CS525 Advanced Software Development

Lesson 8 – The Template Method Pattern

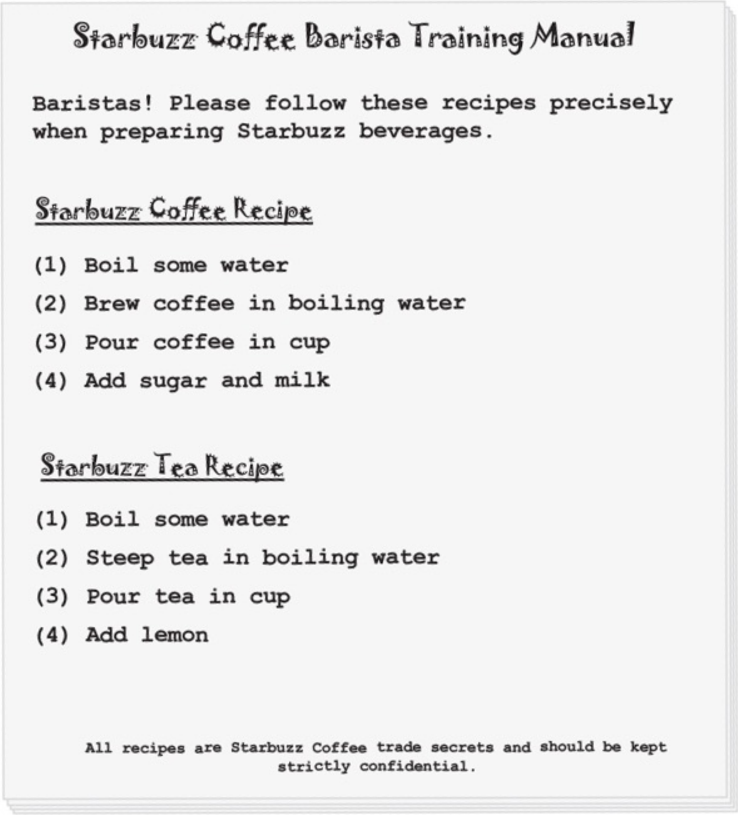
Design Patterns
Elements of Reusable Object-Oriented Software

Payman Salek, M.S.
March 2022

The Inspiration

We’re going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We’re even going to learn about a design principle inspired by Hollywood.

Setting the stage (Starbuzz Coffee)



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

(1) Boil some water

(2) Brew coffee in boiling water

(3) Pour coffee in cup

(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water

(2) Steep tea in boiling water

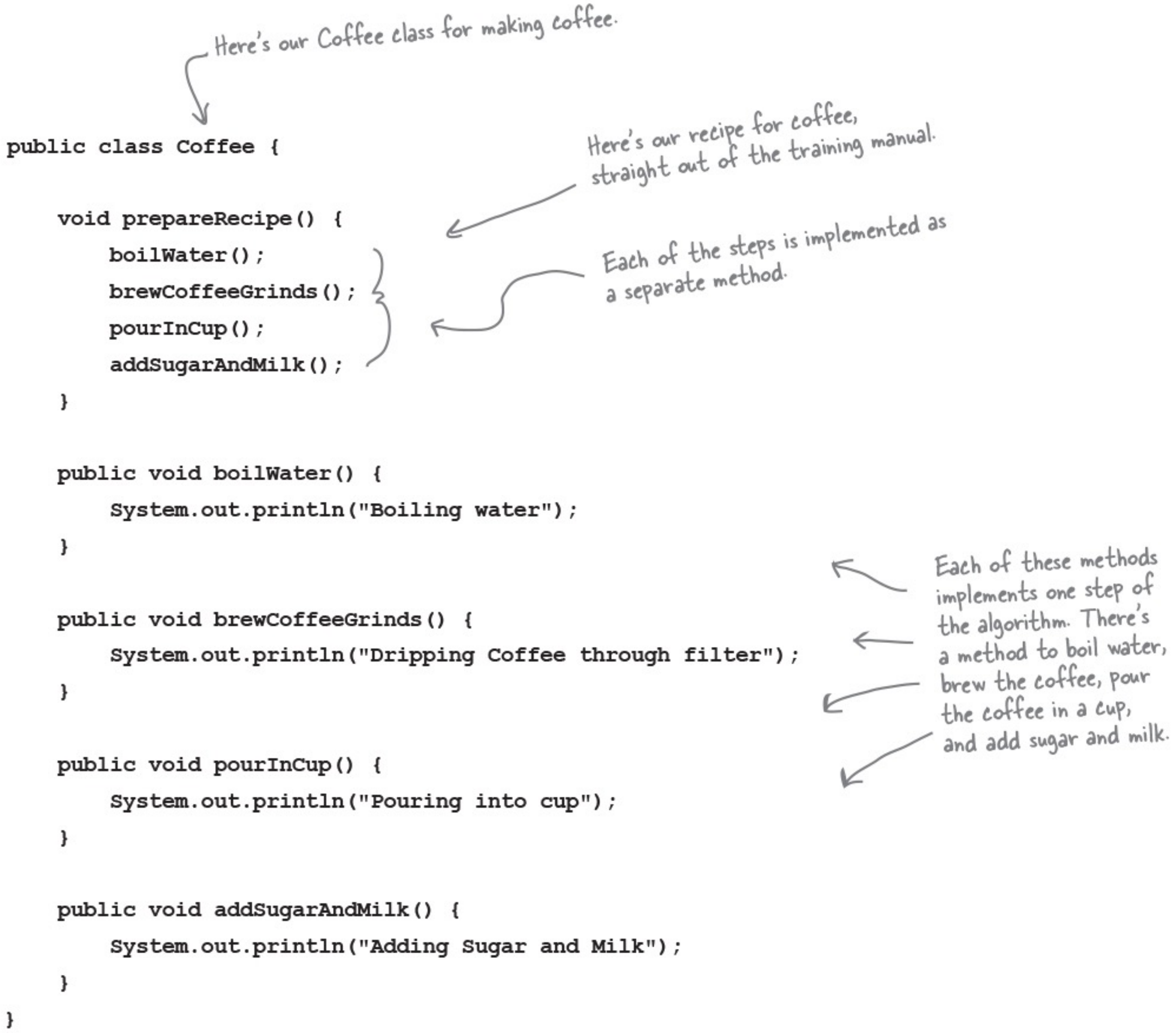
(3) Pour tea in cup

(4) Add lemon

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

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

The Coffee Class



```
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");
    }
}
```

