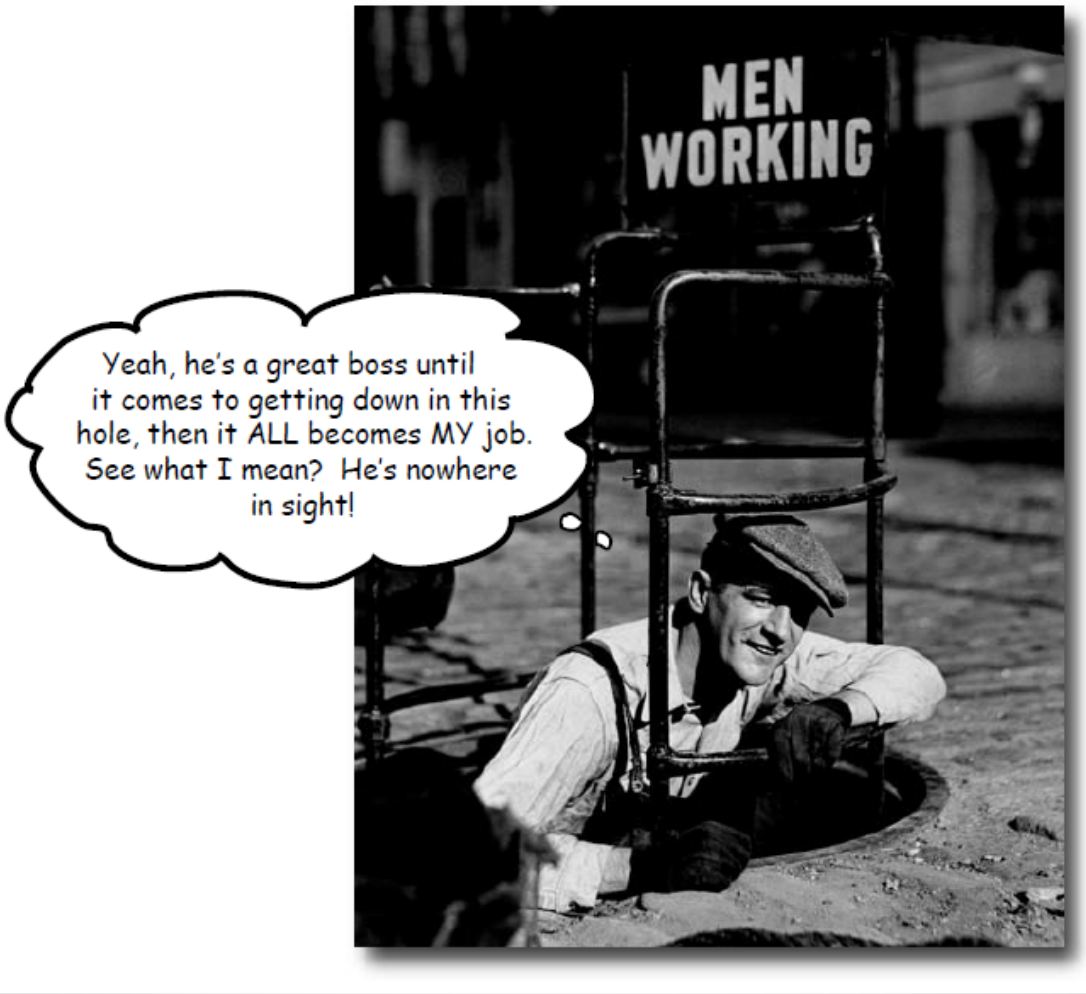
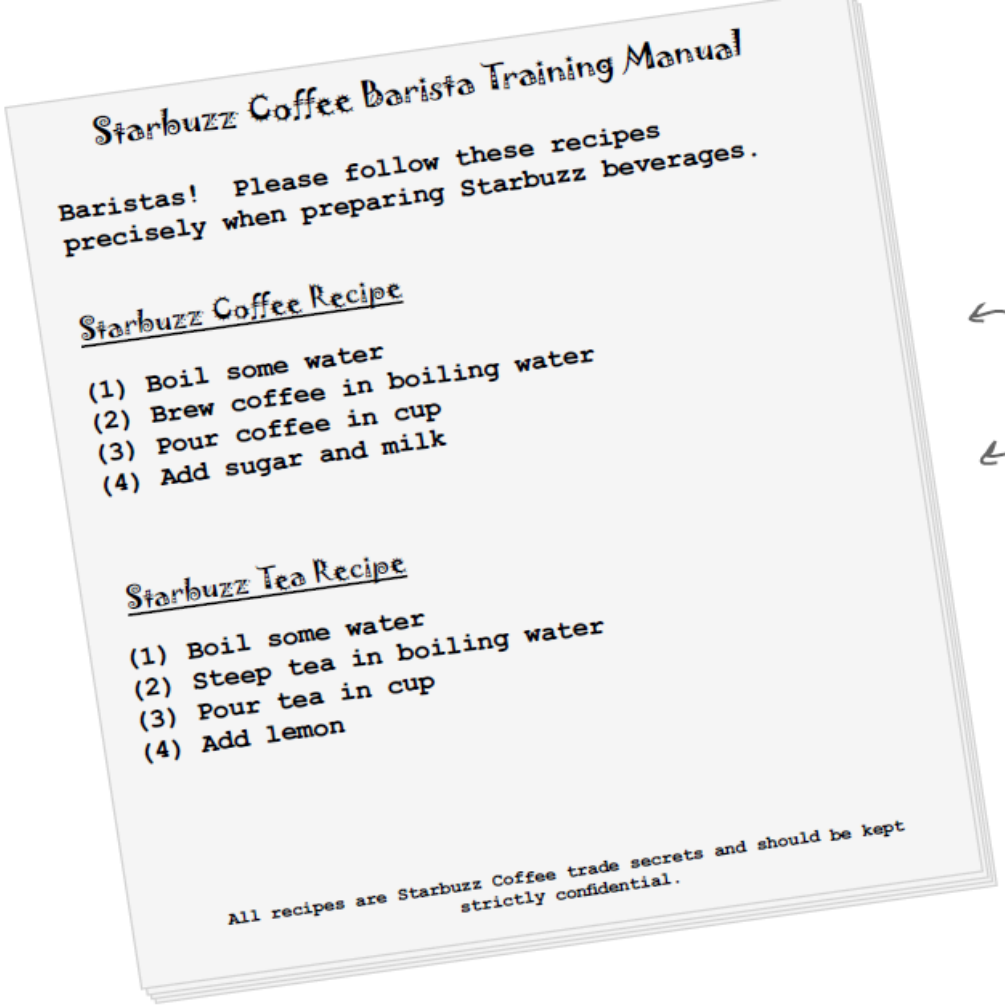
Encapsulating Algorithms

ELSYS 2014/2015
Vasil Kostov
Georgi Yosifov


MEN
WORKING
Yeah, he's a great boss until it comes to getting down in this hole, then it ALL becomes MY job. See what I mean? He's nowhere in sight!



