## 第25章 遵循套路，不走弯路——模版方法模式

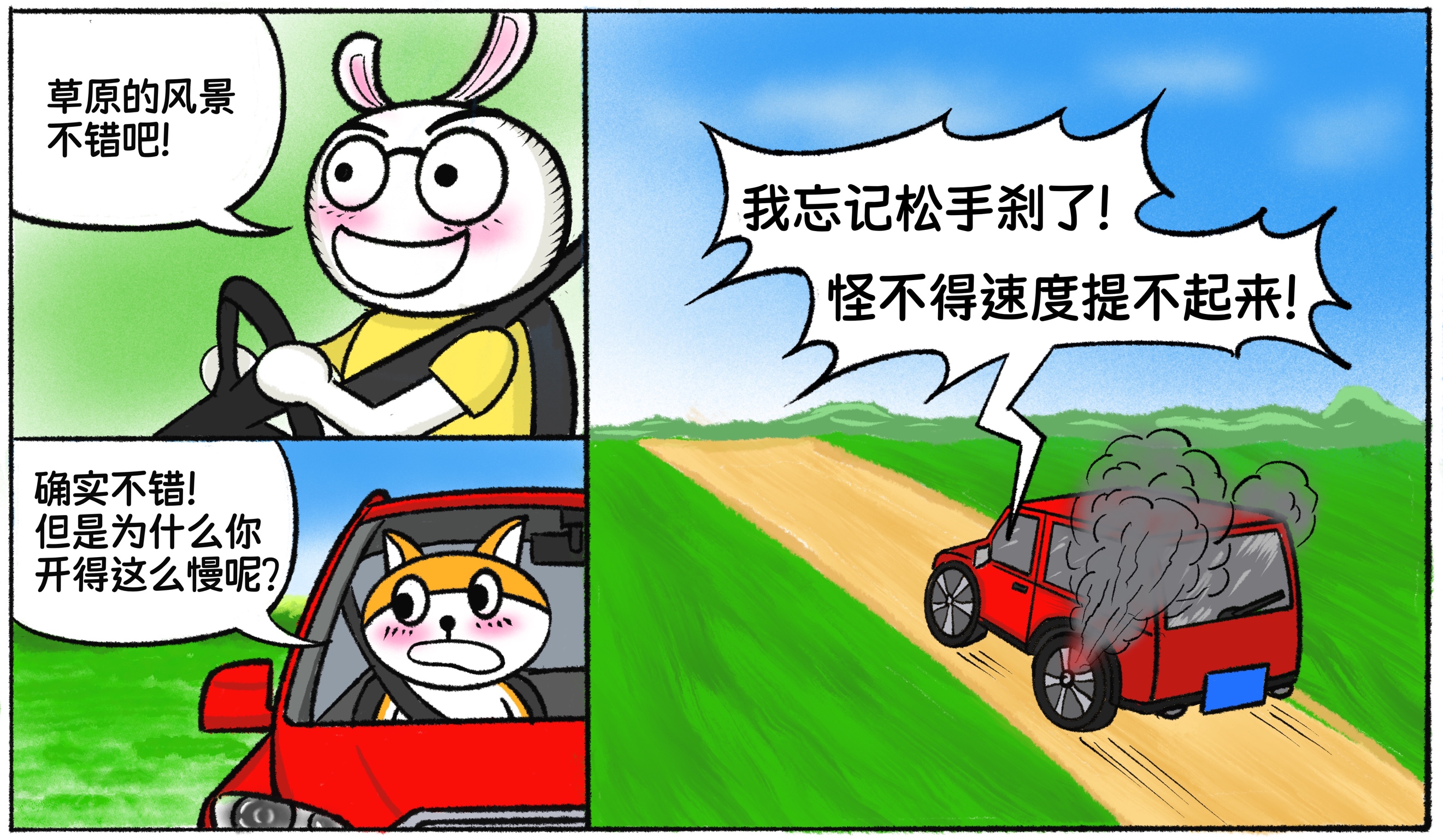
## 25.1 自驾草原行，意外出事故

熊小猫：兔小白，听说你上周末自驾去草原了，玩得怎么样？

兔小白：草原的景色确实很美，只是开车比较累。我还出了点小事故。

熊小猫：草原地广人稀，没有多少车辆，你怎么出的事故呢？

兔小白：哎，我朋友的车是手动挡，我开不习惯......还好是我单方事故。



熊小猫：居然忘了松手刹！其实手动挡和自动挡汽车的起步流程差不多，只是细节上有差别。再说，不管手动挡还是自动挡，都需要松手刹。你对自动挡那么熟悉，不应该忘记松手刹这一步呀。