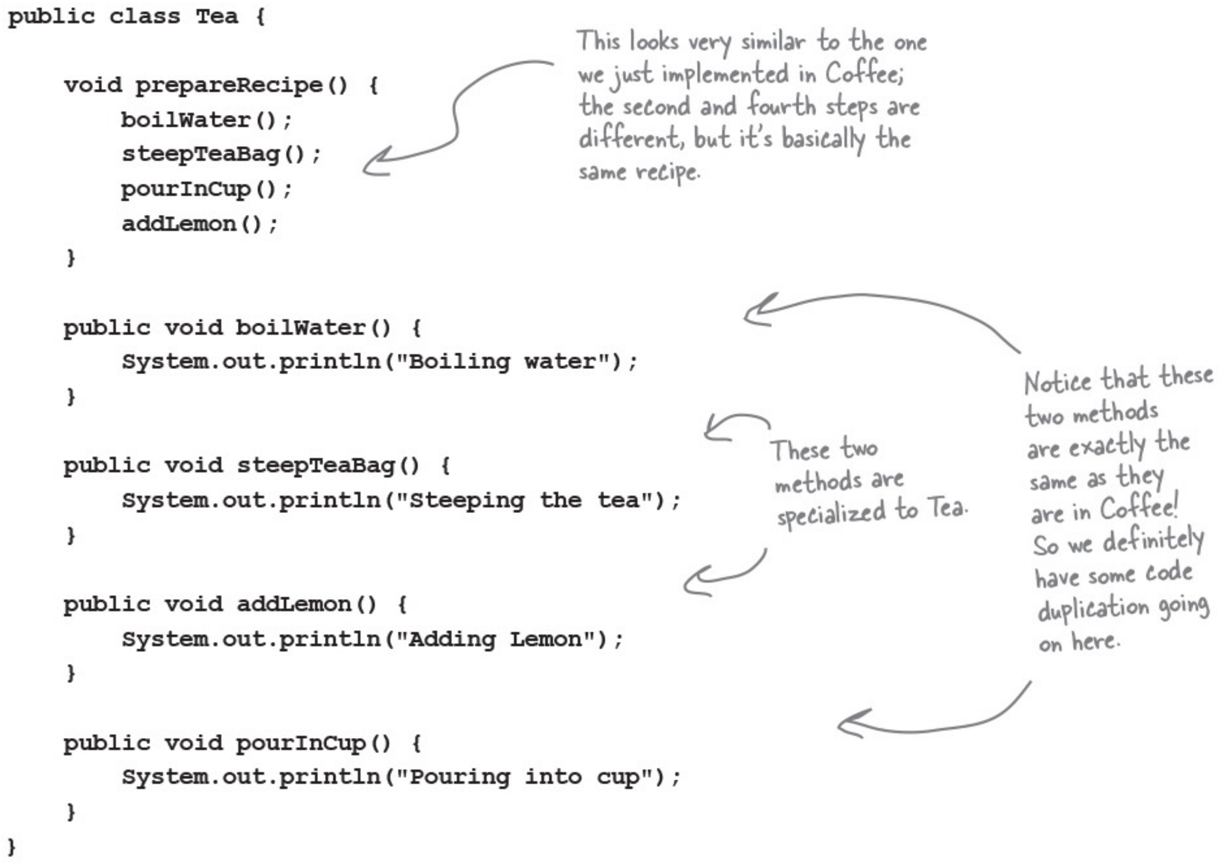
Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

