دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

درس آز مهندسی نرم‌افزار

استاد سیدحسن میریان حسین‌آبادی

گزارش آزمایش دوم

حسین ذاکری‌نیا، مصطفی قدیمی

سؤال ۱. Decorator

برای نوشتن این کلاس‌ها، همان‌طور که در ویدیوی آموزشی گفته شد، یک مجموعه تست به ما داده شده است و باید قدم به قدم و مرحله به مرحله شروع به برطرف کردن خطاها کنیم تا در نهایت همه‌ی تست‌ها پاس شوند.
باید یک interface به نام Beverage ایجاد کنیم که یک فیلد به نام description دارد و سه متد getDescription، cost را پیاده‌سازی کرده است.

```
public interface Beverage {
    public String description = null;
    public String getDescription();
    public Double cost();
}
```

حال باید چهار کلاس HouseBlend، DarkRoast، Espresso و Decaf را ایجاد نماییم و جزئیات (شامل اسم و قیمت با توجه به تست‌های داده‌شده) را اضافه کنیم. برای مثال کلاس Espresso به شکل زیر است:

```
public class Espresso implements Beverage {
    @Override
    public String getDescription() {
        return "Delicious Espresso";
    }

    @Override
    public Double cost() {
        return 99.1 ;
    }
}
```

در این مرحله باید یک کلاس decorator به نام CondimentDecorator را اضافه کنیم تا با ارتباطی که با Beverage دارد، قابلیت انعطاف و پویایی را اضافه کند.

```
public abstract class CondimentDecorator  implements Beverage {
    private Beverage beverage;

    public CondimentDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
```

```
        return beverage.getDescription();
    }

    @Override
    public Double cost() {
        return beverage.cost();
    }
}
```

حال با استفاده از super در زیر کلاس‌های مربوط به decorator این پویایی به برنامه اضافه می‌شود.
برای مثال، کلاس Mocha به شکل زیر است:

```
public class Mocha extends CondimentDecorator {

    public Mocha(Beverage beverage) {
        super(beverage);
    }

    public String getDescription() {
        return super.getDescription() + " with mocha";
    }

    public Double cost() {
        return super.cost() + 20.0 ;
    }
}
```

در نهایت تمامی تست‌ها پاس شد.

سؤال ۲. Bridge

در این قسمت، برای این‌که کارها و مسئولیت‌ها را به کلاس‌های متفاوت (Abstract و پیاده‌سازی‌ها) را از یک‌دیگر جدا کنیم، از این الگوی طراحی استفاده می‌کنیم.
با توجه به Test Driven Development ابتدا باید تست‌ها را طراحی کنیم و سپس اقدام به نوشتن منطق برنامه کنیم.

```
public class PowerTest {
    @Test
    public void testPowerStandardRecursive() {
        Power power = new PowerImplStandard(new MultImplRecursive());
        Assert.assertEquals(81, power.power(3, 4), 0) ;
    }

    @Test
    public void testPowerRecursiveRecursive() {
        Power power = new PowerImplRecursive(new MultImplRecursive());
        Assert.assertEquals(81, power.power(3, 4), 0) ;
    }

    @Test
    public void testPowerRecursiveStandard(){
        Power power = new PowerImplRecursive(new MultImplStandard());
        Assert.assertEquals(81, power.power(3, 4), 0) ;
    }

    @Test
    public void testPowerStandardStandard(){
        Power power = new PowerImplStandard(new MultImplStandard());
        Assert.assertEquals(81, power.power(3, 4), 0) ;
    }
}
```

کلاس مربوط به توان را Abstract در نظر می‌گیریم زیرا باید یک فیلد از Interface ضرب داشته باشد و در جایی که نیاز داریم، از پیاده‌سازی‌ها مربوط به آن استفاده کند:

```
public abstract class Power {
    protected Mult mult;

    public Power(Mult mult) {
        this.mult = mult;
    }

    public abstract Integer power(int a, int b);
}
```

در قسمت بعدی interface ضرب را پیاده‌سازی می‌کنیم:

```
public interface Mult {
    public Integer mult(int a, int b);
}
```

حال کافی است پیاده‌سازی‌های مختلف ضرب و توان را به کار ببندیم.
پیاده‌سازی‌های انجام شده:

- پیاده‌سازی معمولی
- پیاده‌سازی بازگشتی

پیاده‌سازی بازگشتی ضرب

```
public class MultImplRecursive implements Mult {
    @Override
    public Integer mult(int a, int b) {
        if (b == 0) {
            return 0;
        }
        return a + mult(a, b - 1);
    }
}
```

پیاده‌سازی معمولی ضرب

```
public class MultImplStandard implements Mult {

    @Override
    public Integer mult(int a, int b) {
        int result = 0;

        while(b != 0) {
            result += a;
            b -= 1;
        }

        return result;
    }
}
```

پیاده‌سازی بازگشتی توان

```
public class PowerImplRecursive extends Power {

    public PowerImplRecursive(Mult mult) {
        super(mult);
    }

    @Override
    public Integer power(int base, int exponent) {
        if (exponent != 0) {
            return mult.mult(base, power(base, exponent - 1));
        }
        else {
            return 1;
        }
    }
}
```

پیاده‌سازی معمولی توان:

```
public class PowerImplStandard extends Power {

    public PowerImplStandard(Mult mult) {
        super(mult);
    }

    @Override
    public Integer power(int base, int exponent) {
        int result = 1;
        while (exponent != 0) {
            result = mult.mult(result, base);
            --exponent;
        }
        return result;
    }
}
```

به این ترتیب تست‌های مربوط به این قسمت نیز پاس می‌شوند.