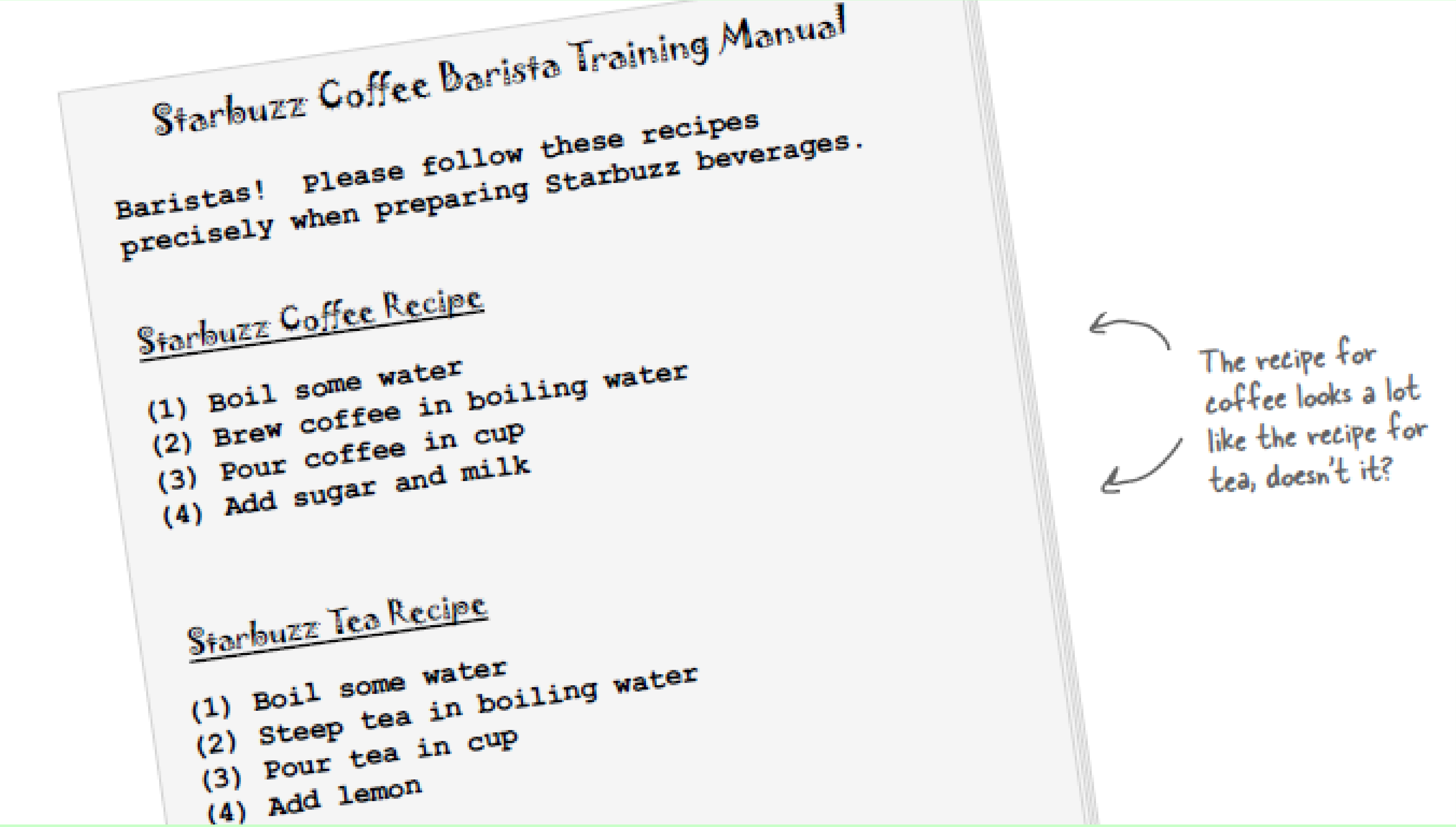
Agenda

- Template Method Pattern

Template Method Pattern

Encapsulating Algorithms

Coffee & Tea Recipe



Starbuzz Coffee Barista Training Manual

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

Starbuzz Coffee Recipe

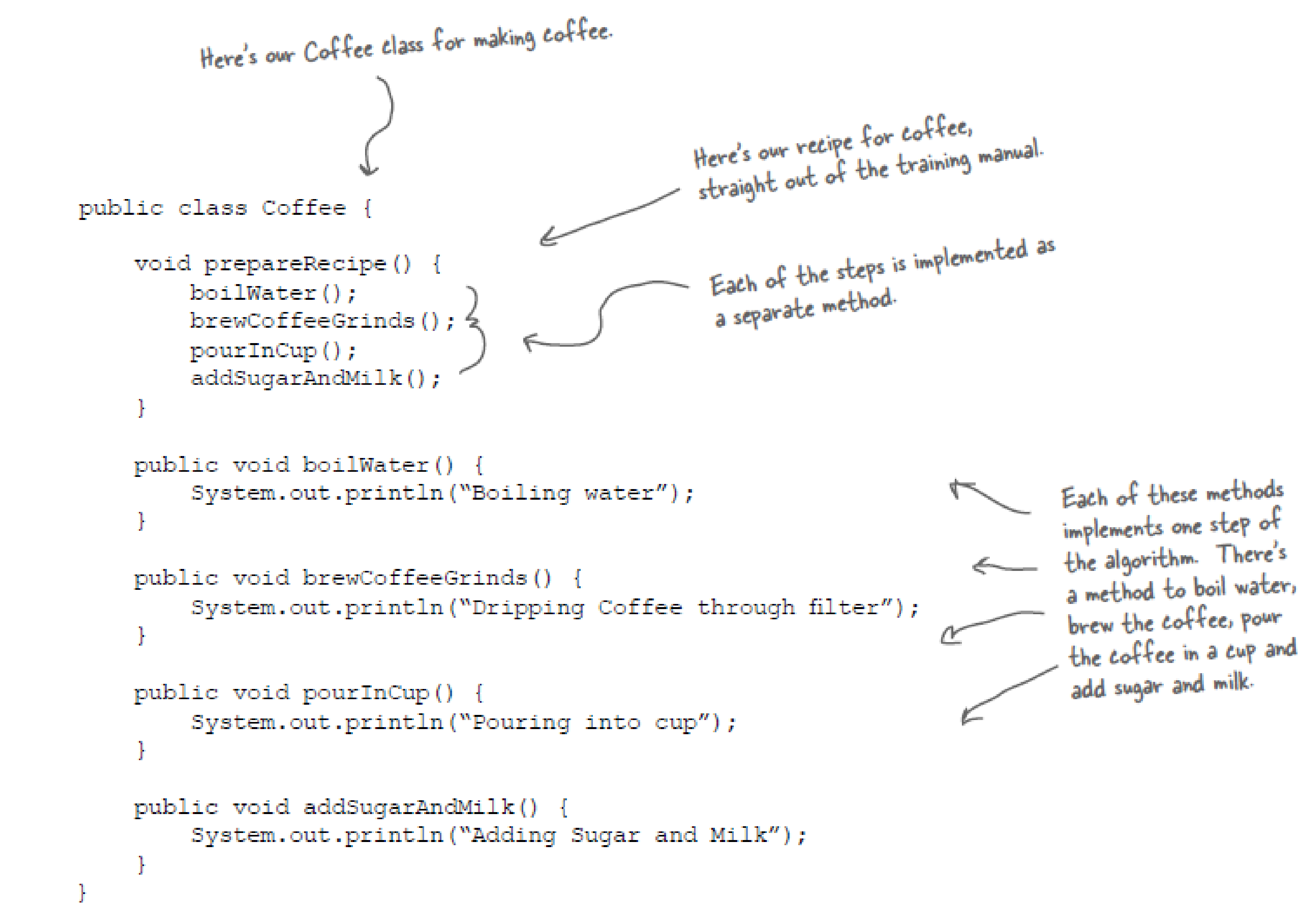
(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

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

Coffee & Tea Classes in Java



Here's our Coffee class for making coffee.