The Tea Class



```
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");
    }

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

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

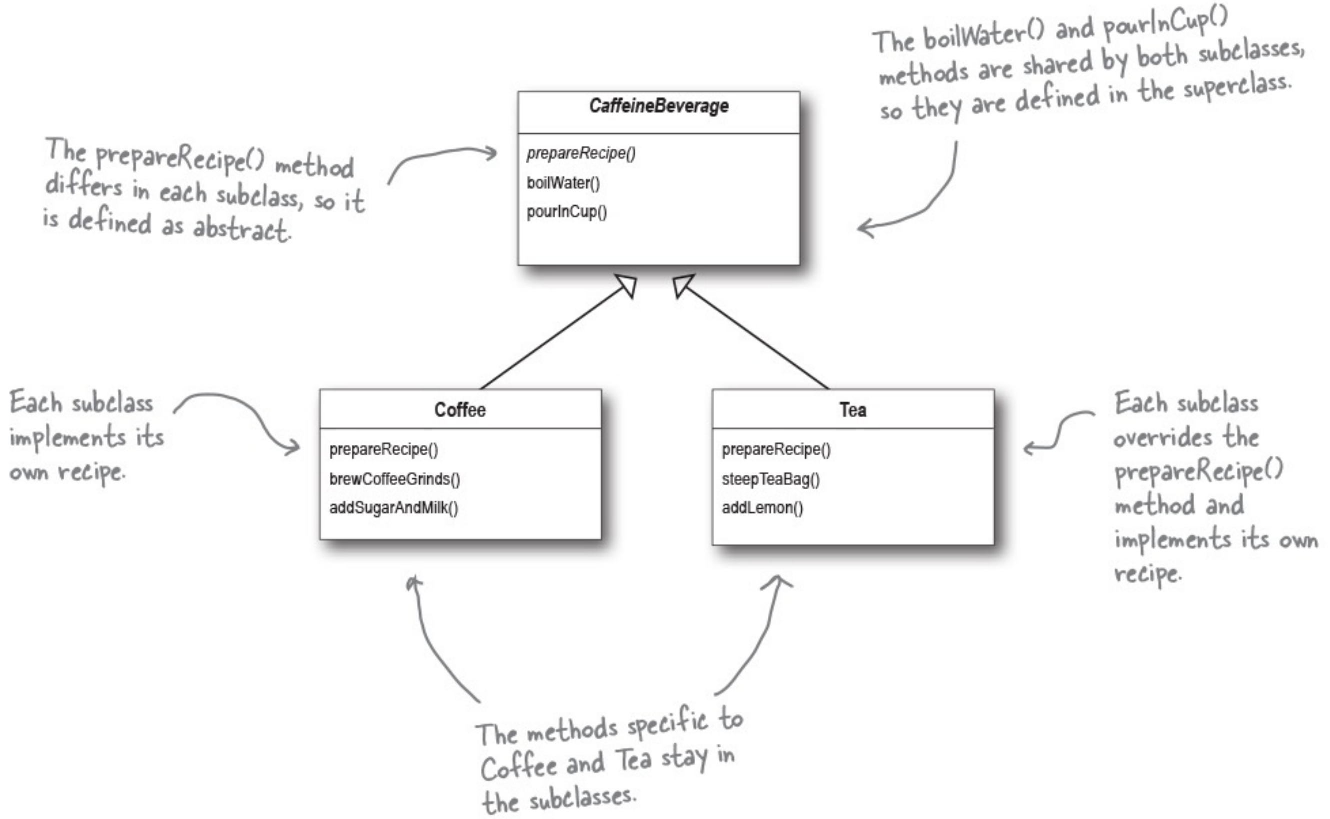
This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

3/2/22

Let’s Add Some Abstraction



CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()

Coffee

prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()

Tea

prepareRecipe()
steepTeaBag()