兔小白：起步流程我当然清楚，但是手动挡需要操作离合器，手忙脚乱中，我忘了松手刹。

熊小猫：这也可以理解，毕竟是人就可能犯错嘛！人又不是机器。不过在软件开发中，有办法解决此类问题。

兔小白：软件也是人写的，不可能一点错误不犯呀！

熊小猫：我的意思不是开发的软件没问题，而是有一种设计模式可以保证这种流程化、模版化的操作不会漏掉某一个步骤。这种设计模式叫做模版方法模式。

你可以思考一下，咱们生活中的很多事情都是模版化的，在摸清套路后，类似的事情就可以按部就班地去执行，只需要改变某些执行的细节。例如学习，无论学什么内容，都要经过预习、听课、练习这三个步骤。开发程序也一样，如果开发的是经典三层架构的系统，大部分需求都需要开发Controller、Service、Dao类，还有对应的单元测试类。一旦你开发完第一个需求，后面的需求都可以按照同样的套路开发。

兔小白：你让我想起做PPT，我一般都是找个模版改一改。你说的模版方法是不是类似？

熊小猫：是这个意思，但还是有些区别的。咱们做个练习，你就明白啦！

## 25.2 程序出Bug，不挂挡也能开车

熊小猫：练习很简单，你先写一段程序，演示自动挡汽车的起步过程。

兔小白：自动挡汽车我熟悉，很快搞定！

10分钟后，兔小白写完了代码。

兔小白：我先定义了司机接口Driver。接口中只有一个startCar方法。

public interface Driver {  
    void startCar();  
}

AtCarDriver是自动挡汽车司机类，实现Driver接口，按照自动挡汽车起步的步骤实现startCar方法。

public class AtCarDriver implements Driver{  
    public void startCar(){  
        System.out.println("踩刹车");  
        System.out.println("一键启动");  
        System.out.println("挂D档");  
        System.out.println("放开电子手刹");  
        System.out.println("松刹车");  
        System.out.println("踩油门");  
   }  
}

客户端代码如下。

Driver atCarDriver = new AtCarDriver();  
atCarDriver.startCar();

程序输出如下。

踩刹车  
一键启动  
挂D档  
放开电子手刹  
松刹车  
踩油门

熊小猫：没问题，你继续来实现手动档汽车起步。

5分钟后，兔小白完成了开发。

兔小白：我新增加了一个Driver接口实现类MtCarDriver。

public class MtCarDriver implements Driver{  
    public void startCar(){  
        System.out.println("踩刹车");  
        System.out.println("踩离合");  
        System.out.println("插入钥匙");  
        System.out.println("转动钥匙打火");  
        System.out.println("放开手刹");  
        System.out.println("松刹车");  
        System.out.println("抬起离合");  
        System.out.println("踩油门");  
   }  
}

熊下猫：你开发的很快嘛！

兔小白：手动档和自动档汽车的起步过程差不多，刚才我直接把AtCarDriver的startCar方法拷贝到MtCarDriver中，稍作修改就搞定了！这就是你说的模版方法吧？