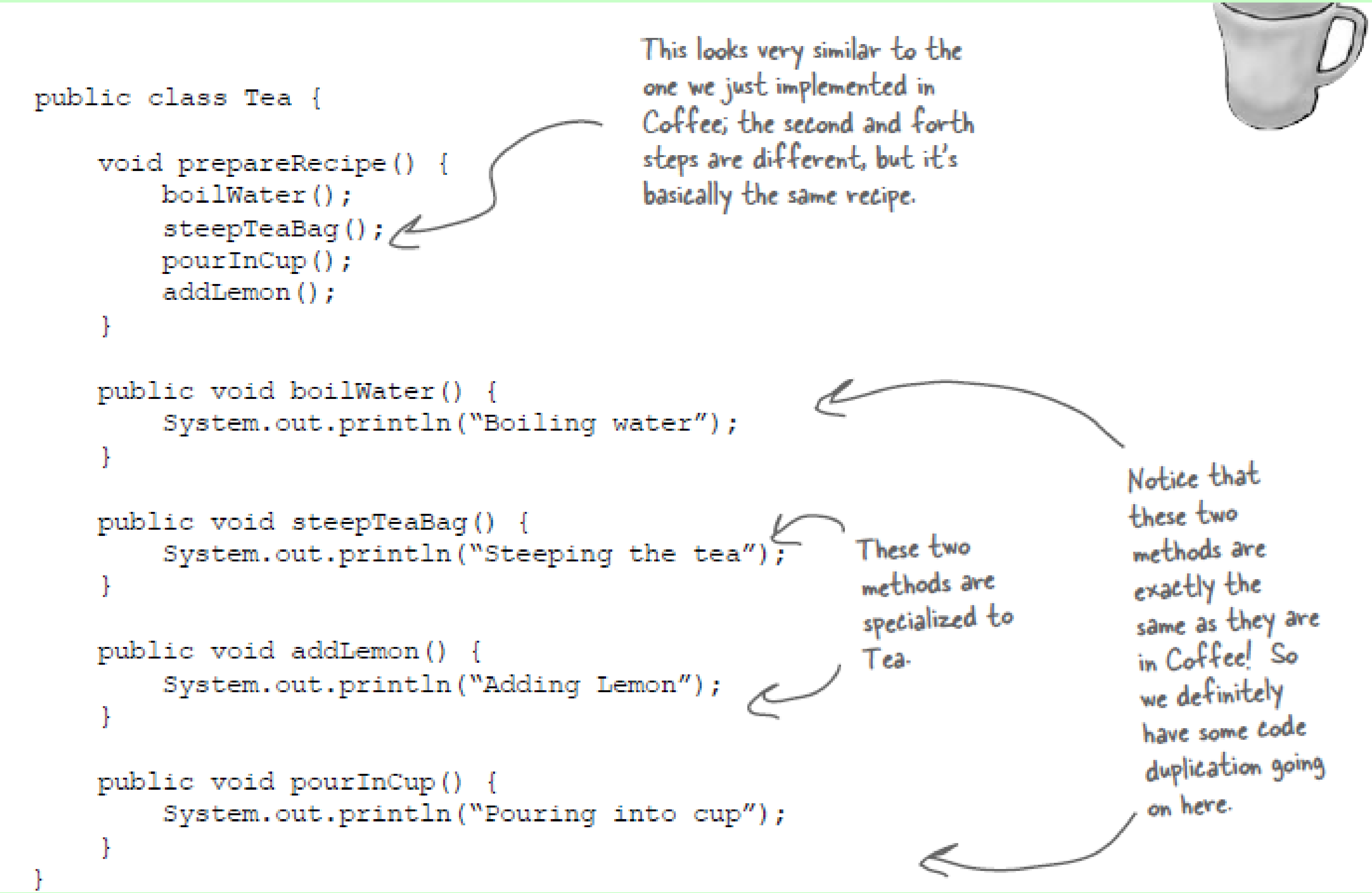
Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```
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”);
    }
}
```

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.

Coffee & Tea Classes in Java



