8 the Template Method Pattern

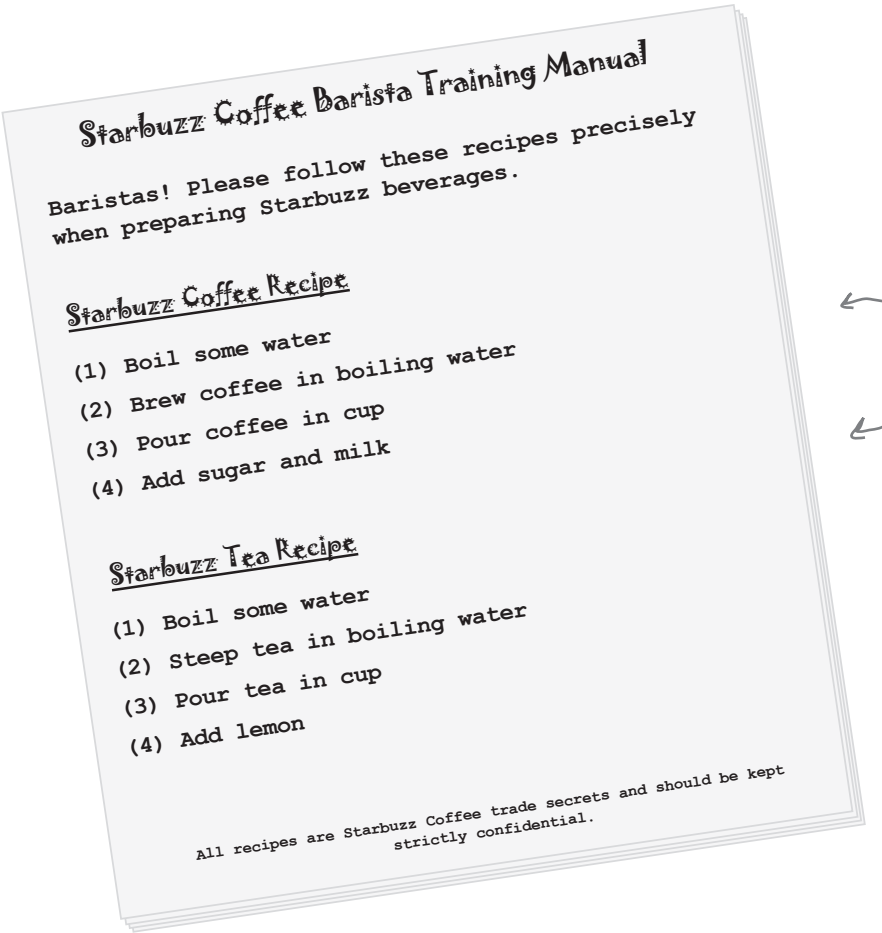
Encapsulating Algorithms

We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas...what could be next? 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. Let's get started...

It's time for some more caffeine

Some people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine, of course!

But there's more; tea and coffee are made in very similar ways. Let's check it out:



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?

Whipping up some coffee and tea classes (in Java)

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

Here's the coffee:

Here's our Coffee class for making coffee.

```
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 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.

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");
    }

    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.

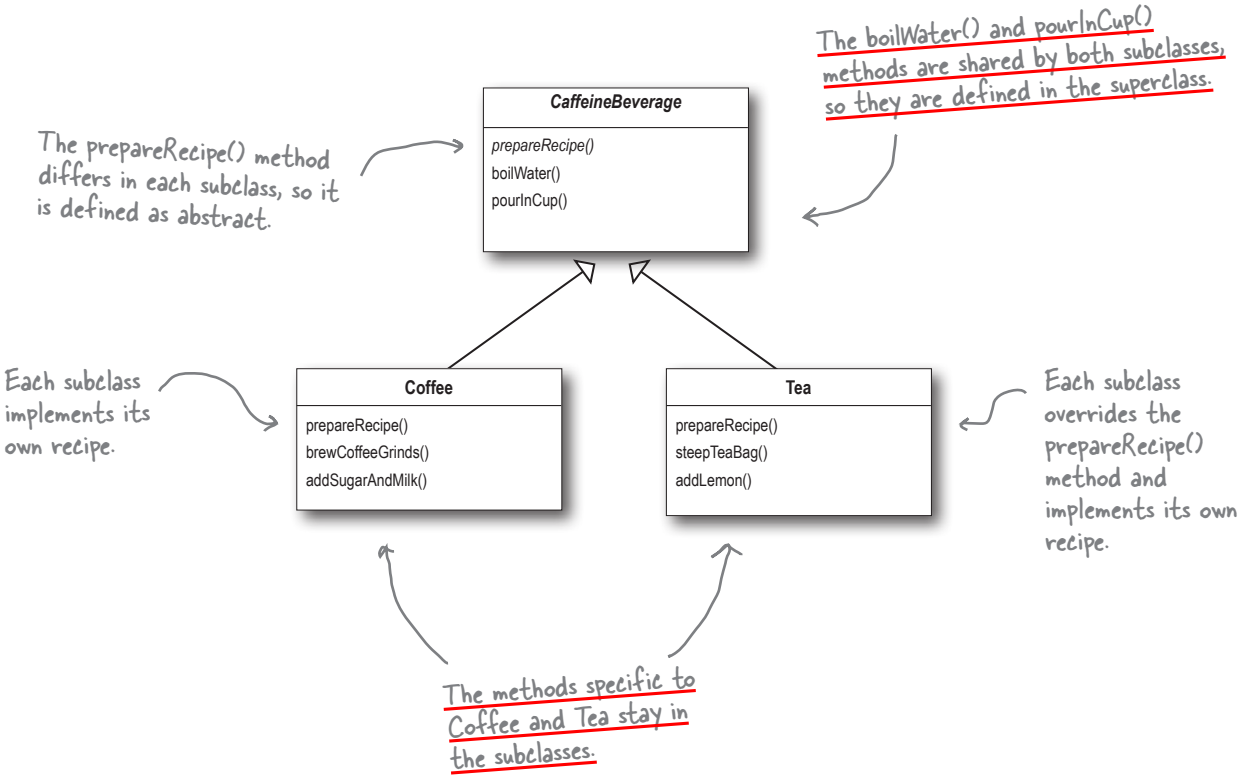
When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar.

Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:

Let's abstract that Coffee and Tea

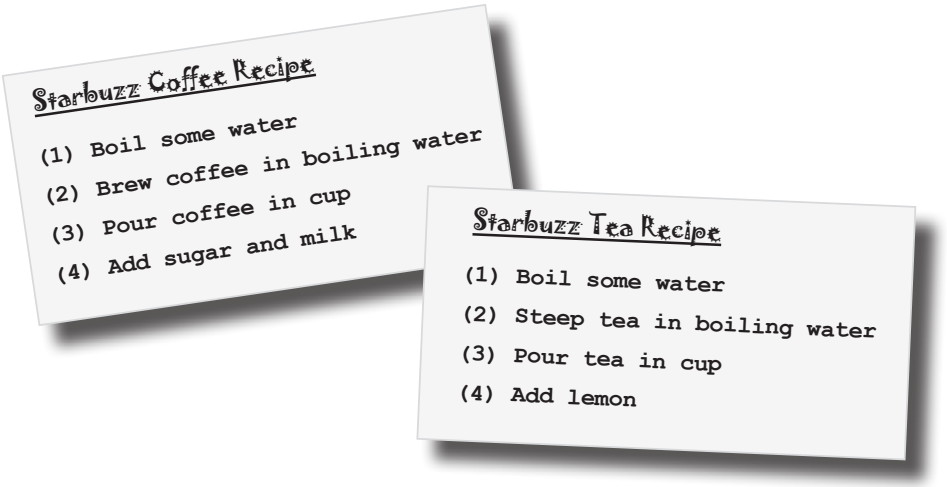
It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:



Did we do a good job on the redesign? Hmmmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

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



Notice that both recipes follow the same algorithm:

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

These aren't abstracted but are the same; they just apply to different beverages.

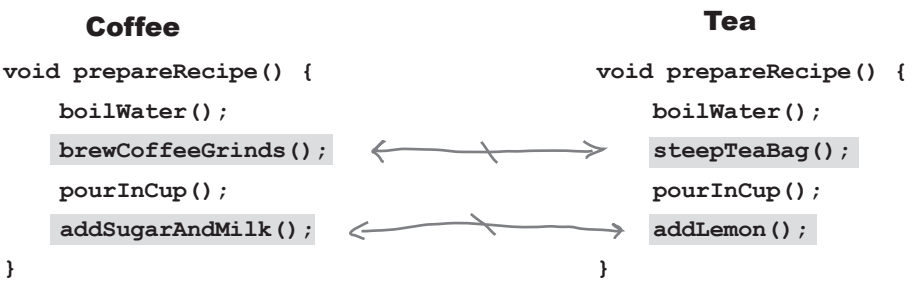
These two are already abstracted into the base class.

So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

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



Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepareRecipe() method will look like this:

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

2. Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we'll start with the CaffeineBeverage superclass:

(Code on the next page.)

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).

3 Finally, we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

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