addLemon()

The prepareRecipe() method differs in each subclass, so it is defined as abstract.

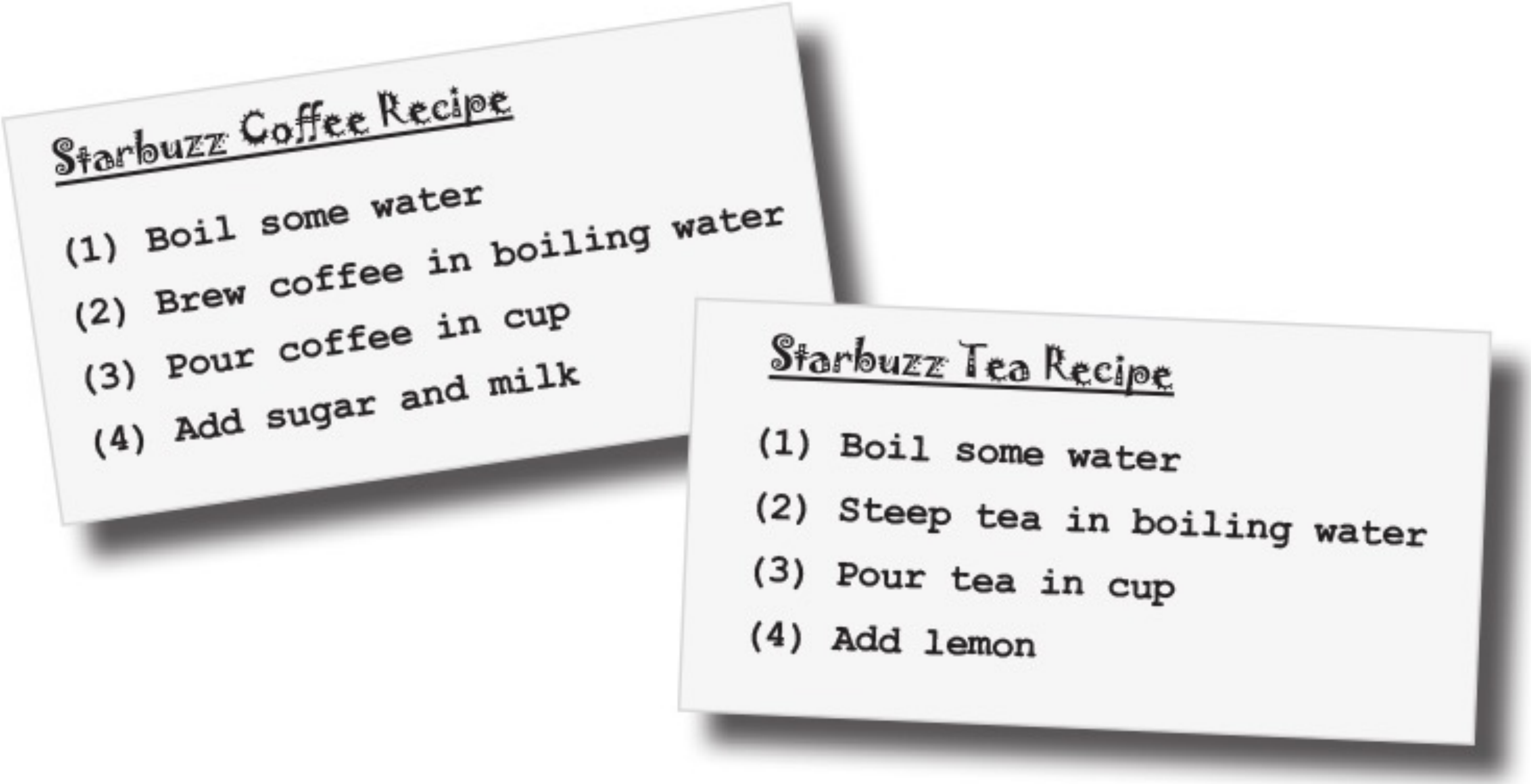
The boilWater() and pourInCup() methods are shared by both subclasses, so they are defined in the superclass.

Each subclass implements its own recipe.

Each subclass overrides the prepareRecipe() method and implements its own recipe.

The methods specific to Coffee and Tea stay in the subclasses.

What’s else is common?



Starbuzz Coffee Recipe

(1) Boil some water