```
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”);
    }
}
```

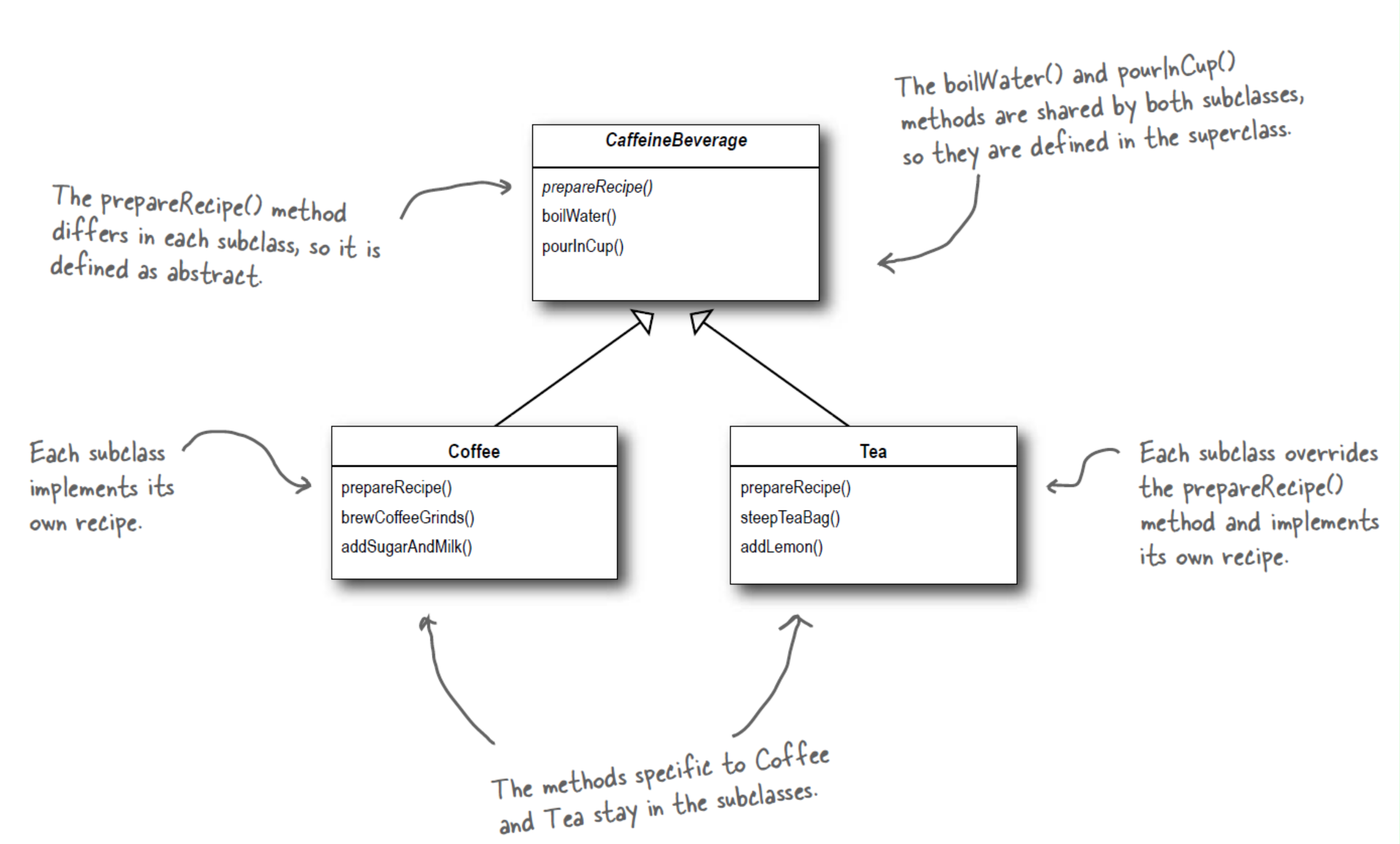
This looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.

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.

How can we avoid code duplication?

How can we avoid code duplication?



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.

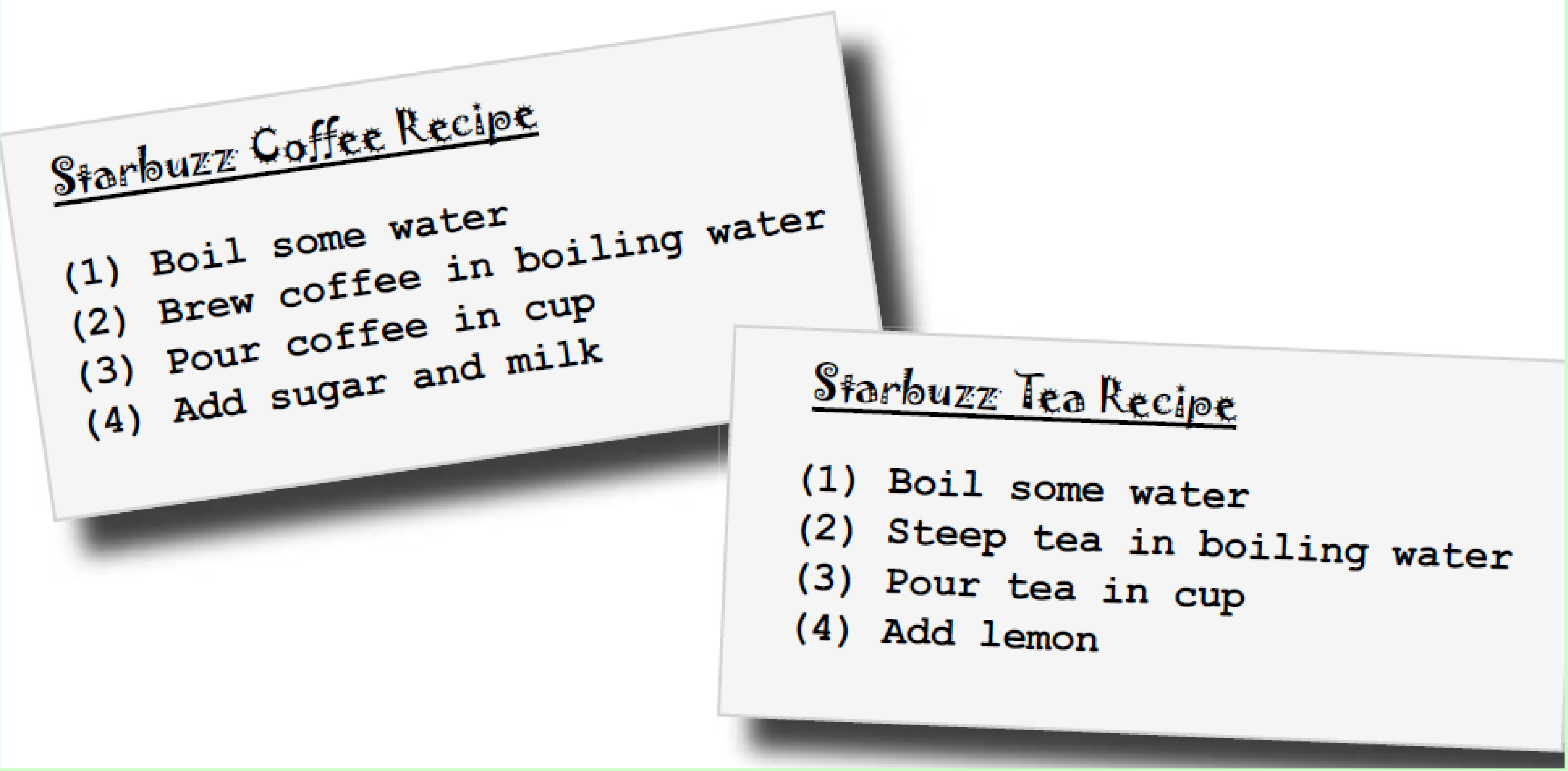
The methods specific to Coffee and Tea stay in the subclasses.

Redesign Coffee & Tea

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

Taking the design further

What else do we have in common?



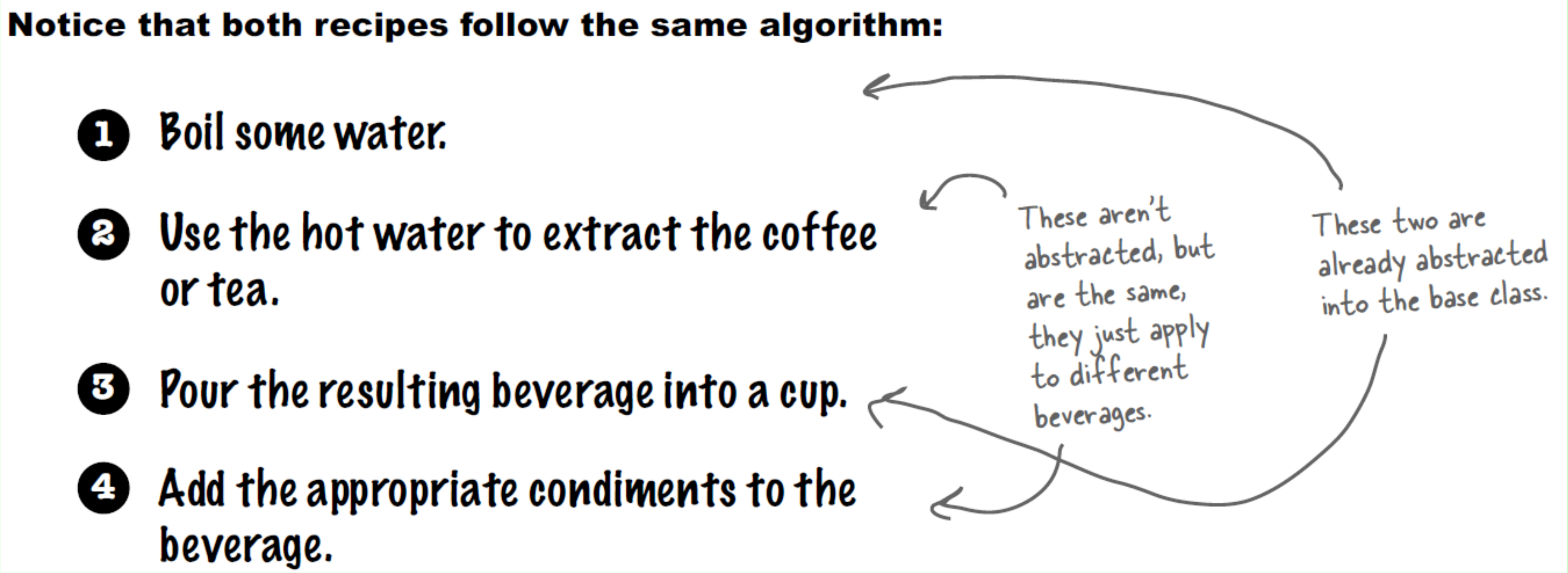
Starbuzz Coffee Recipe

(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

Taking the design further



Notice that both recipes follow the same algorithm:

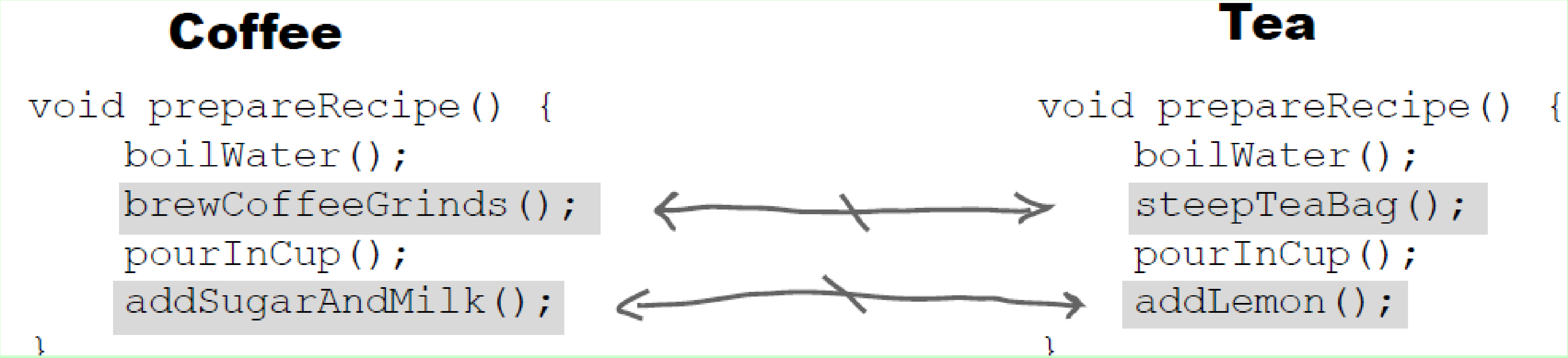
1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

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

These two are already abstracted into the base class.

Abstracting prepareRecipe()

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



Coffee

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

Tea

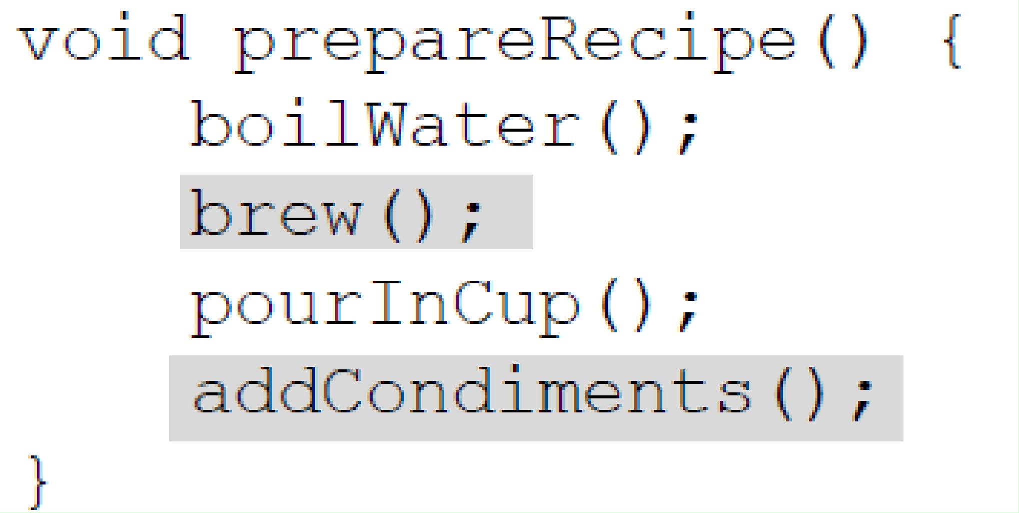
```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Abstracting prepareRecipe()

- 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.