```
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");
    }
}
```

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.

Sharpen your pencil

Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

What have we done?

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the tea bag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

generalize

relies on subclass for some steps

Tea subclass

2. Steep the tea bag in the water
4. Add lemon

relies on subclass for some steps

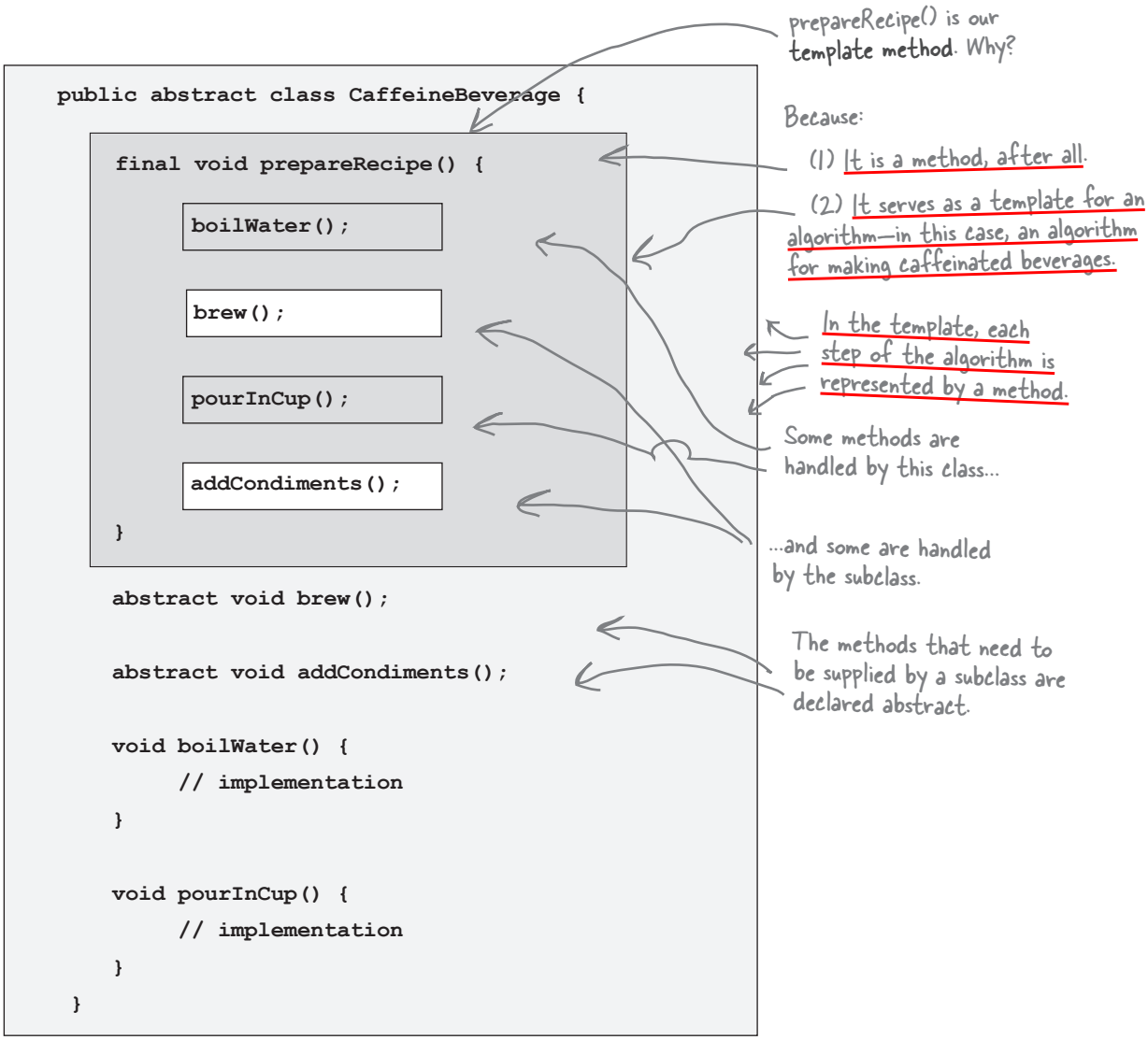
Coffee subclass

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

Meet the Template Method

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method":



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

Let's make some tea...

Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

1. Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

2. Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

3. First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.

4. Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

5. Now we pour the tea in the cup; this is the same for all beverages, so it happens in CaffeineBeverage:

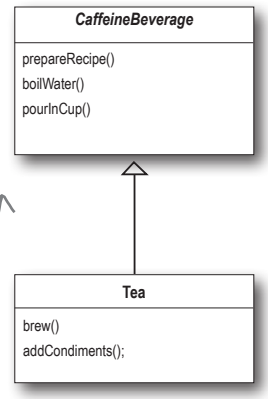
```
pourInCup();
```

6. Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```