熊小猫：你这是复制、粘贴，哪里是模版方法呀！你运行程序看看。

兔小白：没问题，我来运行程序。

踩刹车  
踩离合  
插入钥匙  
转动钥匙打火  
放开手刹  
松刹车  
抬起离合  
踩油门

熊小猫：咦？怎么没挂档？汽车是怎么开起来的？

兔小白：哎呀......我修改代码时，一不小心把挂档操作的代码误删除了！

## 25.3 起步操作模版化

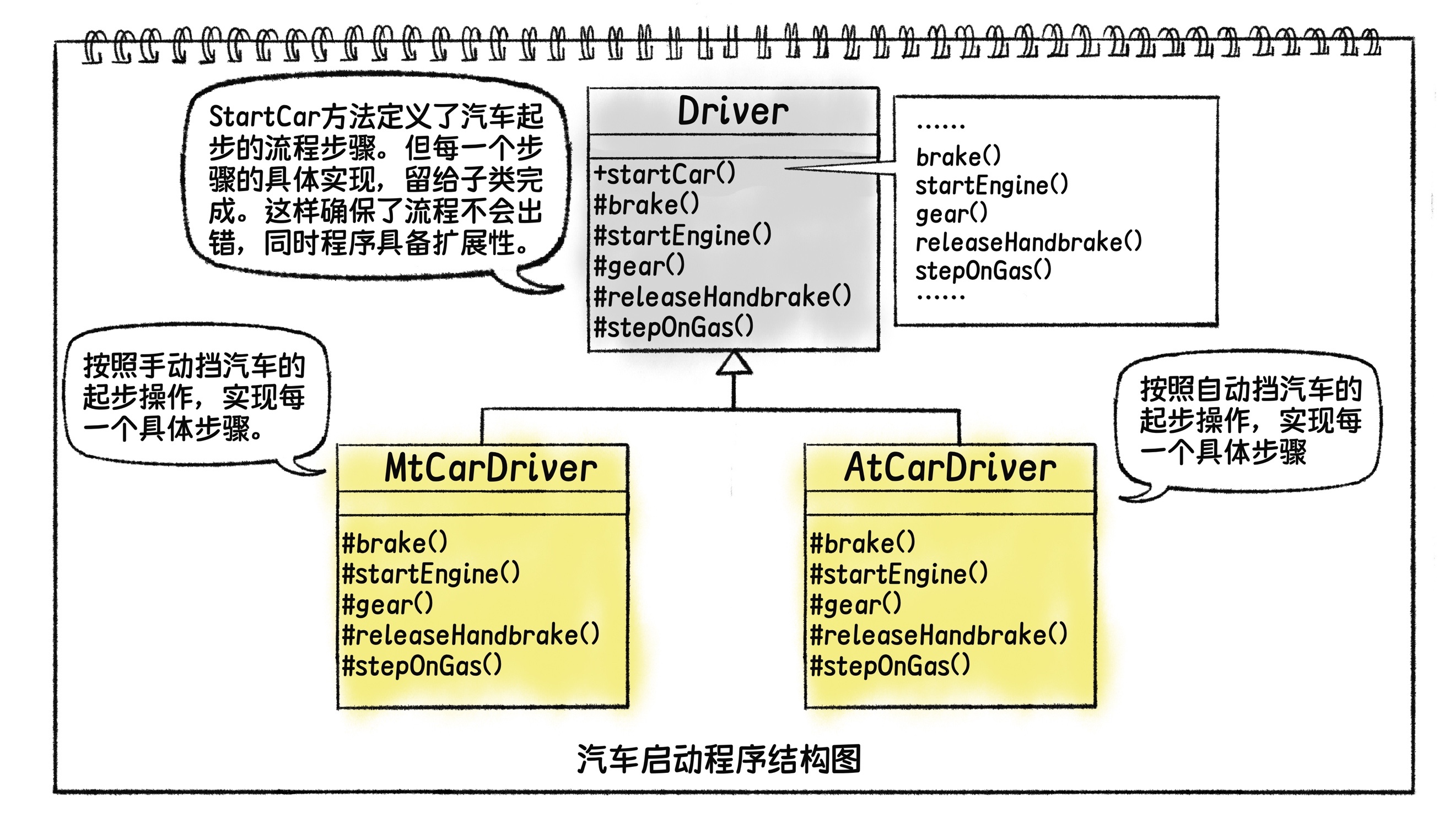
熊小猫：你拷贝AtCarDriver的startCar方法代码，其实是为了复用自动挡汽车起步的流程，然后按照手动挡的操作修改细节。以拷贝的方式来“复用”代码是错误的。我们需要在程序的设计上做文章。

咱们先梳理一下汽车起步的流程。

1. 点火前准备
2. 汽车点火
3. 挂档
4. 松手刹
5. 加油出发

无论是手动档还是自动档汽车，汽车起步都是这5个步骤。你可以将每一个步骤定义为一个方法。在父类Driver的startCar方法中，按照顺序调用这5个方法。Driver子类通过继承获得startCar方法，一定会按照父类中定义的步骤顺序执行汽车起步。每个步骤的方法由各个子类自己实现。