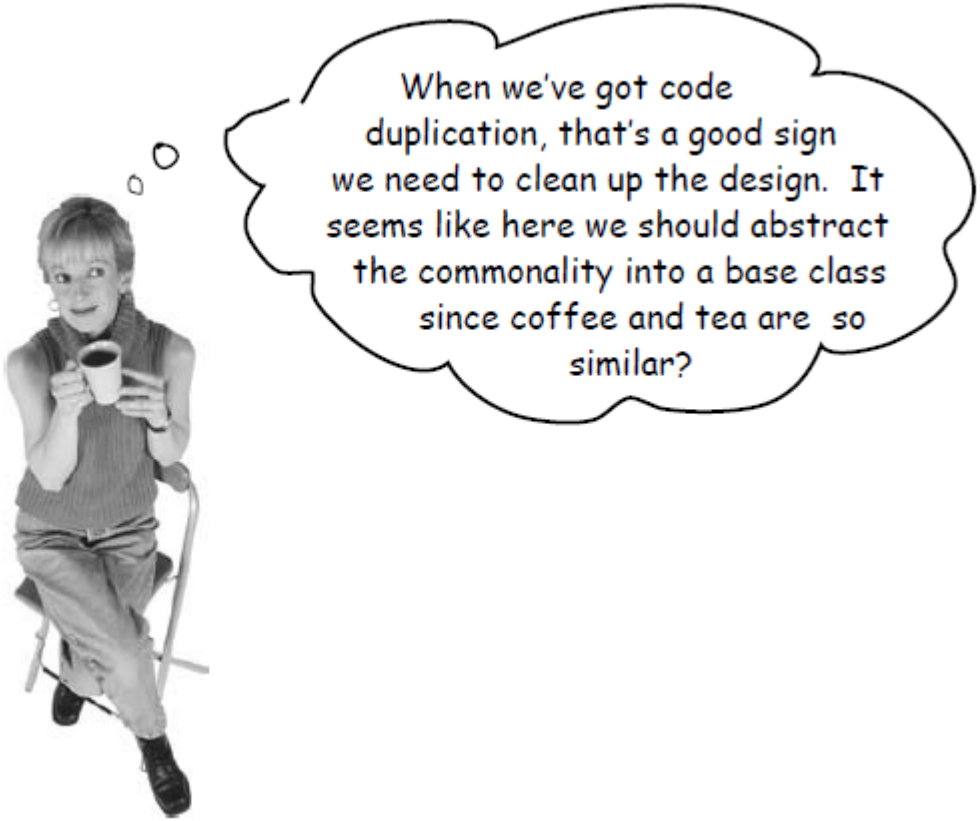
The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Let's play "coding barista" and write some code for creating coffee and tea.

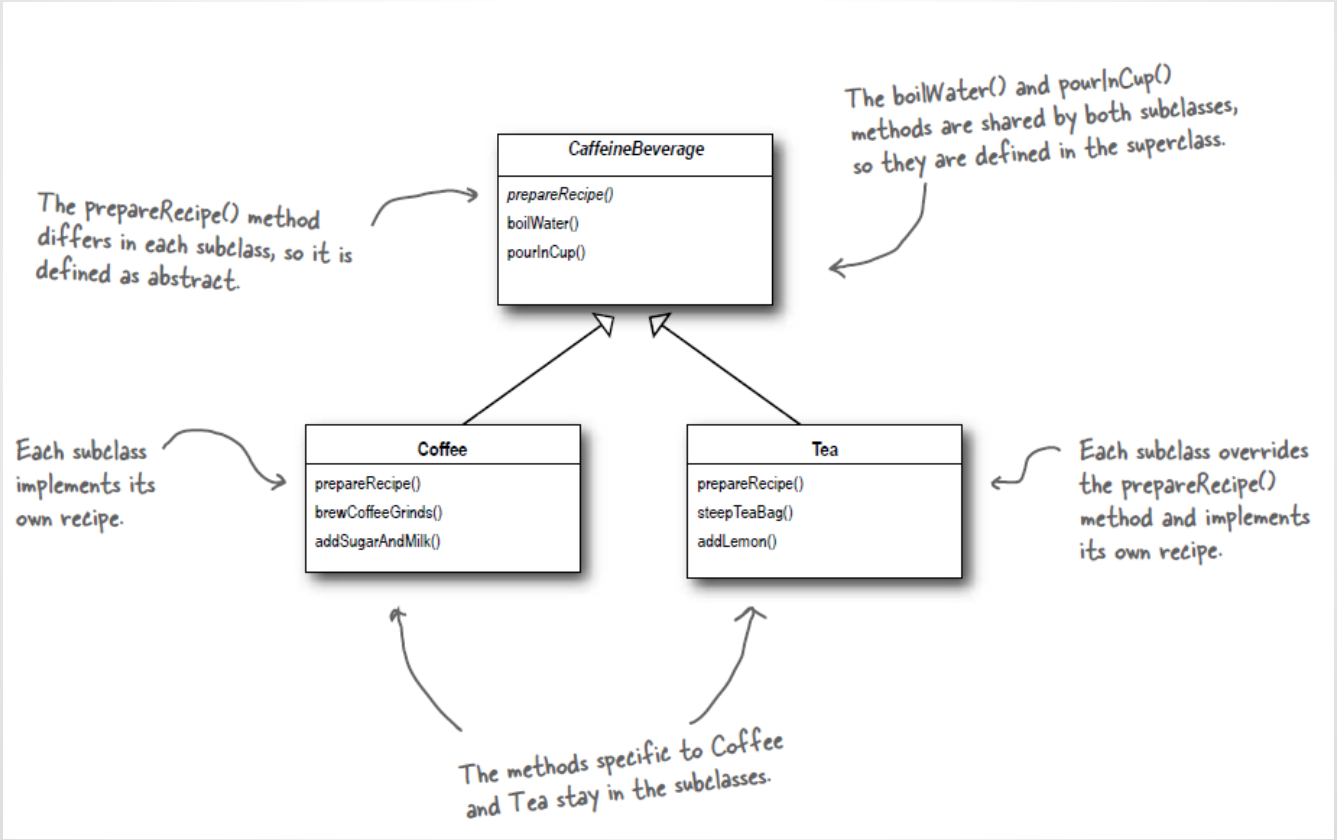
```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