```
boilWater();
brew();
pourInCup();
addCondiments();
```

The prepareRecipe() method controls the algorithm. No one can change this, and it counts on subclasses to provide some or all of the implementation.



What did the Template Method get us?

Underpowered Tea & Coffee implementation

New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.	The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
Code is duplicated across Coffee and Tea.	The CaffeineBeverage class maximizes reuse among the subclasses.
Code changes to the algorithm require opening the subclasses and making multiple changes.	The algorithm lives in one place and code changes only need to be made there.
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	The Template Method Pattern provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
Knowledge of the algorithm and how to implement it is distributed over many classes.	The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

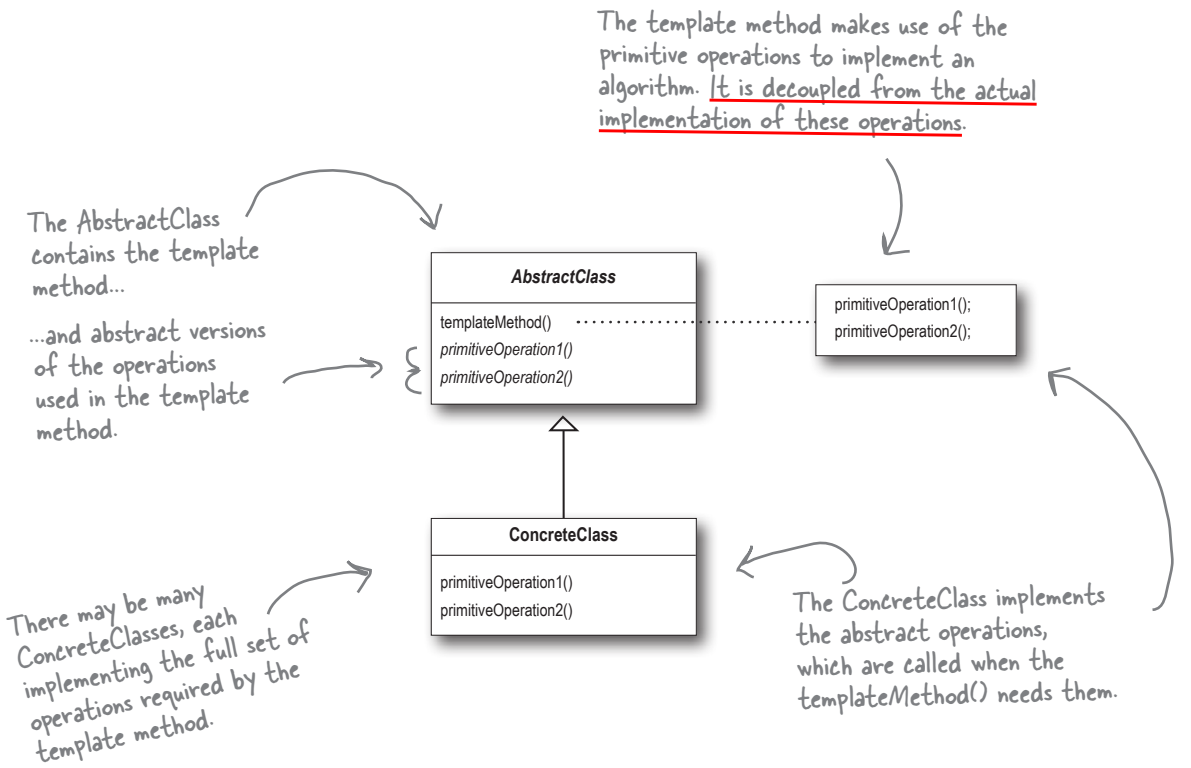
Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

> **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.

This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

Let's check out the class diagram:



Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