兔小白：按照你的思路，我先画出程序结构图，你看看对不对。



熊小猫：没有问题，可以按照结构图去开发。

15分钟后，兔小白改好了程序。

兔小白：首先将Driver接口改为抽象类。按汽车起步流程的步骤，定义相应的方法。 在startCar方法中，依次调用这几个步骤的方法。汽车起步操作的流程被固化在startCar方法中。

public abstract class Driver {  
    public void startCar(){  
        prepareToStart();  
        startEngine();  
        gear();  
        releaseHandbrake();  
        stepOnGas();  
   }  
​  
    protected abstract void prepareToStart();  
    protected abstract void startEngine();  
    protected abstract void gear();  
    protected abstract void releaseHandbrake();  
    protected abstract void stepOnGas();  
}

兔小白：AtCarDriver和MtCarDriver类继承Driver抽象类，这意味着它们复用了Driver的startCar方法，也就是复用了汽车起步的固定流程。这两个子类可以根据不同的汽车种类去实现每个步骤的具体操作。

AtCarDriver的实现如下。

public class AtCarDriver extends Driver {  
   
    protected void prepareToStart() {  
        System.out.println("踩刹车");  
   }  
​  
    protected void startEngine() {  
        System.out.println("一键启动");  
   }  
​  
    protected void gear() {  
        System.out.println("挂D档");  
   }  
​  
    protected void releaseHandbrake() {  
        System.out.println("放开电子手刹");  
   }  
​  
    protected void stepOnGas() {  
        System.out.println("松刹车");  
        System.out.println("踩油门");  
   }  
}

MtCarDriver的实现如下。

public class MtCarDriver extends Driver {  
​  
    protected void prepareToStart() {  
        System.out.println("踩刹车");  
   System.out.println("踩离合");  
   }  
​  
    protected void startEngine() {  
        System.out.println("插入钥匙");  
        System.out.println("转动钥匙打火");  
   }  
​  
    protected void gear() {  
        System.out.println("挂1档");  
   }  
​  
    protected void releaseHandbrake() {  
        System.out.println("放开手刹");  
   }  
​   
    protected void stepOnGas() {  
        System.out.println("松刹车");  
        System.out.println("抬起离合");  
        System.out.println("踩油门");  
   }  
}