(2) Brew coffee in boiling water

(3) Pour coffee in cup

(4) Add sugar and milk

Starbuzz Tea Recipe

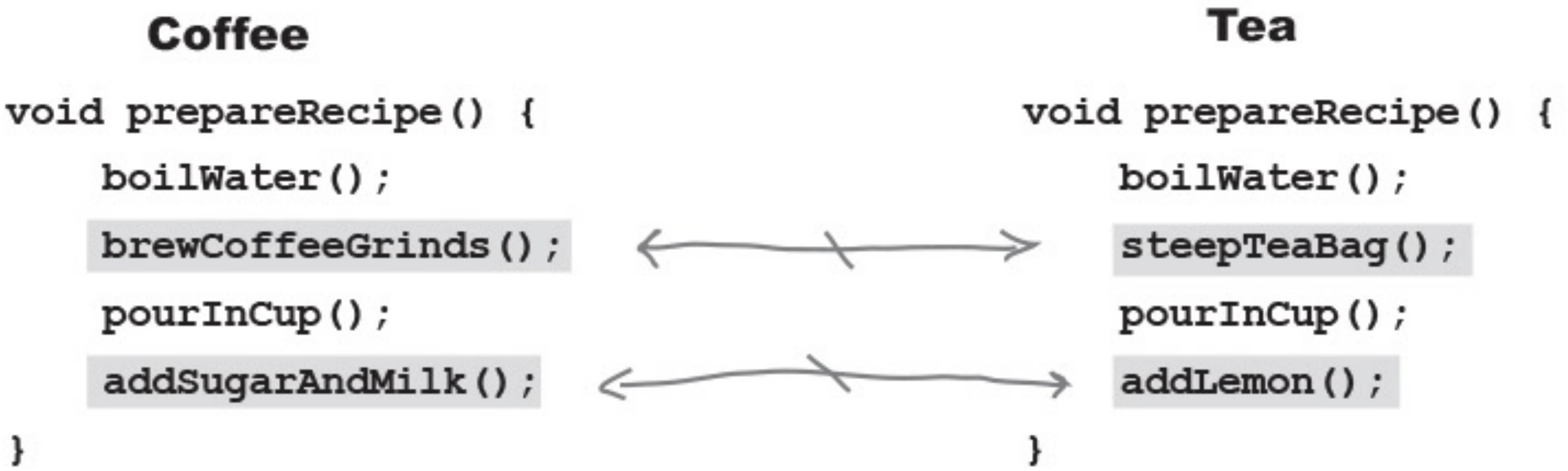
(1) Boil some water

(2) Steep tea in boiling water

(3) Pour tea in cup

(4) Add lemon

What’s else is common?



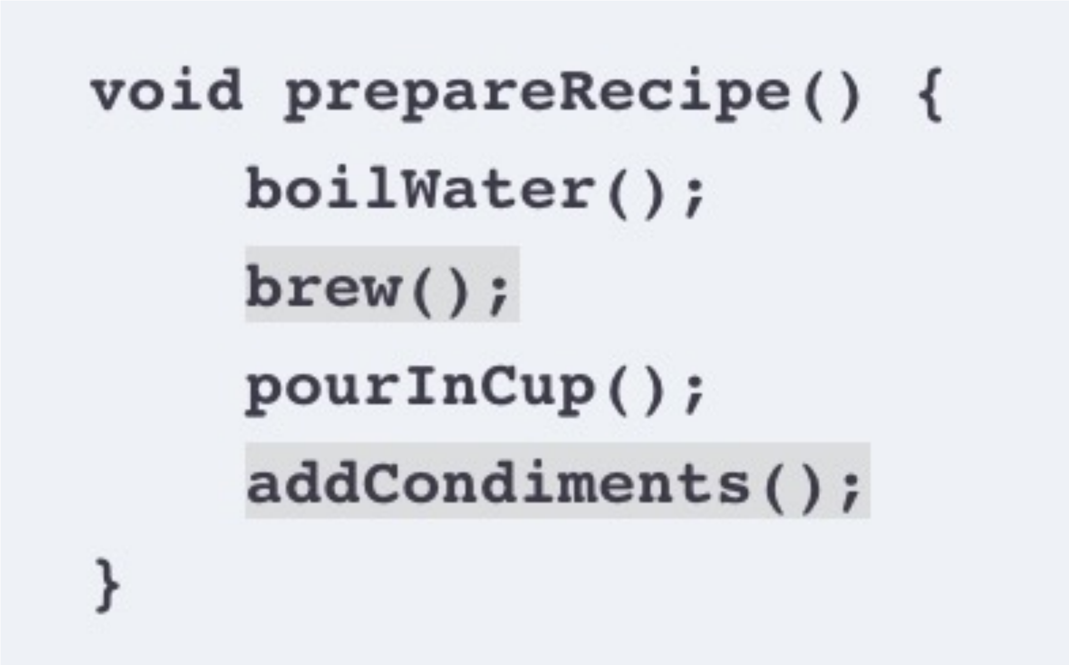
Coffee

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

Tea

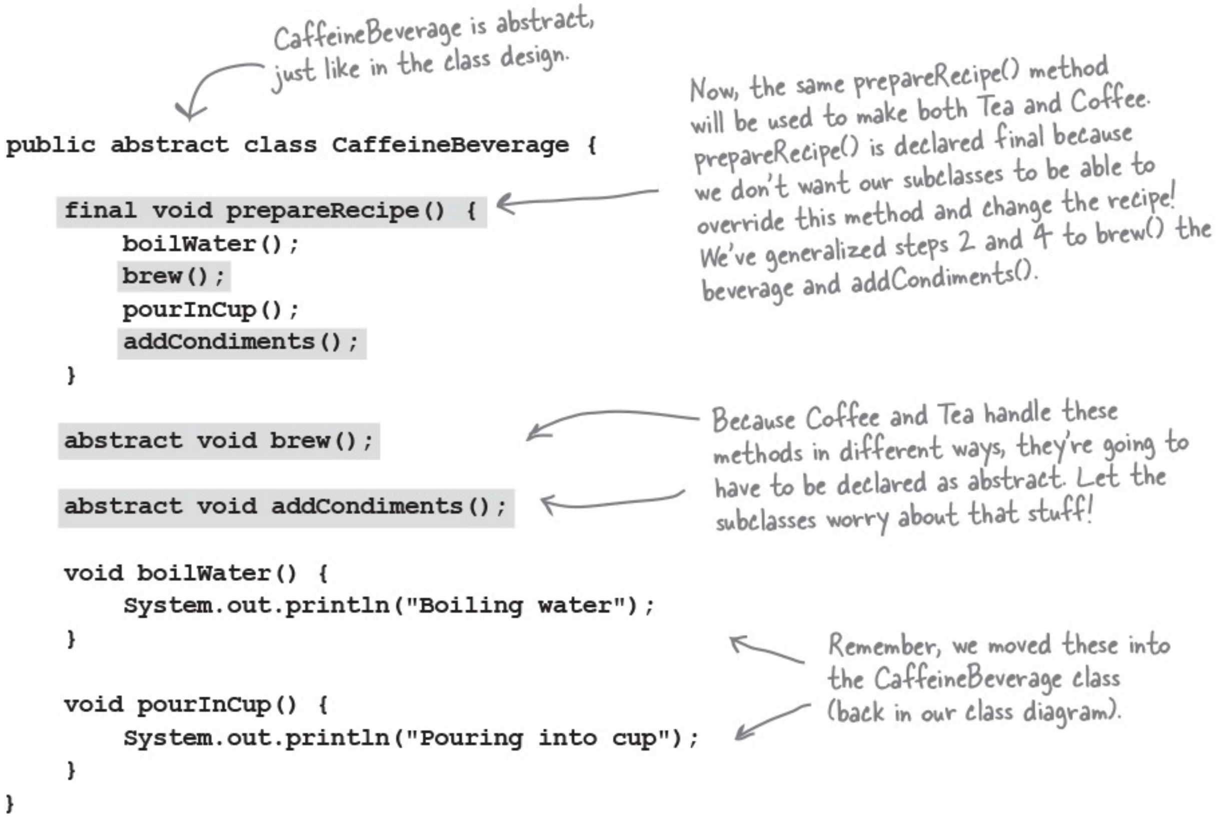
```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Even More Abstraction