```
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.

We also have a concrete operation defined in the abstract class. This could be overridden by subclasses, or we could prevent overriding by declaring concreteOperation() as final. More about this in a bit...

Code Way Up Close

Now we're going to look even closer at the types of method that can go in the abstract class:

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.

Hooked on Template Method...

A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

There are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later:


```
public abstract class CaffeineBeverageWithHook {

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

    abstract void brew();

    abstract void addCondiments();

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

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

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage class evaluates a certain part of the algorithm—that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

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

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

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false, depending on the input.

This code asks if the user would like milk and sugar and gets the input from the command line.

Let's run the Test Drive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee.

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

Create a tea.

Create a coffee.

And call prepareRecipe() on both!

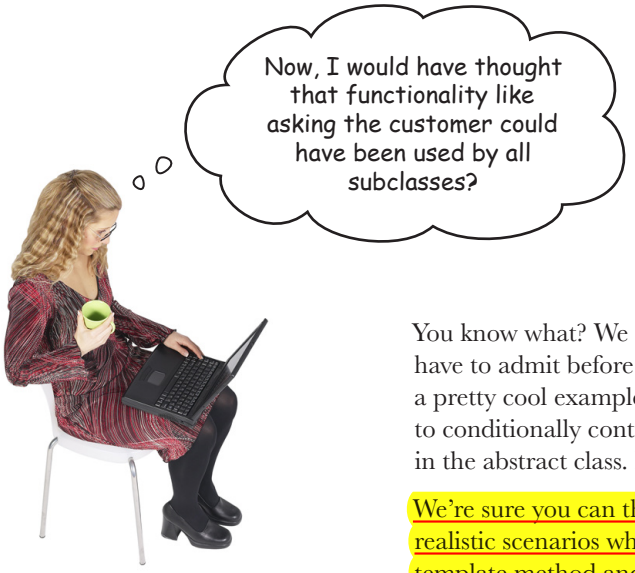
And let's give it a run...

```
File  Edit  Window  Help  send-more-honesttea
%java BeverageTestDrive
Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

A steaming cup of tea, and yes, of course we want that lemon!

And a nice hot cup of coffee, but we'll pass on the waistline-expanding condiments.



You know what? We agree with you. But you have to admit before you thought of that, it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no Dumb Questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: What are hooks really supposed to be used for?