and now the Tea...

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");  igh filter");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

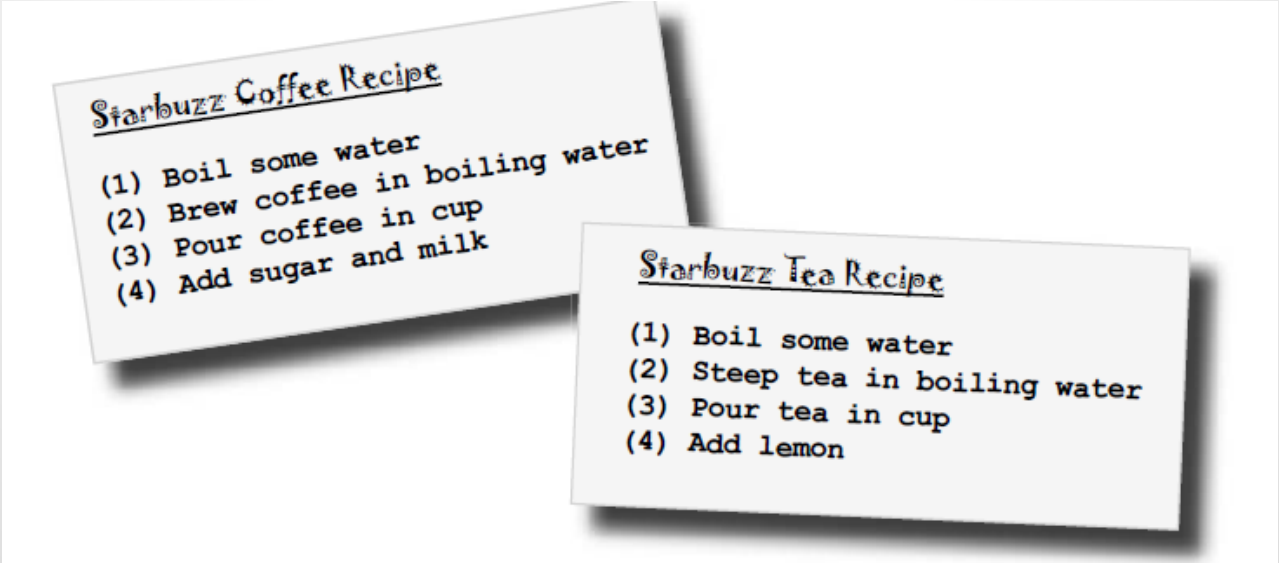

When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

Sir, may I abstract your Coffee, Tea?