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

The non-Changing Part



CaffeineBeverage is abstract, just like in the class design.

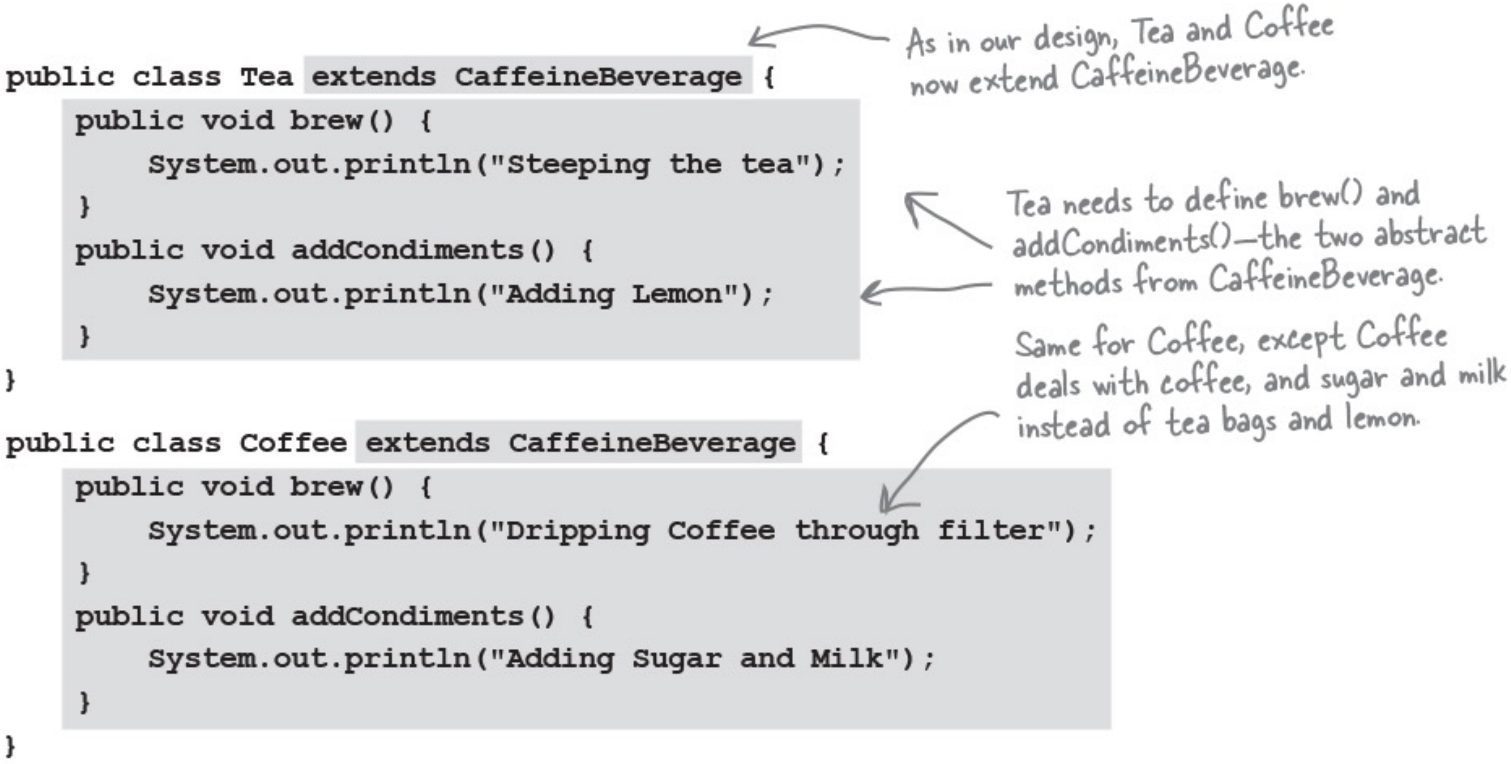
```
public abstract class CaffeineBeverage {

    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");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

The Changing Part



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

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

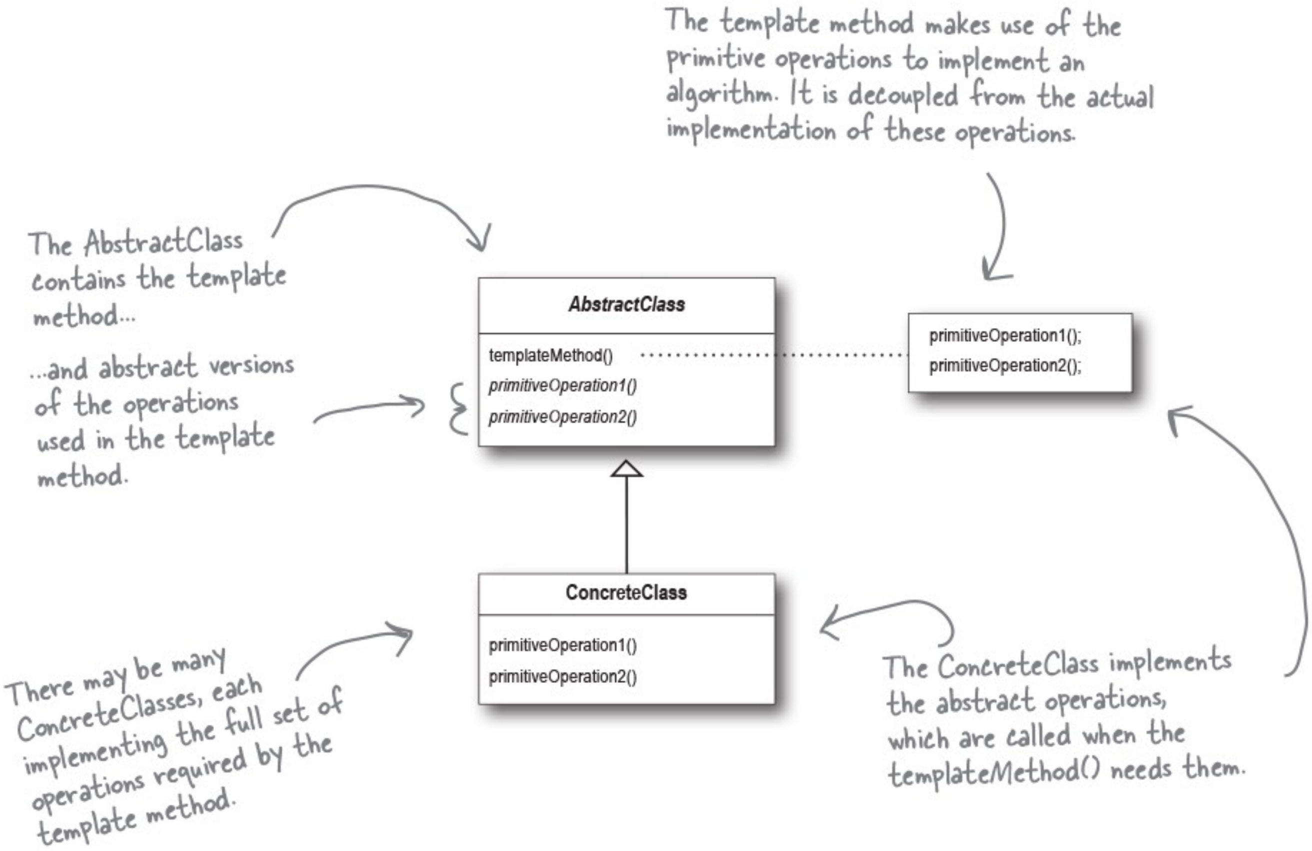
Tea needs to define brew() and addCondiments()—the two abstract methods from CaffeineBeverage.