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass's implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen or just happened. For instance, a hook method like justReorderedList() allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen, a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: Does a subclass have to implement all the abstract methods in the AbstractClass?

A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: It seems like I should keep my abstract methods small in number; otherwise, it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes you can do this by not making the steps of your algorithm too granular. But it's obviously a tradeoff: the less granularity, the less flexibility.

Remember, too, that some steps will be optional, so you can implement these as hooks rather than abstract methods, easing the burden on the subclasses of your abstract class.

The Hollywood Principle

We've got another design principle for you; it's called the Hollywood Principle:

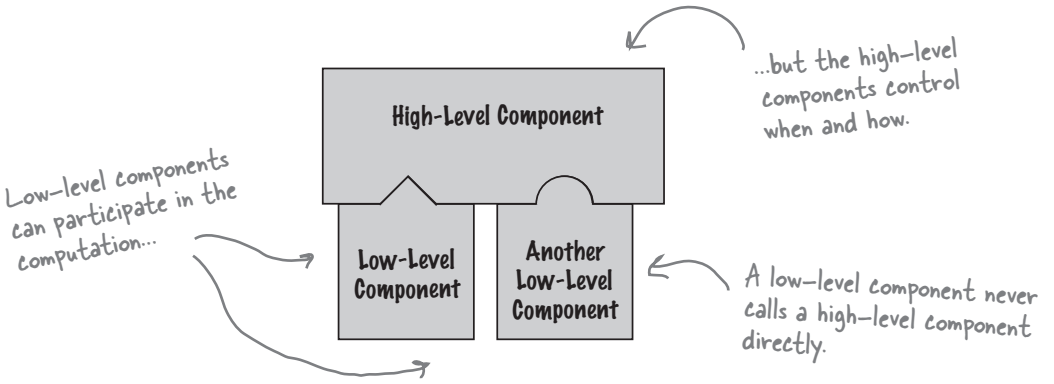
The Hollywood Principle

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

Easy to remember, right? But what has it got to do with OO design?

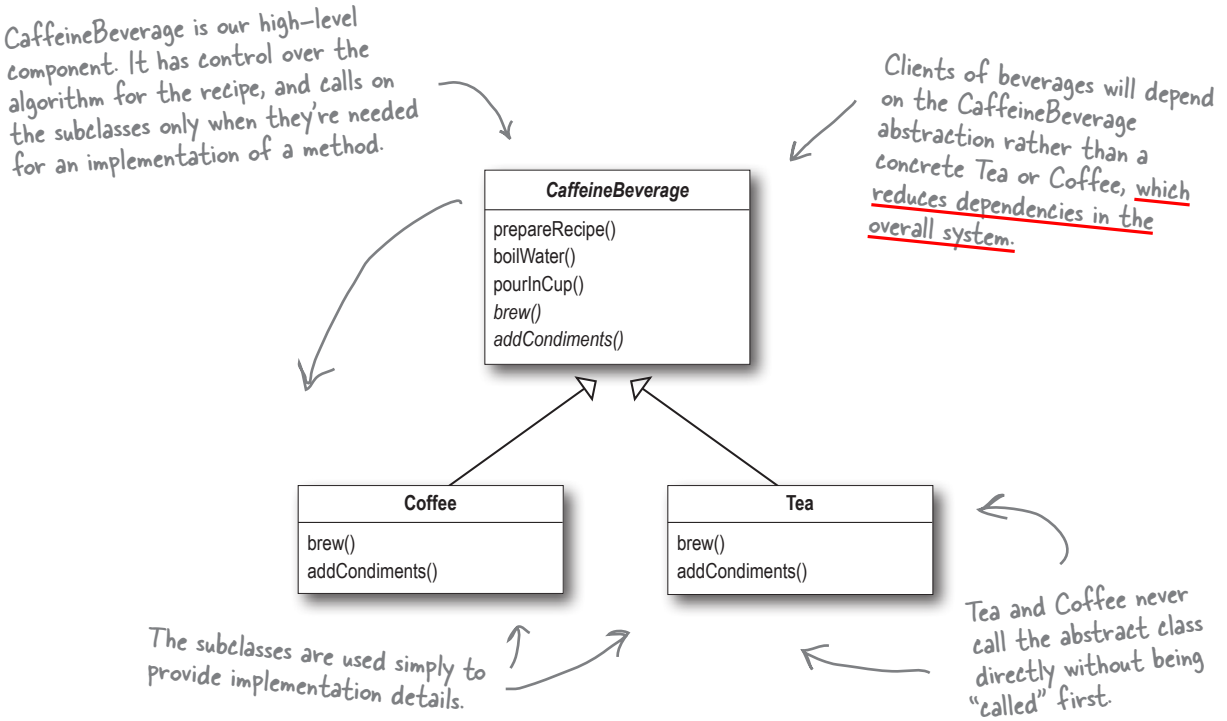
The Hollywood Principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components the "don't call us, we'll call you" treatment.



