设计原则-模板方法模式

这篇文章用于学习设计原则中的模板方法模式。
很有趣的一点是，之前的数据结构课程中，我和大家一起讨论的也是模板方法，只不过是C++中的模板方法。

定义

首先，学习一个东西，查个Wiki再说，下面给出定义。

“**模板方法模型**是一种行为设计模型和设计模式。**模板方法**（template method）是一个定义在父类的方法，负责处理流程、算法的不变部分。**模板方法**会调用多个定义在父类的其他工具方法（helper method），而这些方法是算法的可变部分，有可能只是抽象方法并没有实现。**模板方法**仅决定这些抽象方法的执行顺序，这些抽象方法由子类负责实现，并且子类不允许覆盖模板方法（即不能重写处理流程）。

这种设计模式是一种控制反转的实现方式。因为高层代码不再确定（控制）算法的处理流程。”

——摘自维基百科。

看起来每个字都认识，但有点读不太懂是不是，但是真正理解起来并不困难。

引例

我和一个朋友最近厌倦了joja公司996的生活，转而去继承爷爷的农场，~~从此过上了627的生活~~。

为了复兴农村，我们决定通过酿酒来赚钱，在书中学到，酿酒需要以下流程：

1. 购买种子
2. 耕地、播种。
3. 选择洒水器（普通洒水器，优质洒水器或者铱洒水器）
4. 收获作物。
5. 将作物放入酿酒桶（小桶，陈酿桶）

农场的每个季节，我们能种的作物是不同的，显然一套流程下来我们只能满足一个季度的需求。但是生活所迫，我们每个季度都需要卖酒挣钱，每次都进行一遍流程显然是很麻烦的，所以我们静下心来整理了一下发现，流程中的第2、4、5步都是重复的，如果我们能够使用某种方式，提高该流程的复用性，又能保证该流程的步骤不会出错，就太好了。所以，对他使用模板方式模式吧！

这里我们将酿酒的流程抽象出来，编写一个抽象父类：

```
public abstract class WinemakingProcess {
    final void makeWine() {
        buySeed();      //购买种子
        tillingAndSowing();     //耕种、播种
        chooseSprinkler();      //选择洒水器
        harvest();      //收获
        if(isUseCask()) {       //是否使用木桶
            makeWineWithCask();     //使用木桶酿酒
            return;
        }
        makeWineWithKeg();      //使用小桶酿酒
```

```
    }
    abstract void buySeed();

    void tillingAndSowing() {
        System.out.println("Step2: 播种、耕种");
    }

    abstract void chooseSprinkler();

    void harvest() {
        System.out.println("Step4: 收获农作物");
    }

    void makeWineWithKeg() {
        System.out.println("Step5: 使用小桶进行酿造");
    }

    void makeWineWithCask() {
        System.out.println("Step5: 使用木桶进行陈酿");
    }
    //钩子方法：
    boolean isUseCask() {
        return false;
    }
}
```

接下来，我们需要进行具体流程了，春天我们使用普通种植大黄，因此利用子类继承该父类，实现大黄的流程。

```
class Rhubarb extends WinemakingProcess {
    @Override
    void buySeed() {
        System.out.println("Step1: 购买大黄种子");
    }

    @Override
    void chooseSprinkler() {
        System.out.println("Step3: 使用普通洒水器");
    }
}
```

同理，夏天我们打算种植杨桃，但是我们更想售卖陈酿的杨桃果酒，这里采用钩子方法（在模板方法模式的父类中，可以定义一个方法，它默认不做任何事，子类可以视情况要不要覆盖它，该方法称为“钩子”。），改用木桶进行陈酿。

```
class Starfruit extends WinemakingProcess {
    @Override
    void buySeed() {
        System.out.println("Step1: 购买杨桃种子");
    }

    @Override
    void chooseSprinkler() {
        System.out.println("Step3: 选择优质洒水器");
```

```
    }

    @Override
    boolean isUseCask() {
        //采用木桶陈酿
        return true;
    }
}
```

秋天我们则种植南瓜，利用铱制洒水器浇水，最终酿为南瓜果汁。

```
class Pumpkin extends WinemakingProcess {
    @Override
    void buySeed() {
        System.out.println("Step1: 购买南瓜种子");
    }

    @Override
    void chooseSprinkler() {
        System.out.println("Step3: 使用铱制洒水器");
    }
}
```

那么，方案我们制定好了，我们最终定一个农夫类来执行该流程吧：

```
public class Farmer {
    public static void main(String[] args) {
        System.out.println("------制作大黄果酒------");
        WinemakingProcess rhubarb = new Rhubarb();
        rhubarb.makeWine();
        System.out.println("----------------------");

        System.out.println("----制作陈酿杨桃果酒-----");
        WinemakingProcess starfruit = new Starfruit();
        starfruit.makeWine();
        System.out.println("----------------------");

        System.out.println("------制作南瓜果汁------");
        WinemakingProcess pumpkin = new Pumpkin();
        pumpkin.makeWine();
        System.out.println("----------------------");
    }
}
```

执行！我们完美的酿完了酒，赚了很多很多的钱。

```
————制作大黄果酒————
Step1: 购买大黄种子
Step2: 播种、耕种
Step3: 使用普通洒水器
Step4: 收获农作物
Step5: 使用小桶进行酿造
————————————————————
——制作陈酿杨桃果酒————
Step1: 购买杨桃种子
Step2: 播种、耕种
Step3: 选择优质洒水器
Step4: 收获农作物
Step5: 使用木桶进行陈酿
————————————————————
————制作南瓜果汁————
Step1: 购买南瓜种子
Step2: 播种、耕种
Step3: 使用铱制洒水器
Step4: 收获农作物
Step5: 使用小桶进行酿造
————————————————————

进程已结束，退出代码为 0
```

分析

另一个定义

通过以上流程，相信我们已经基本明白了模板方法模式是用来做什么的。
这里，我们给出另外一个定义。
模板方法模式：定义一个操作中的算法的框架， 而将一些步骤延迟到子类中。 使得子类可以不改 变一个算法的结构即可重定义该算法的某些特定步骤。
换句话说，我们可以在父类中定义一个执行框架，子类只重写流程中的方法。

优点

- 遵循“开闭原则”
- 既统一了算法，也提供了很大的灵活性
- 提高了代码的复用度

缺点

- 调用控制反转，一般情况下，是程序执行中子类调用父类的方法，但是在该模式下却变为了父类去调用子类的方法，这会使得程序难以跟踪。
- 每一个不同的实现都需要一个子类实现，引起子类泛滥。

总结

- 模板方法模式是一种类的行为型模式，在它的结构图中只有类之间的继承关系，没有对象关联关系。
- 模板方法模式是基于继承的代码复用基本技术，模板方法模式的结构和用法也是面向对象设计的核心之一。在模板方法模式中，可以将相同的代码放在父类中，而将不同的方法实现放在不同的子类中。
- 为了防止子类改变模板方法中的算法骨架，一般将模板方法声明为final。
- 策略模式和模板方法都是用于封装算法，前者是利用组合和委托模型，而后者则是继承。