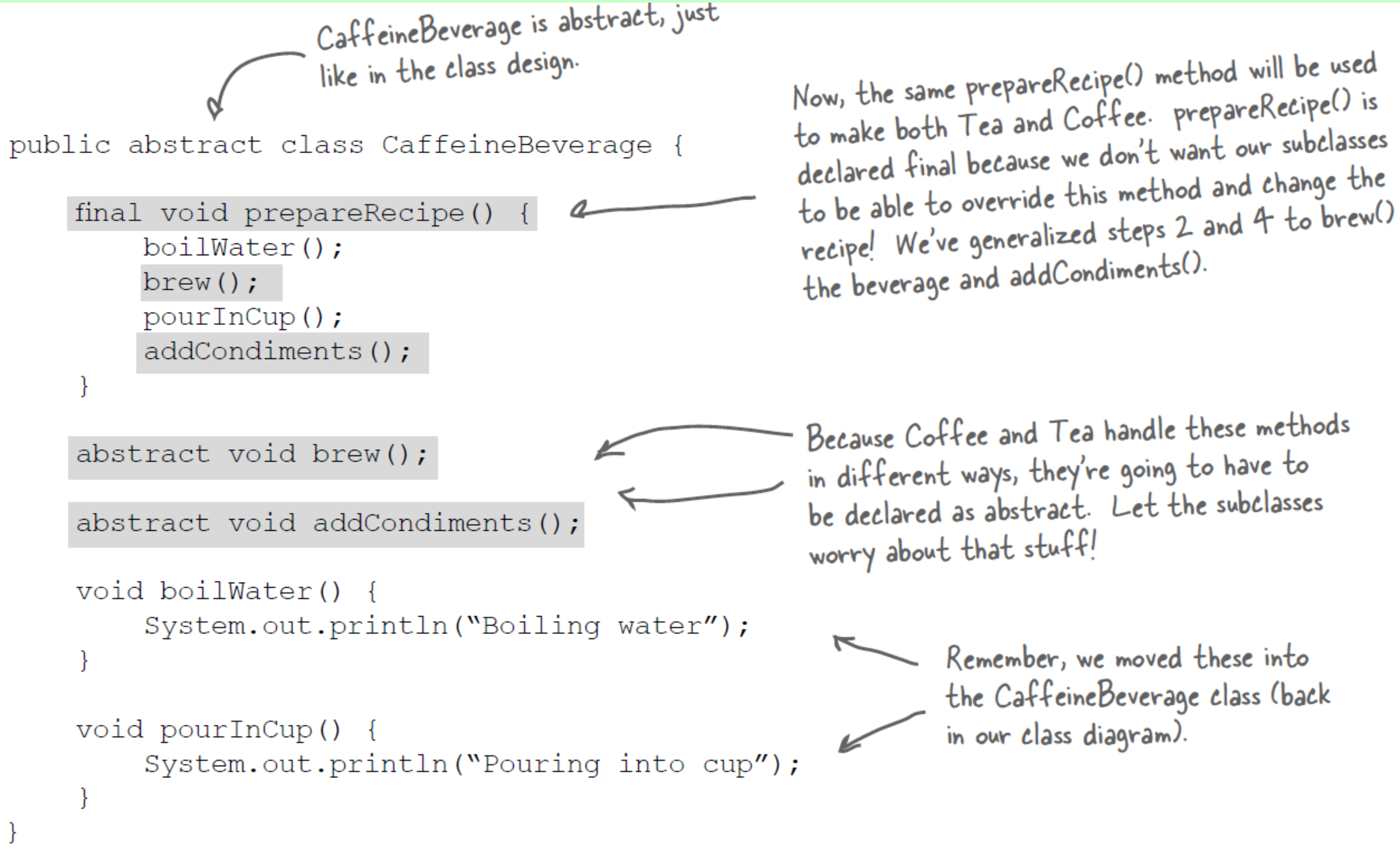
Abstracting prepareRecipe()

- So, our new prepareRecipe() method will look like this:



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

Redesign Caffeine Beverage



CaffeineBeverage is abstract, just like in the class design.

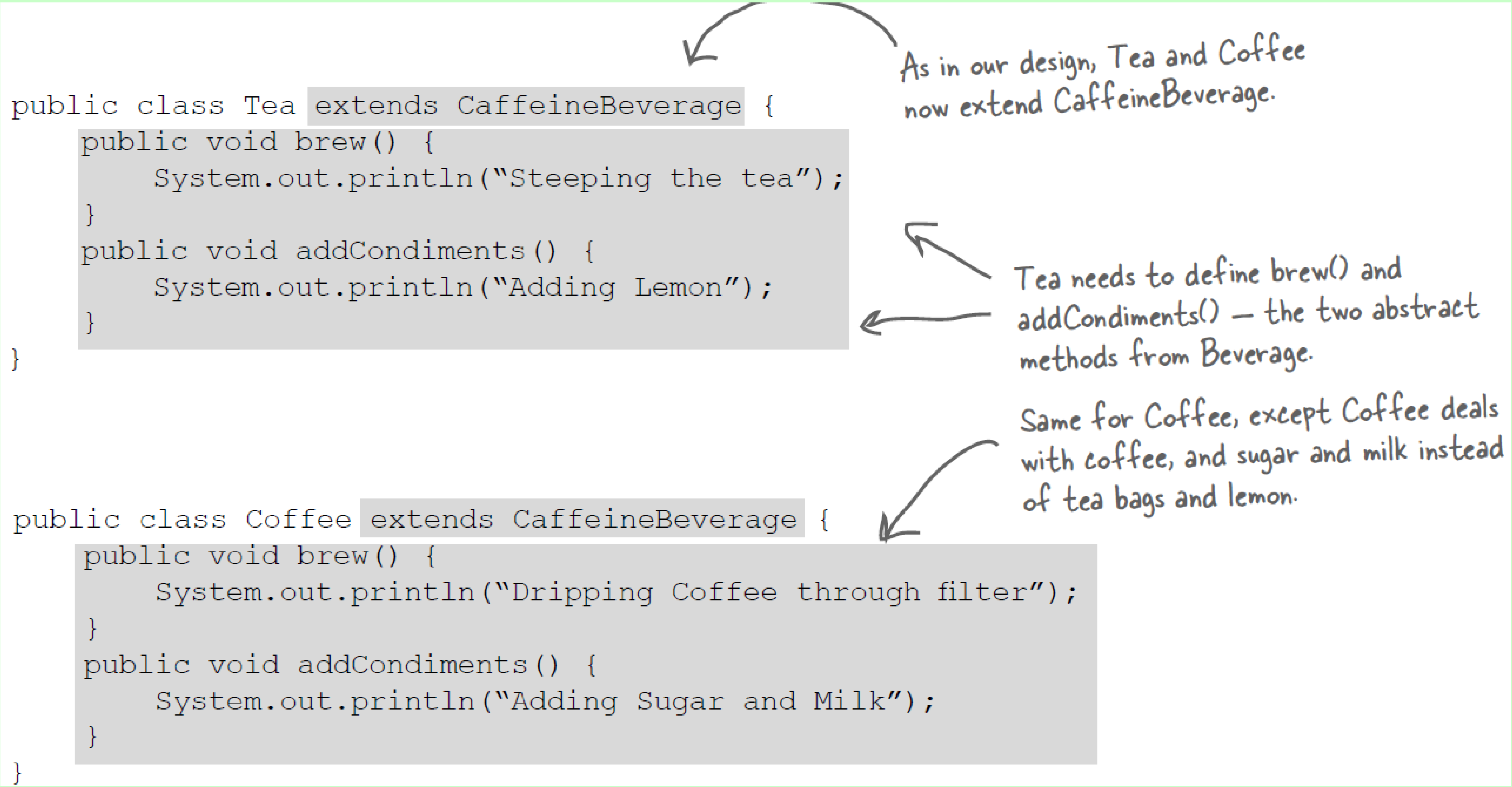
```
public abstract class CaffeineBeverage {

    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).

Redesign Tea & Coffee



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

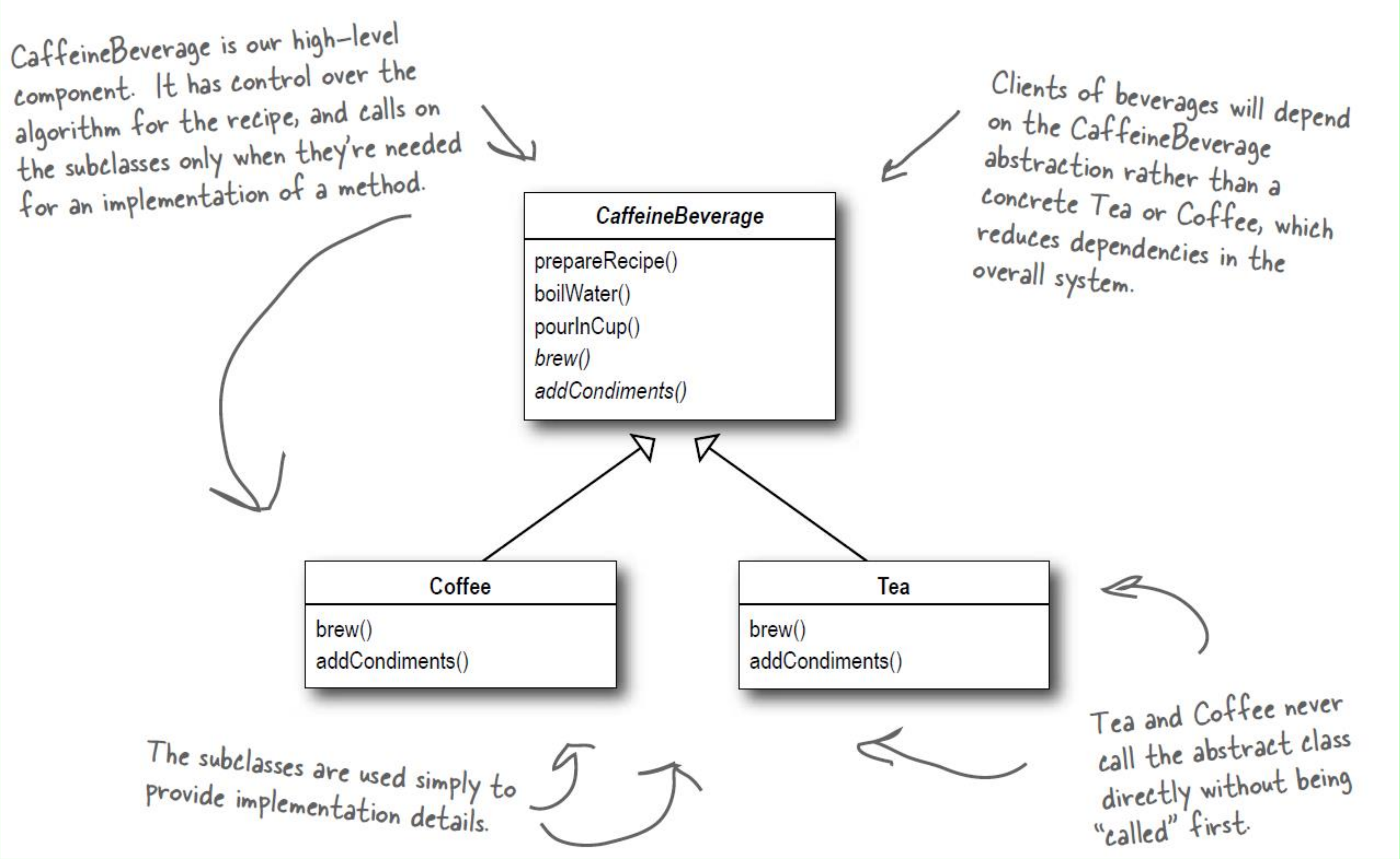
As in our design, Tea and Coffee now extend CaffeineBeverage.

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");
    }
}
```