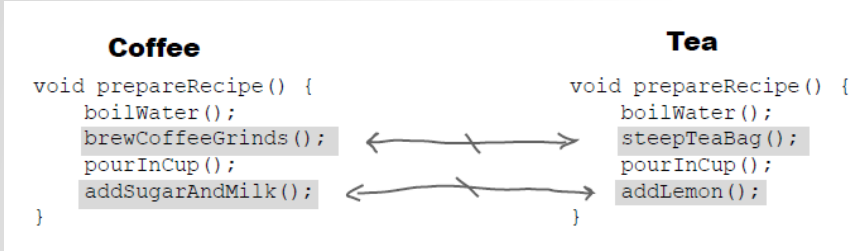


Taking the design further...

- So what else do Coffee and Tea have in common? Let's start with the recipes.



Abstracting prepareRecipe()



- The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

As in our design, Tea and Coffee now extend CaffeineBeverage.

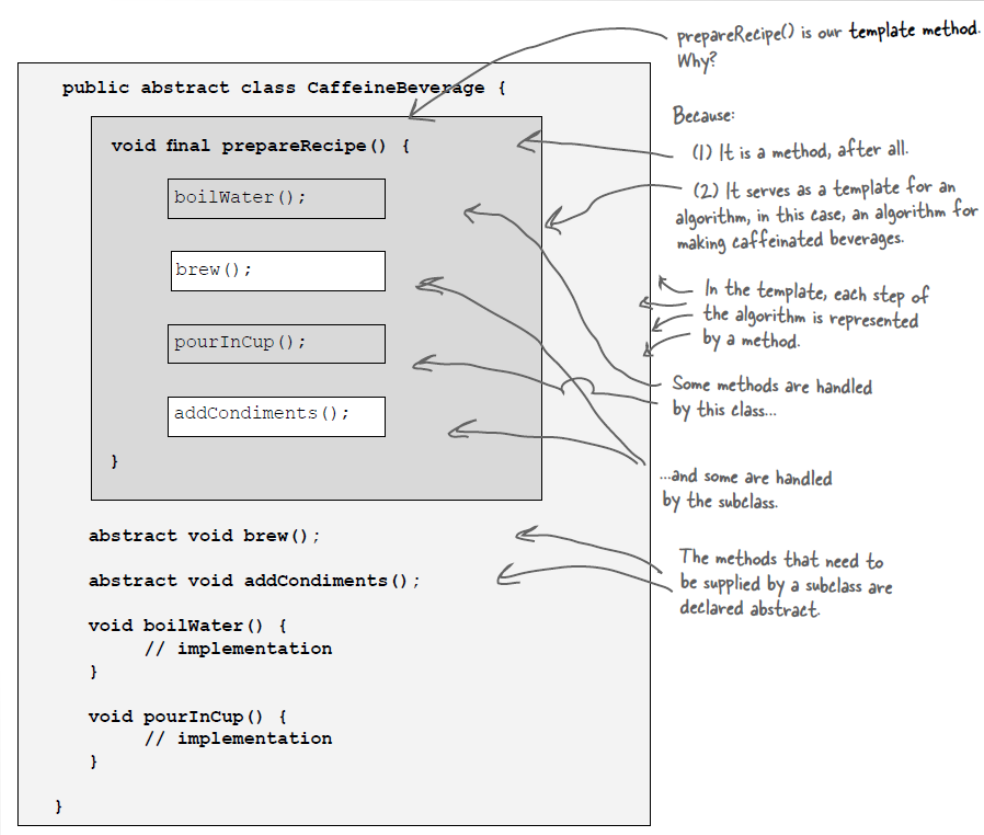
Tea needs to define brew() and addCondiments() – the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

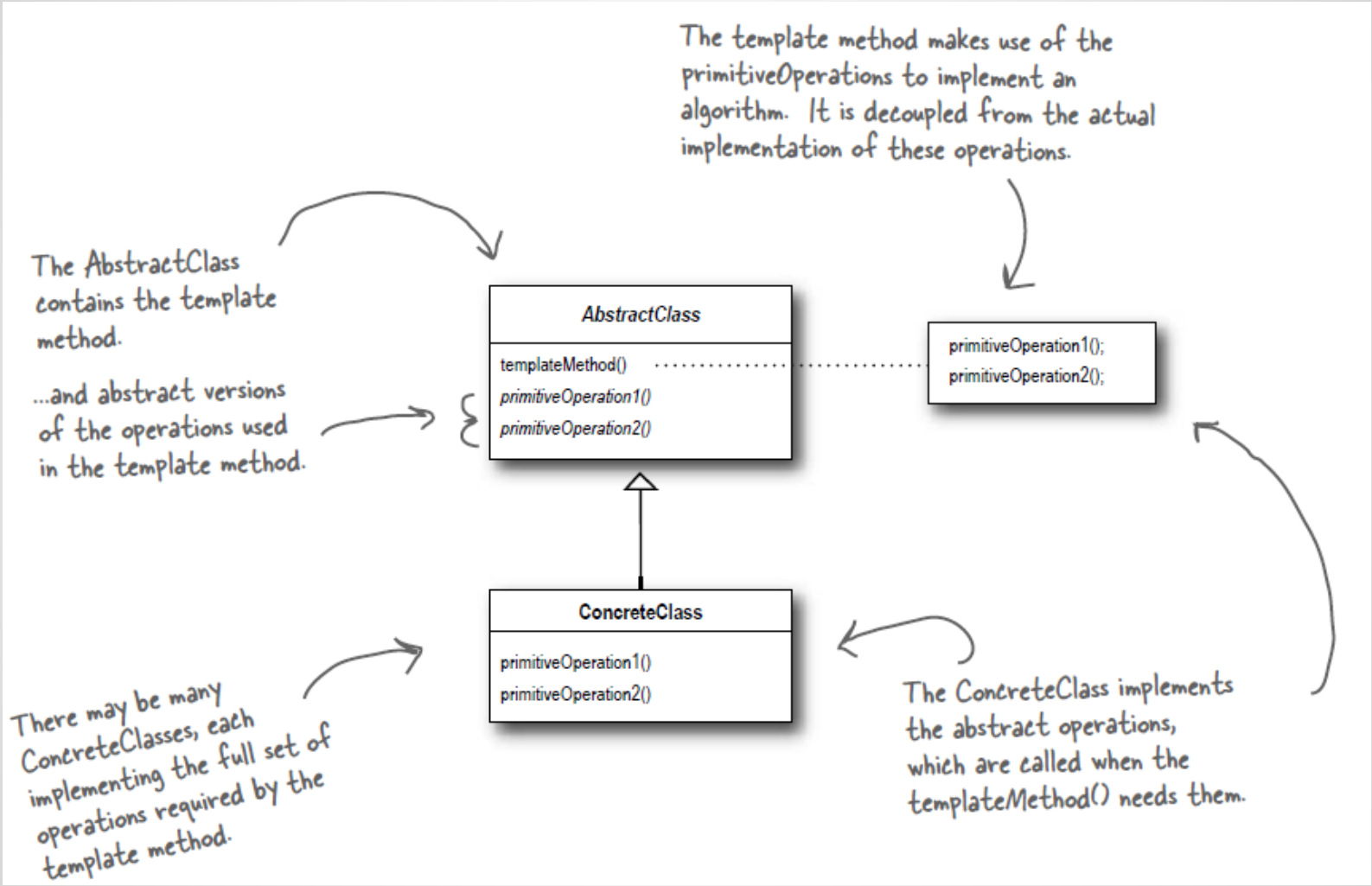
Meet the Template Method

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.



Template Method Pattern defined

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.


The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.
The AbstractClass contains the template method.
...and abstract versions of the operations used in the template method.
AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
primitiveOperation1();
primitiveOperation2();
ConcreteClass
primitiveOperation1()
primitiveOperation2()
There may be many ConcreteClasses, each implementing the full set of operations required by the template method.
The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you."