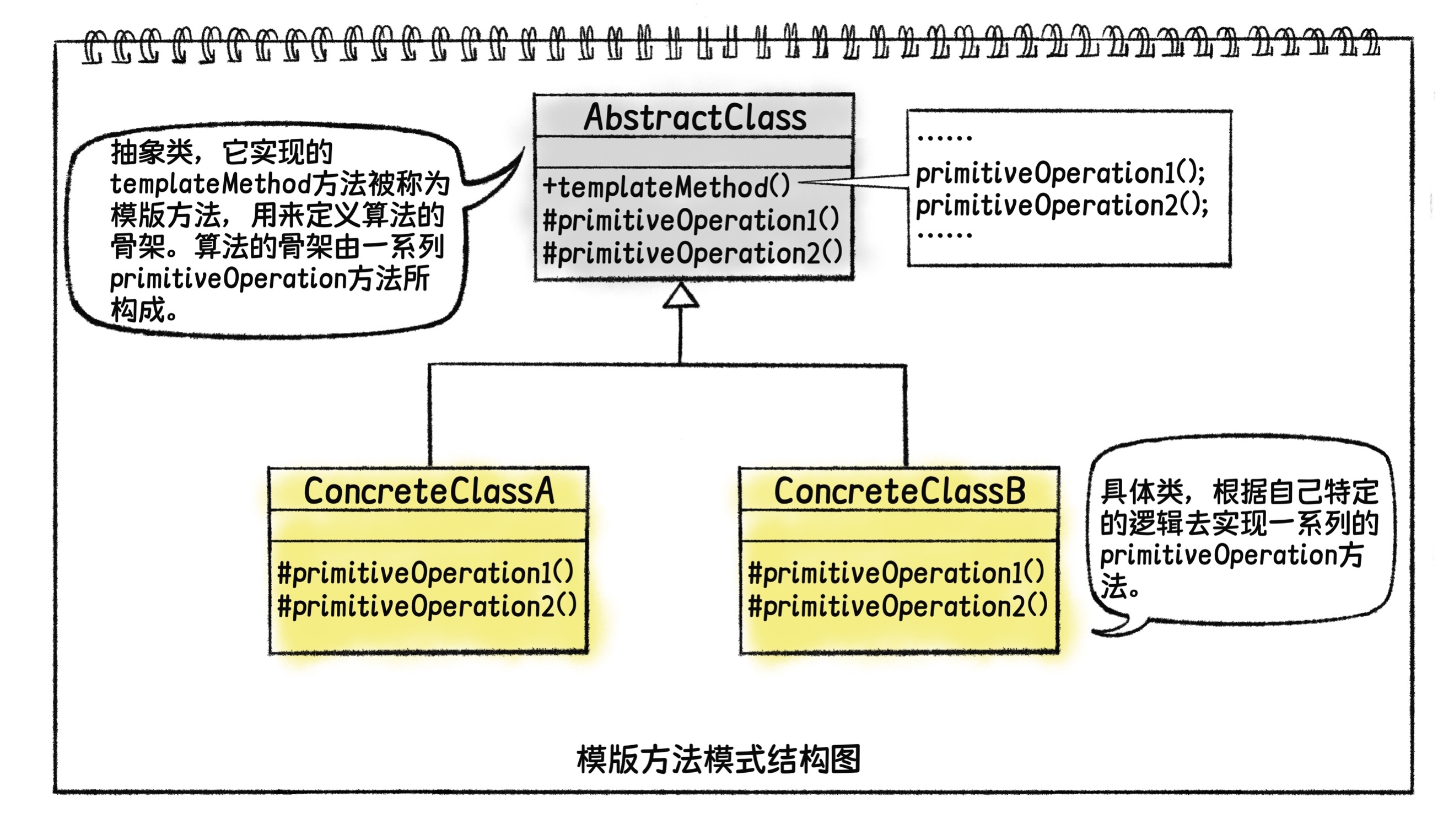
两个子类对每一个步骤的实现都有不同，但是起步的流程保持一致，不会漏掉步骤。

熊小猫：父类定义好不变的起步流程，任何子类通过继承获得startCar的能力，都会执行这个流程。子类只需要关心每个步骤的具体实现，也就是变化的部分。

父类中定义的不会变化的流程就是模版，子类负责实现流程中每一个步骤的方法。这就是模版方法模式。

## 25.4 模版方法模式适用场景

熊小猫：我们来看看模版方法模式的结构图。



模版方法模式的结构非常简单，只有一个抽象类和若干个实现类。我们先看AbstractClass，它好比练习中的Driver类。它实现的templateMethod方法被称为模版方法，用来定义算法的骨架。算法的骨架由一系列primitiveOperation方法所构成。

ConcreteClass好比练习中的AtCarDriver和MtCarDriver。它复用AbstractClass的templateMethod实现，确保算法骨架的稳定。ConcreteClass会根据自己特定的逻辑去实现一系列的primitiveOperation方法。

兔小白：模版方法不但保证了算法骨架的稳定，还做到了算法骨架中的每个步骤都是灵活可扩展的。

熊小猫：我们分析的更深入一点，模版方法分离了算法中不变和变化的部分。算法高层次的步骤定义和执行顺序固定不变，在父类中实现。每个步骤的具体实现可以有多种方式，交给各个子类实现。

模版方法模式的适用场景具有如下特点。

1. 算法比较复杂。如果算法简单，那么算法的骨架也会比较简单，这样便失去了复用算法骨架的意义。
2. 算法的不同实现，呈现出同样的步骤流程。如果算法的不同实现大相径庭，无法抽象出同一套算法骨架，那么也就无法使用模版方法。使用模版方法模式，一定是建立在算法骨架一致的基础上。

