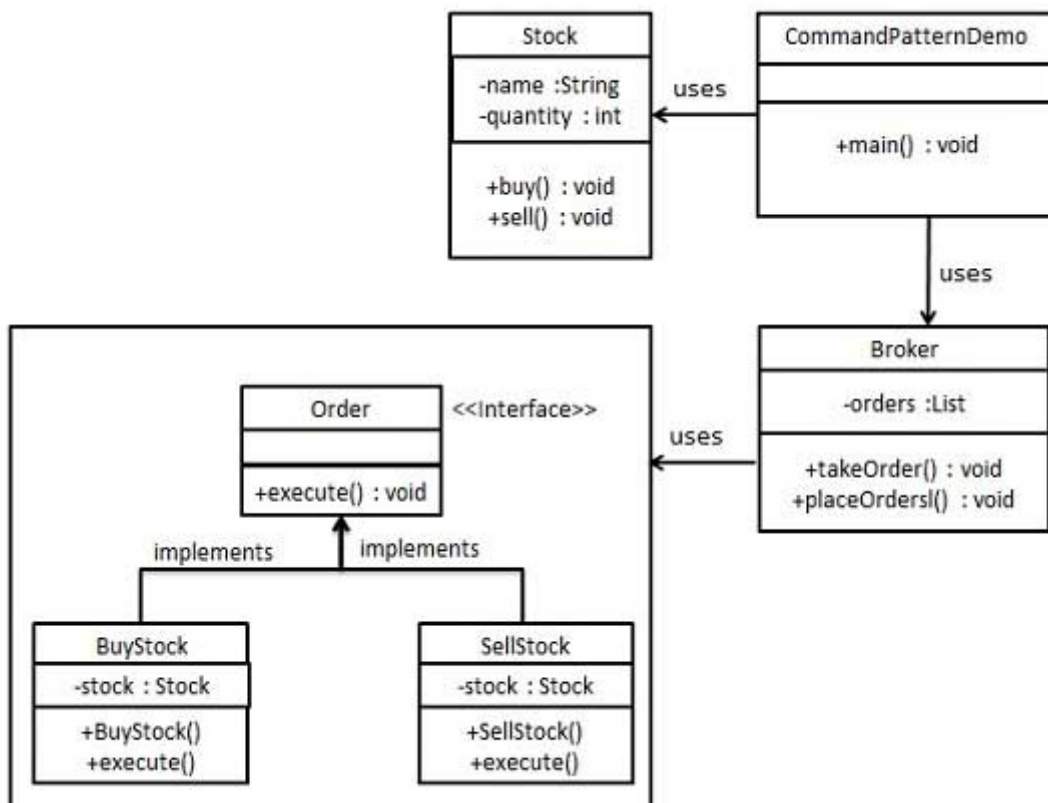


## 命令模式

命令模式是一种数据驱动的设计模式<sup>2</sup>，它属于行为型模型。请求以命令的形式包裹在对象中，并传给调用对象，调用对象可以寻找处理该命令的合适的对象，并把该命令传给响应的对象，该对象执行命令。

意图：将一个请求封装成一个对象，从而使可以用不同的请求对客户进行参数化。

将行为请求者与行为执行者解耦

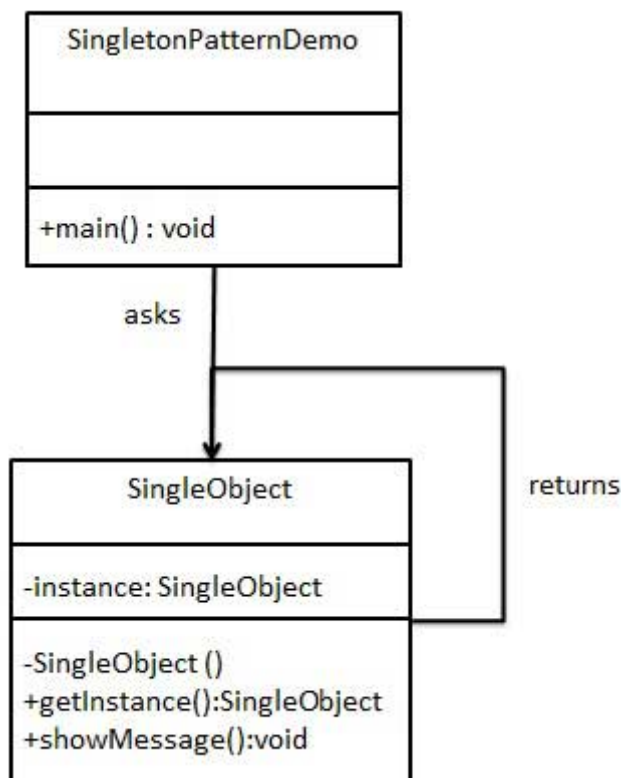


## 单例模式

这个类提供了一种访问器唯一的对象方式，可以直接访问，不需要实例化该类的对象。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。



实现方式：



### 1. 懒汉式，线程不安全.

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

### 2. 懒汉式，线程安全，

```
public class Singleton {  
    private volatile static Singleton instance; //volatile关键字特性包括可见性，  
    原子性，禁止指令重排  
    private Singleton (){}  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

### 3. 饿汉式

**描述：**这种方式比较常用，但容易产生垃圾对象。

**优点：**没有加锁，执行效率会提高。

**缺点：**类加载时就初始化，浪费内存。

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