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

The Template Method Pattern

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

Class Diagram for the Pattern



The template method makes use of the primitive operations to implement an algorithm. It is decoupled from the actual implementation of these operations.

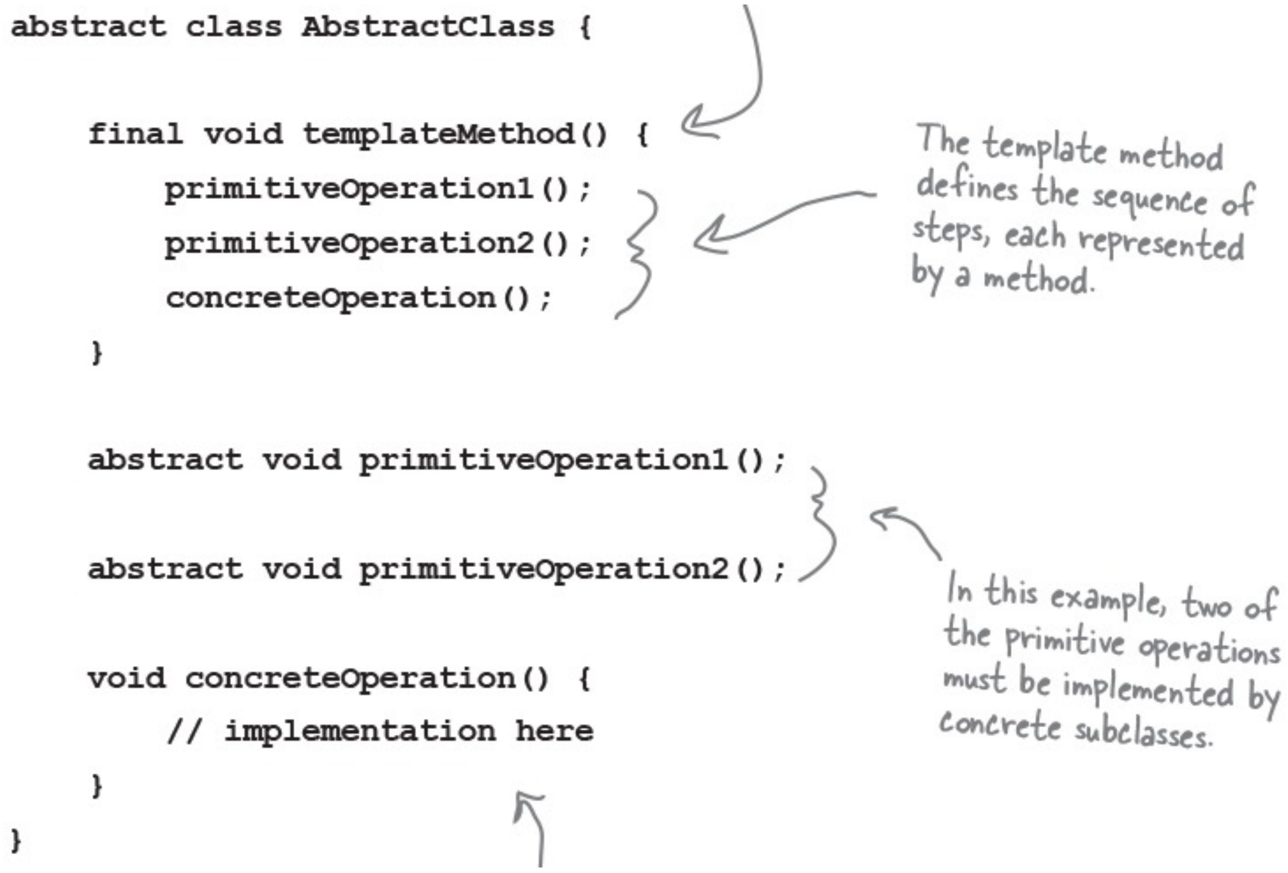
The AbstractClass contains the template method...

...and abstract versions of the operations used in the template method.

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.

Classical Implementation

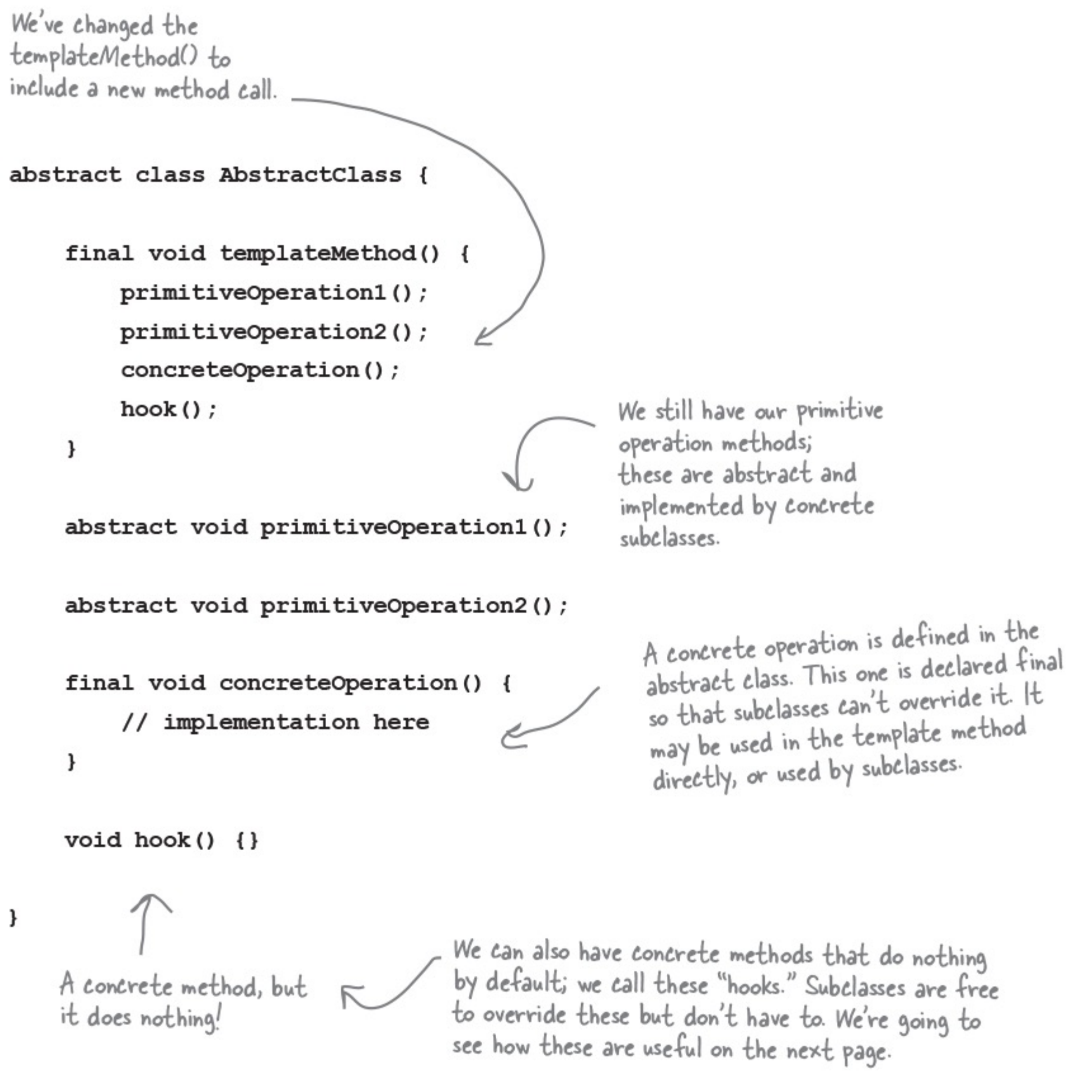