New Class Diagram

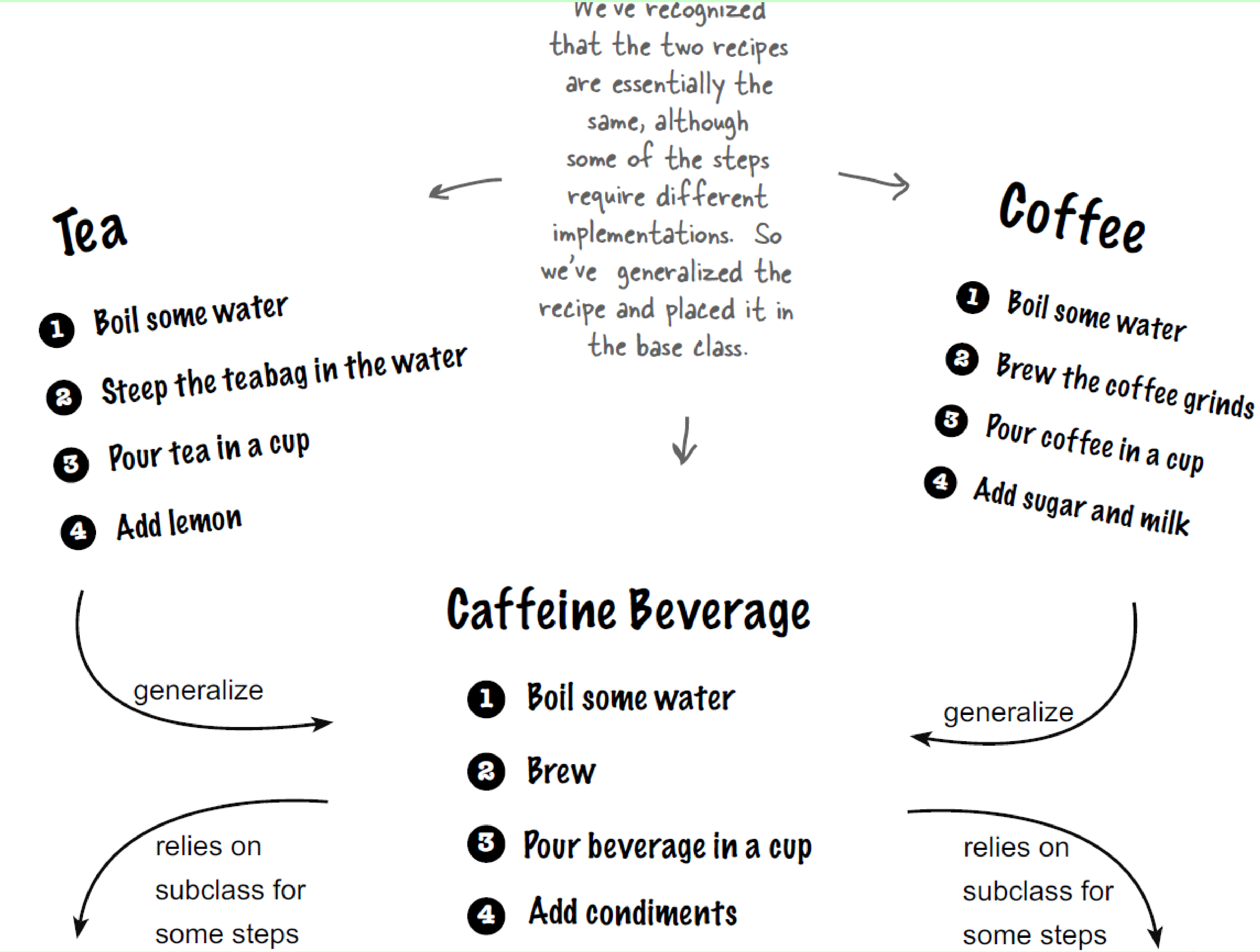


CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.

The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being "called" first.



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 teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

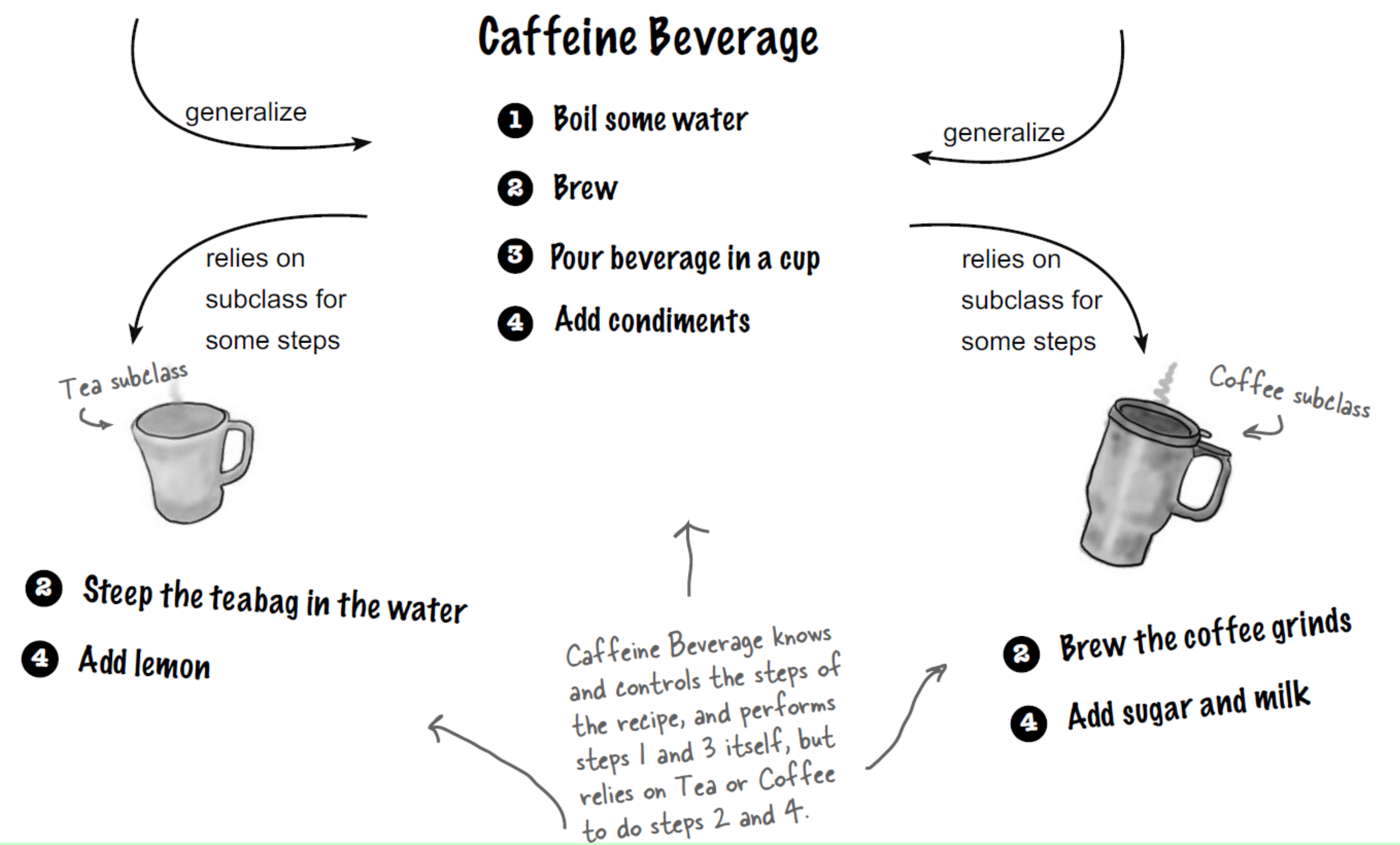
1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

Caffeine Beverage

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

generalize
generalize
relies on subclass for some steps
relies on subclass for some steps