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." How? Let's take another look at our CaffeineBeverage design:

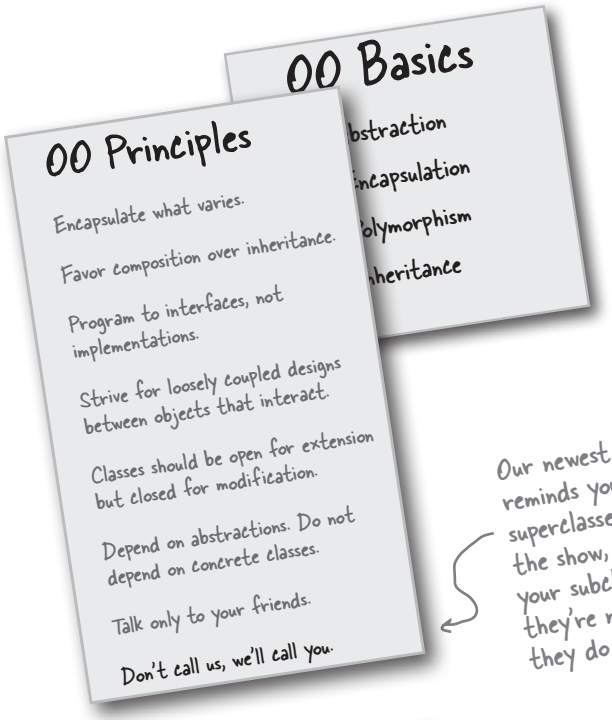


What other patterns make use of the Hollywood Principle?

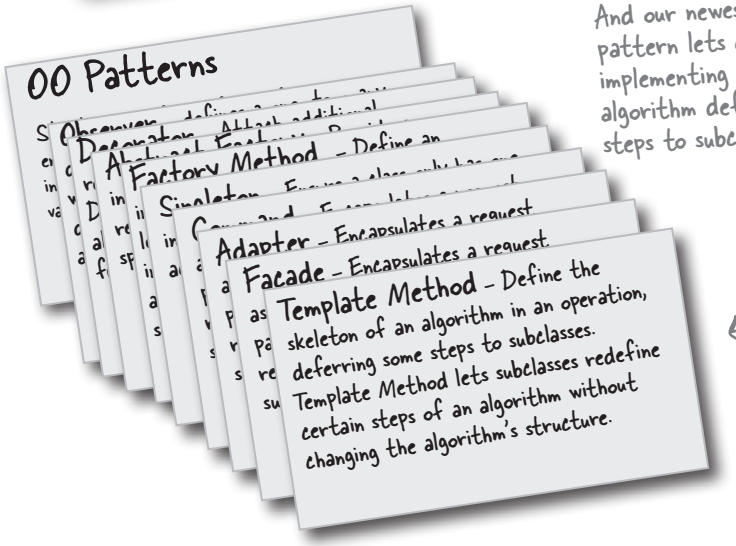
The Factory Method and Observer; any others?

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method, you can reuse code like a pro while keeping control of your algorithms.



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.



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

BULLET POINTS

- 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.
- 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.