```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

Classical Implementation + Hook Method



We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive operation methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

The Hollywood Principle

Don’t call us, we’ll call you!

Template Method in Practice

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

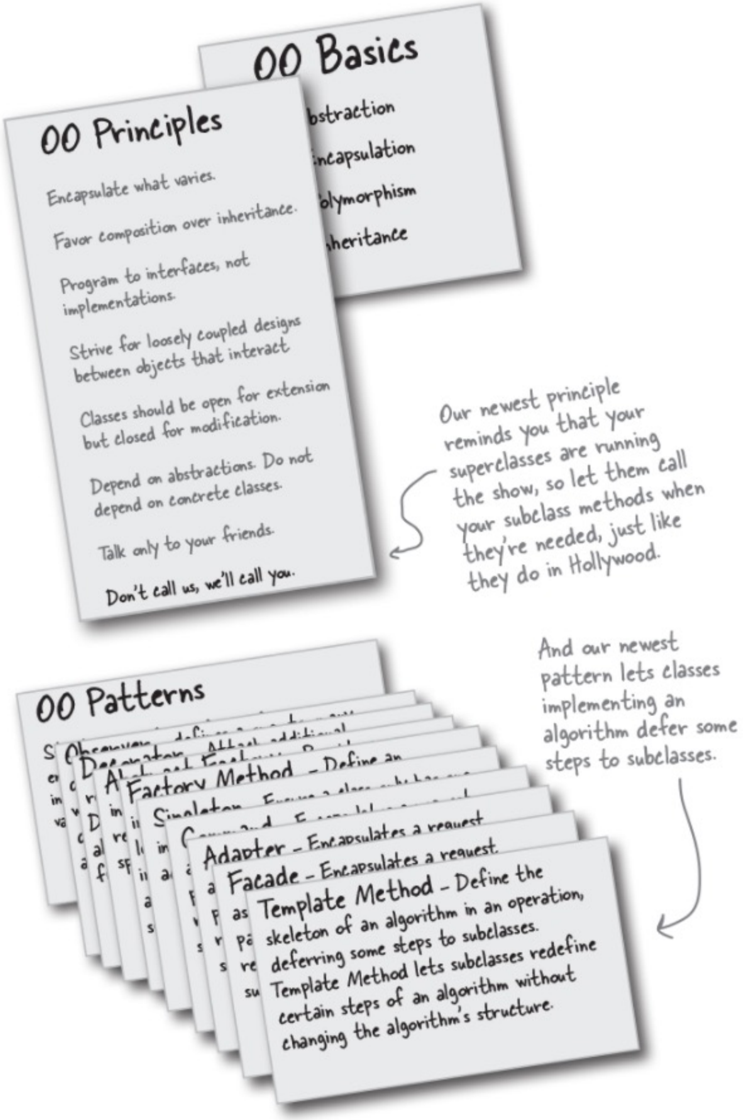
Summary

- A template method defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method's abstract class may define concrete methods, abstract methods, and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.

Summary - continued

- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- You’ll see lots of uses of the Template Method Pattern in real-world code, but (as with any pattern) don’t expect it all to be designed “by the book.”
- The Strategy and Template Method Patterns both encapsulate algorithms, the first by composition and the other by inheritance.
- Factory Method is a specialization of Template Method.

The Template Method Pattern



OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Talk only to your friends.

Don't call us, we'll call you.

Our newest principle reminds you that your superclasses are running the show, so let them call your subclass methods when they're needed, just like they do in Hollywood.

OO Patterns
Observer
Decorator
Abstract Factory
Factory Method – Define an
Singleton
Command – Encapsulates a request
Adapter – Encapsulates a request
Facade –

Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.