Caffeine Beverage



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

generalize

generalize

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

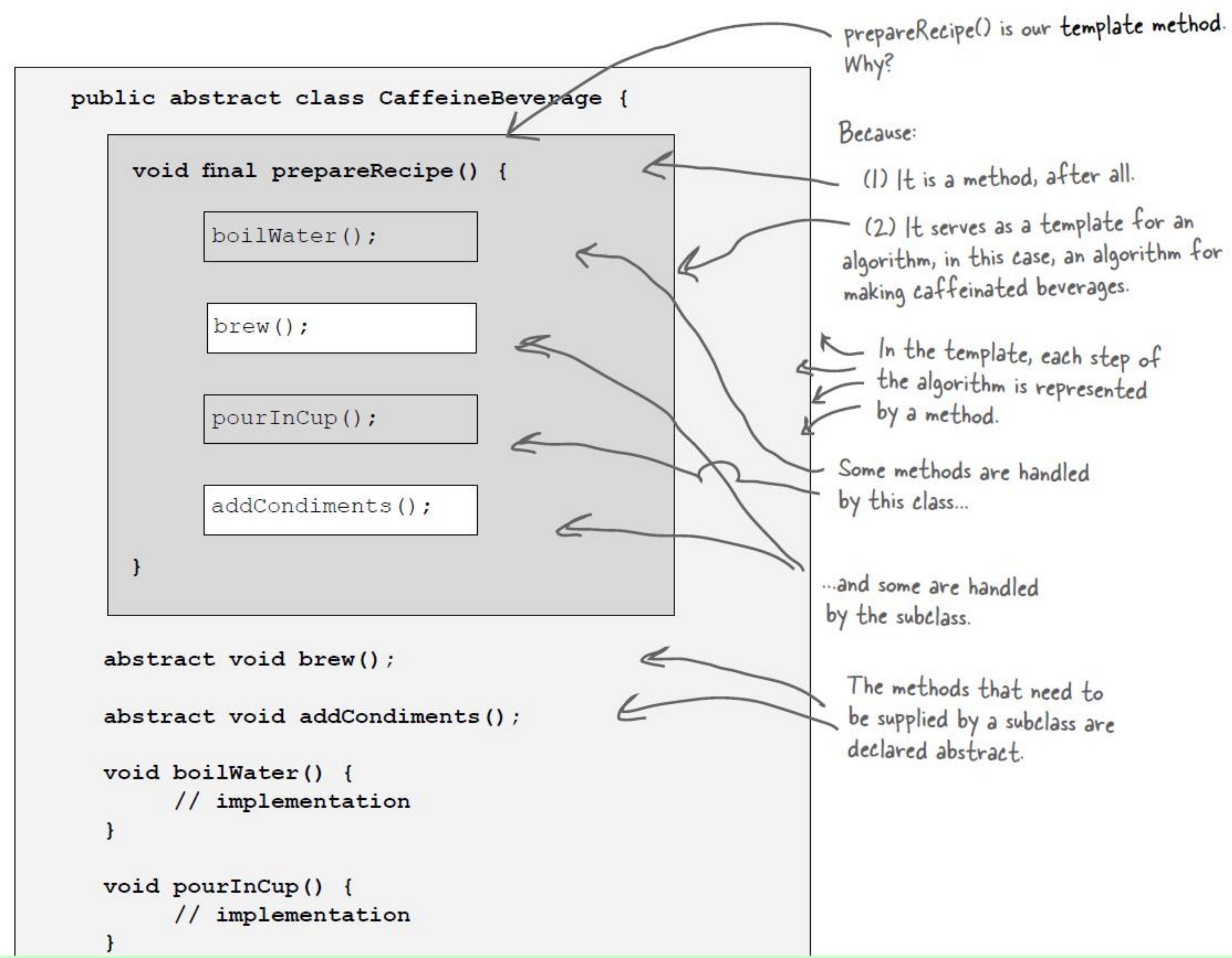
2. Steep the teabag in the water
4. Add lemon

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



```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {

        boilWater();

        brew();

        pourInCup();

        addCondiments();

    }

  abstract void brew();

  abstract void addCondiments();

  void boilWater() {
        // implementation
  }

  void pourInCup() {
        // implementation
  }
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

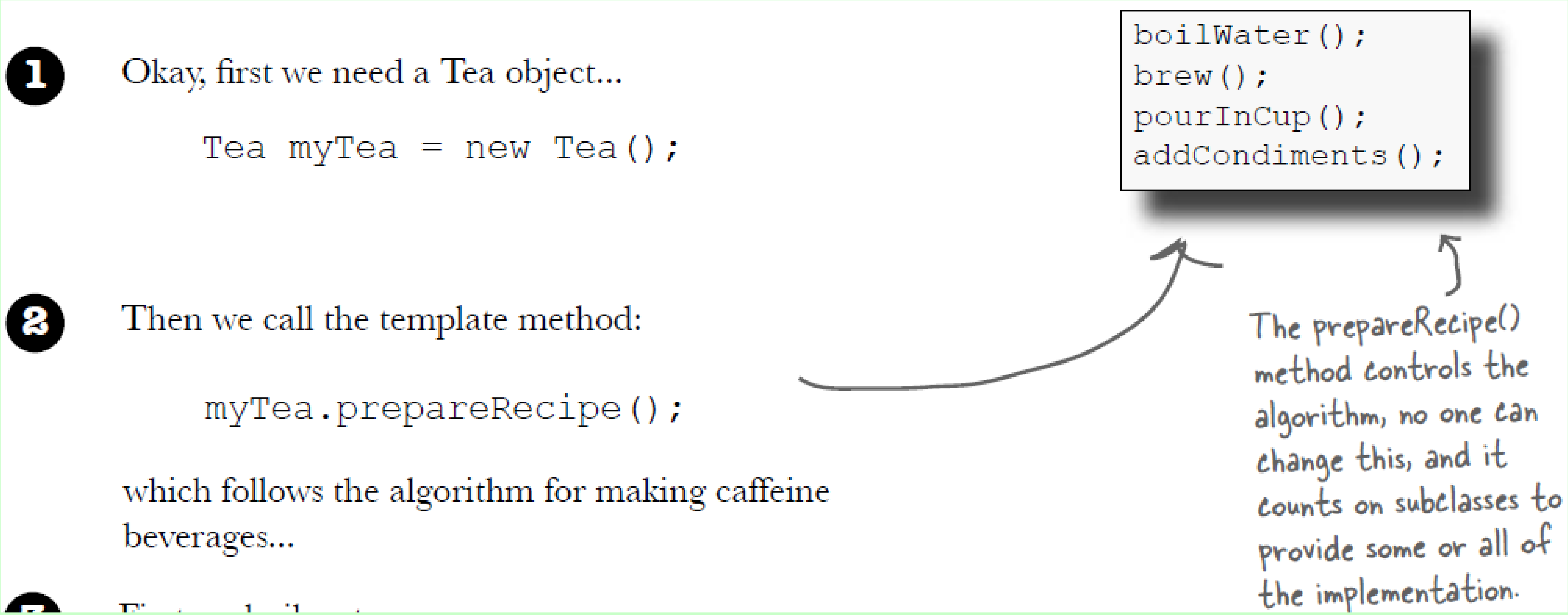
...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

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.

How the template method works



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

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

2 Then we call the template method:

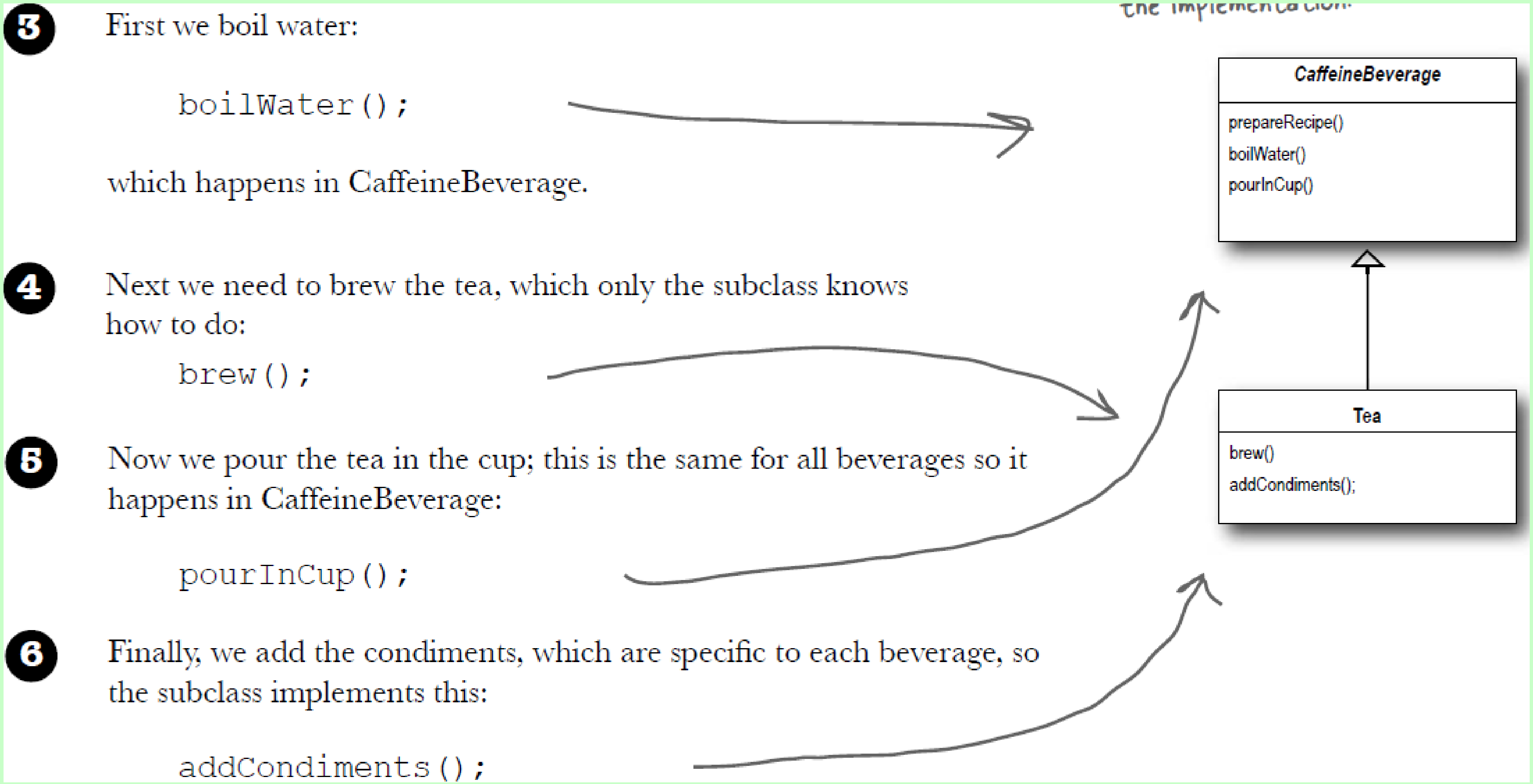
```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

```
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.

How the template method works



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();
```