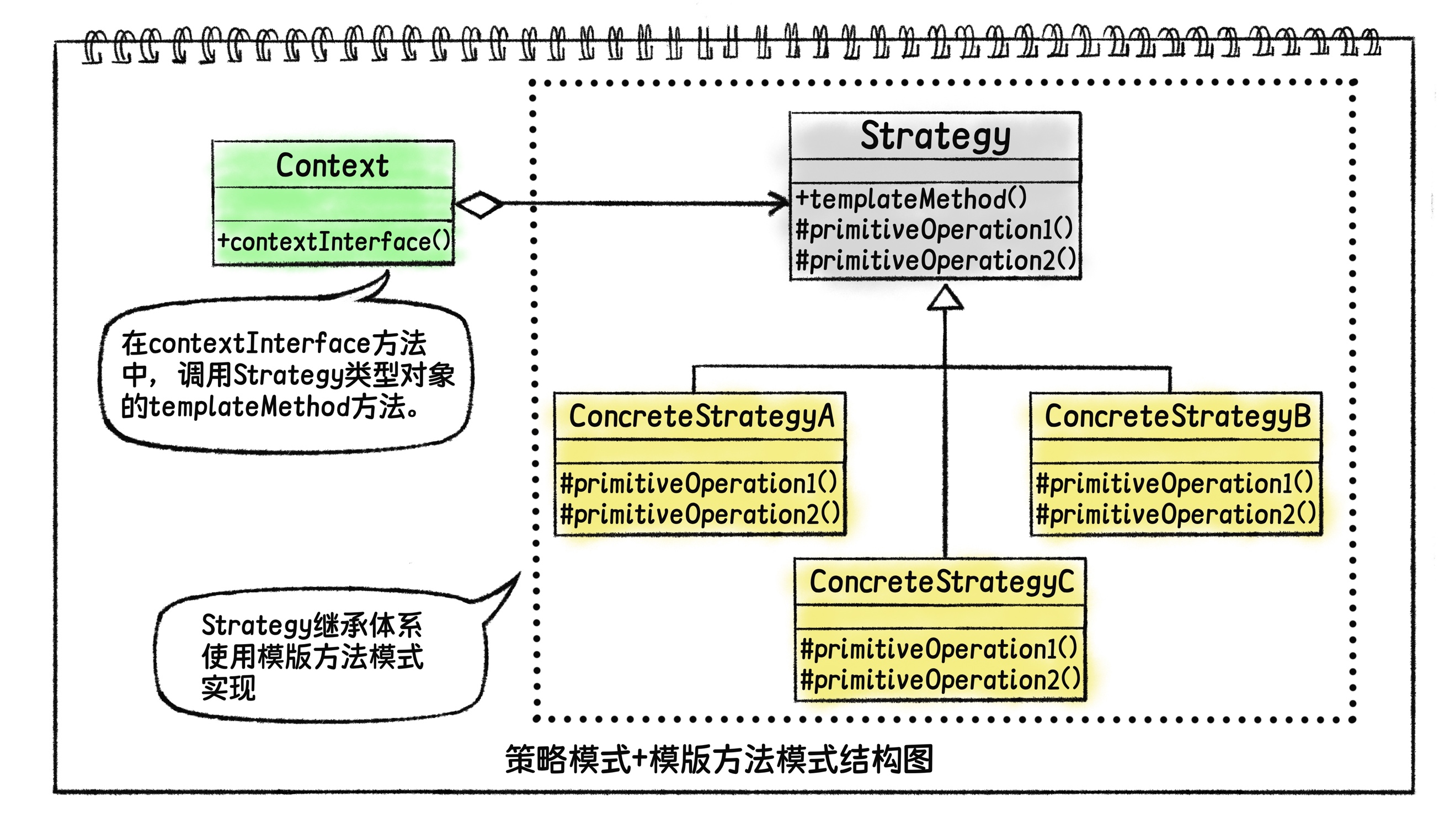
## 25.5 模版方法模式与策略模式的对比和结合

兔小白：策略模式也可以解决算法的扩展性问题。这两种设计模式各有什么侧重点呢？

熊小猫：两者的相同之处在于，都解决了同一个算法不同实现的扩展问题。但是策略模式的侧重点在于算法的灵活切换。模版方法模式的侧重点是复杂算法的骨架复用。

兔小白：两者的侧重点完全不同，我以后在使用的时候还是需要多思考。

熊小猫：你要思考的并不仅仅是该用哪一种设计模式。有些时候，这两种模式可以结合起来使用，只需要将策略模式中的Strategy继承体系替换成模版方法模式的结构。我们看下面的结构图。



这样的程序设计，既可以复用算法骨架，又可以灵活切换算法。

工作中的大多数场景都比较复杂，需要多种设计模式配合使用。我们需要从问题出发，使用一种设计模式解决一个问题。如果还有未解决的问题，继续思考还可以用哪种设计模式来解决。最后，再去思考如何将多种设计模式结合起来。

兔小白：虽然没有万能的设计模式，但是将设计模式组合起来却有无限种可能呀！