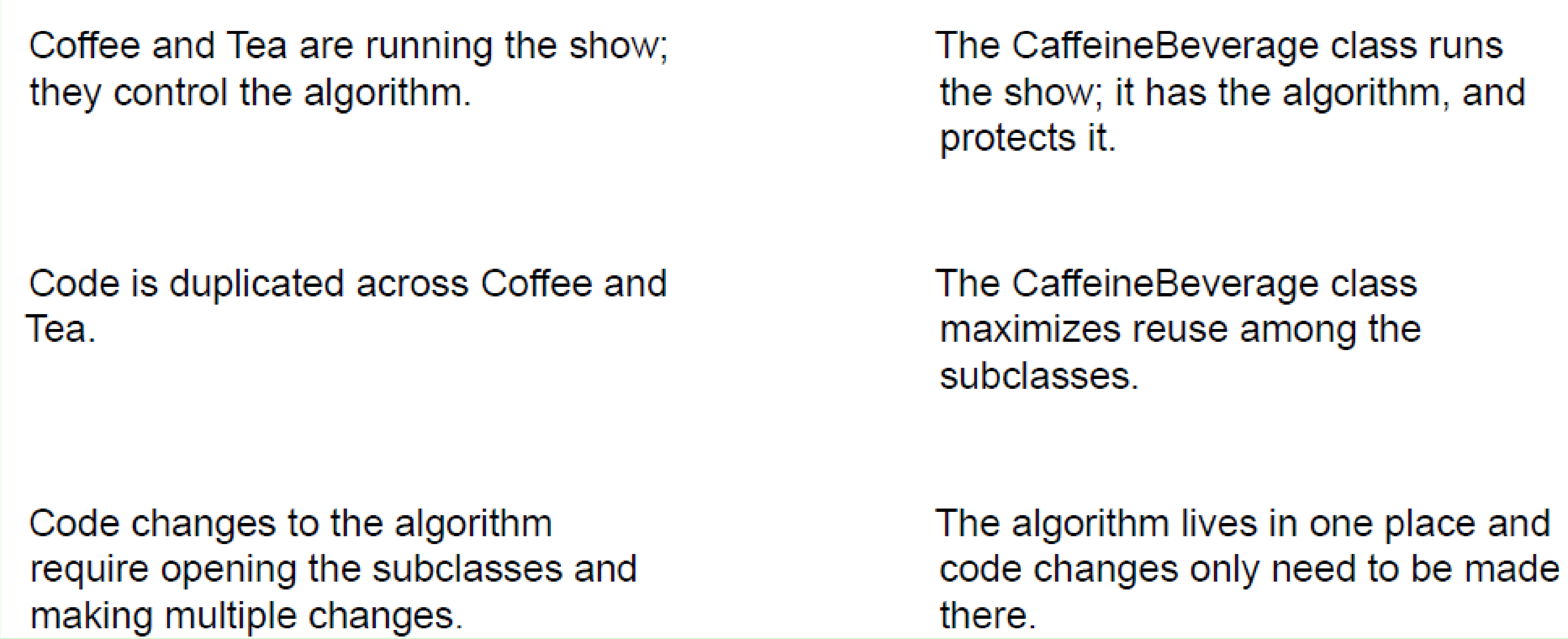
5 Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

```
pourInCup();
```

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

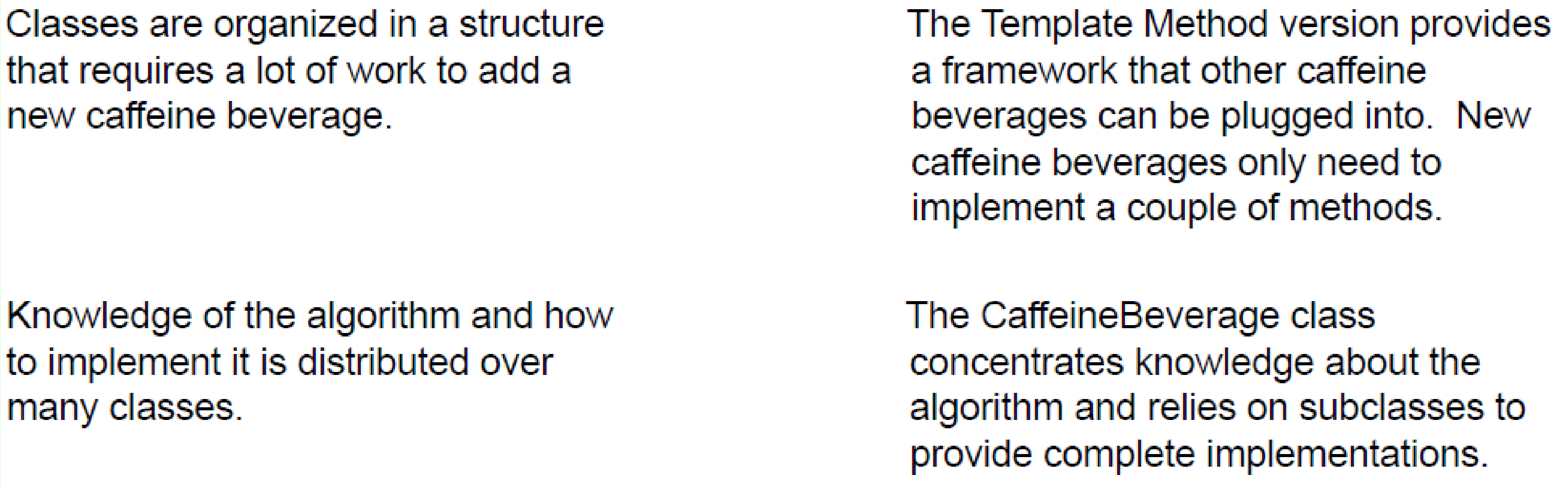
```
addCondiments();
```

Comparison of first solution with Template Method



First Solution	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.

Comparison of first solution with Template Method

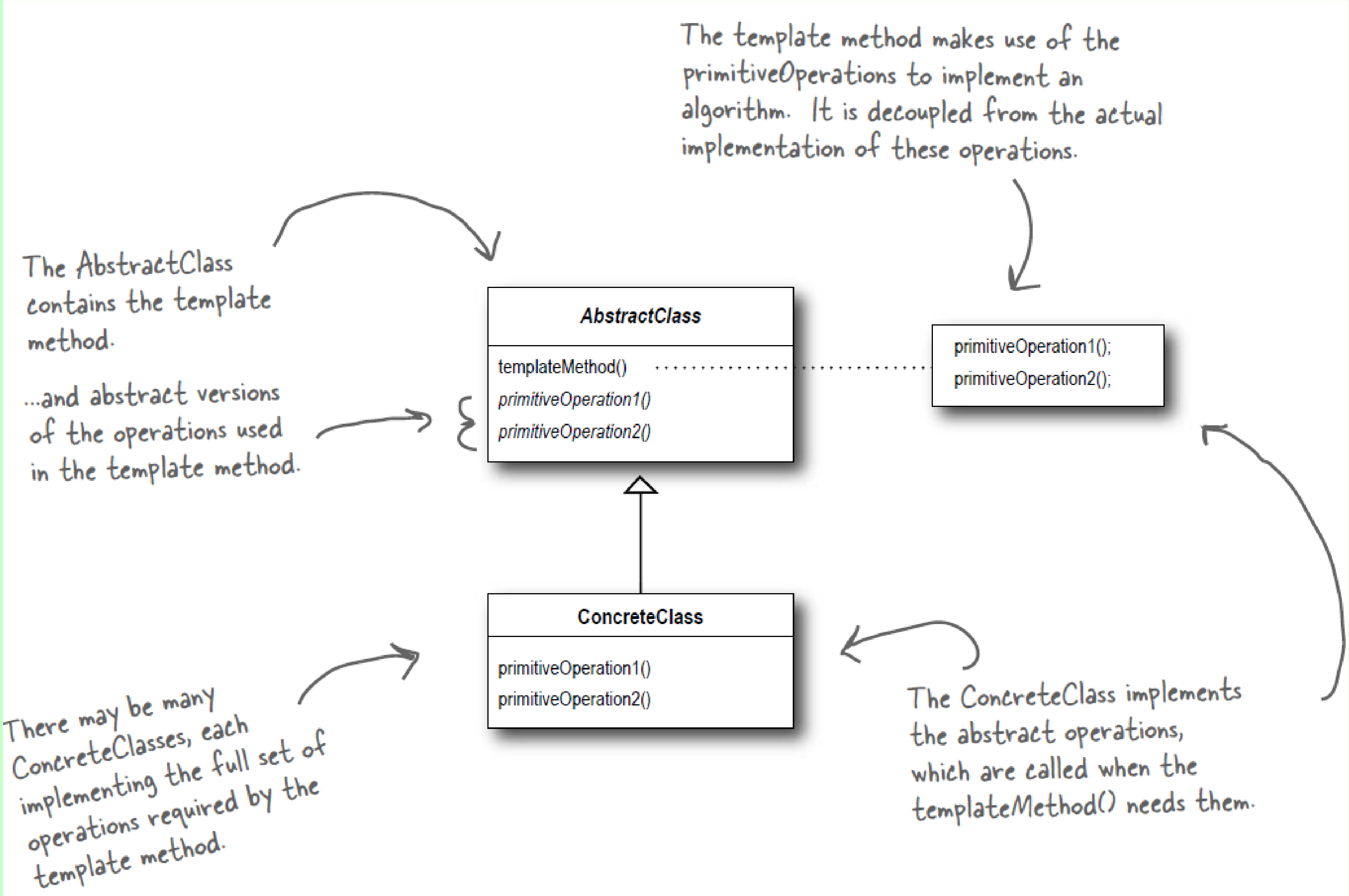


First Solution	Template Method
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	The Template Method version provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
Knowledge 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

- **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.

Class Diagram



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

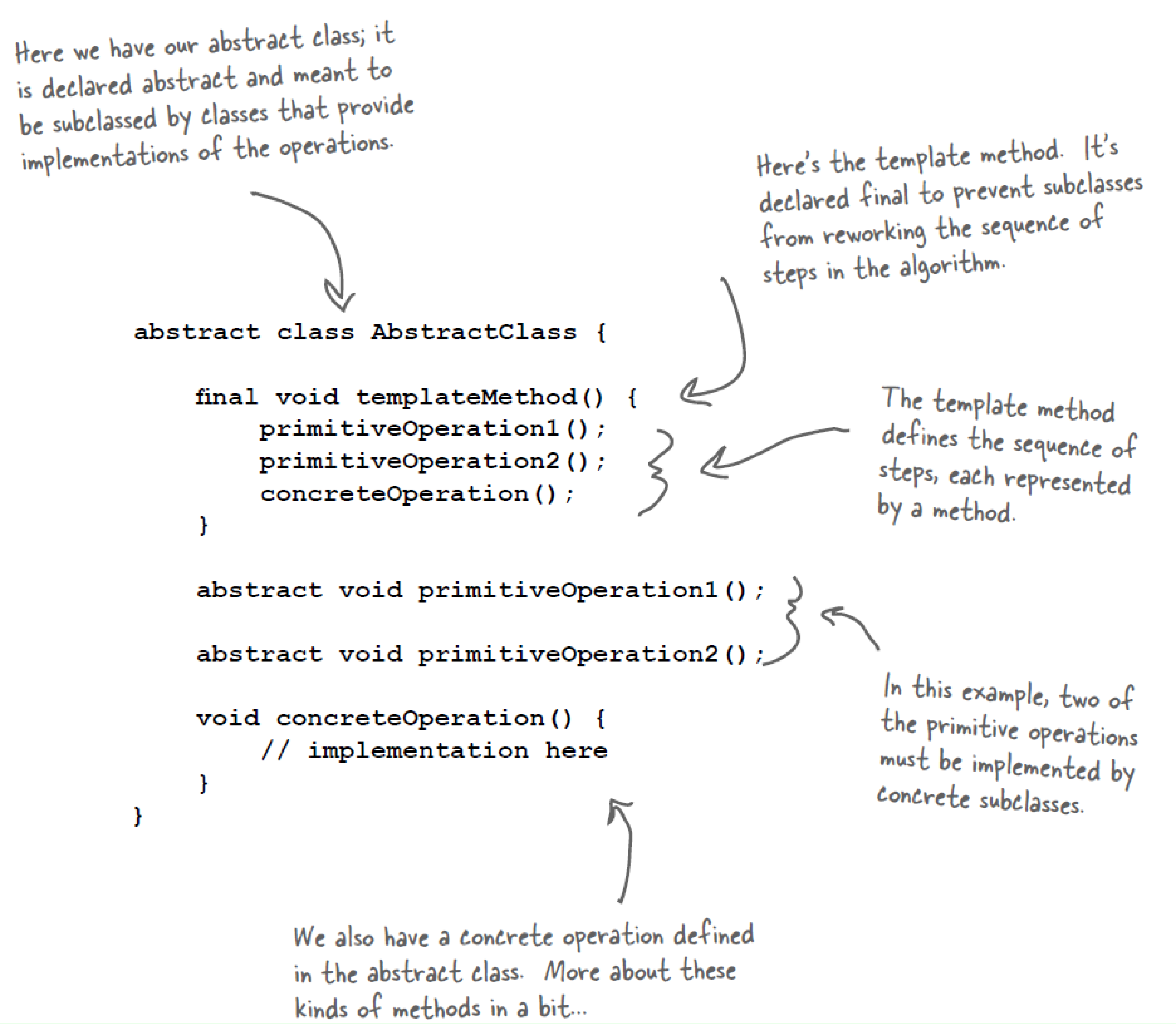
The AbstractClass contains the template method.

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

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.

Abstract Class



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

```
abstract class AbstractClass {

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

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

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

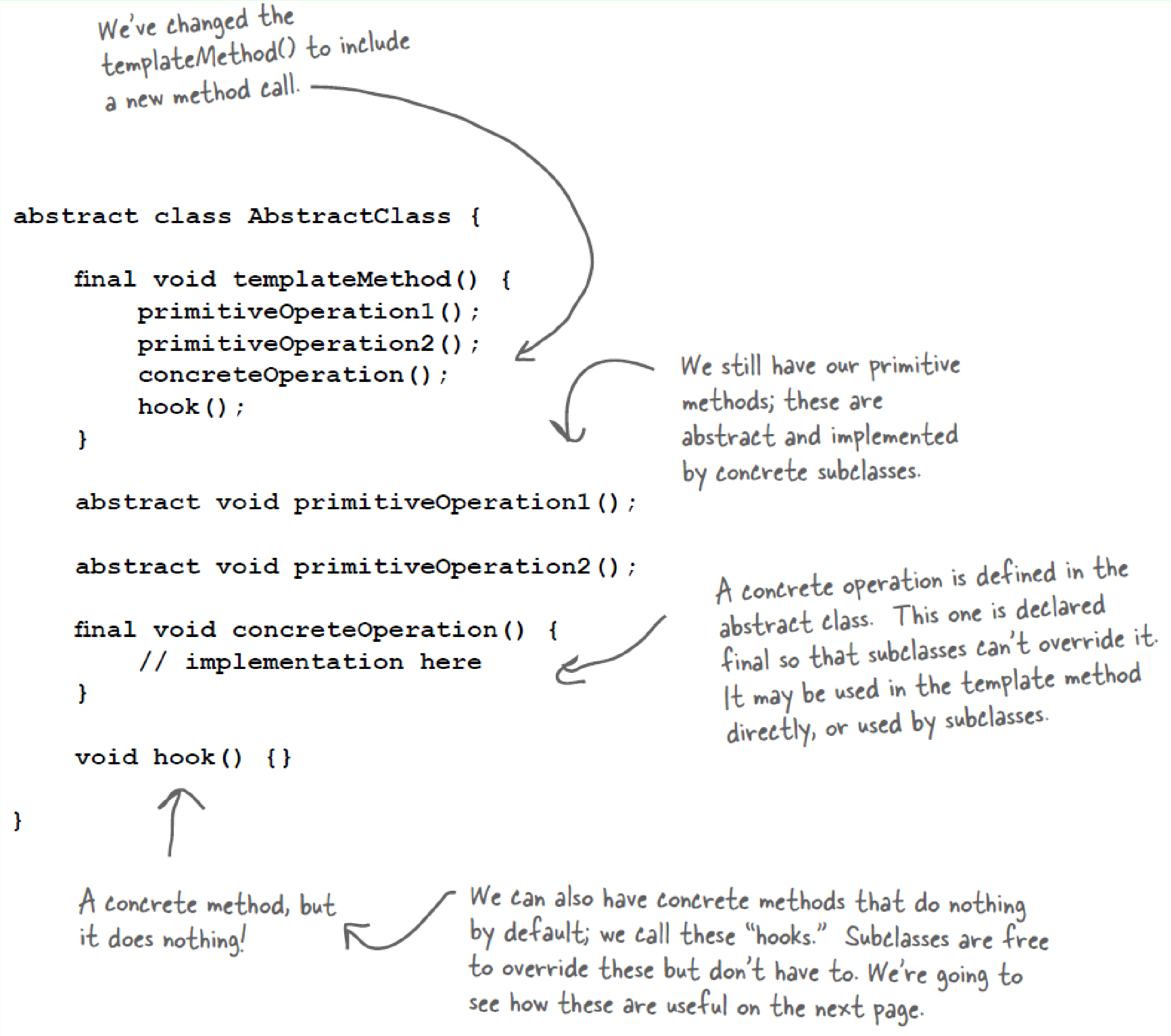
Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

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. More about these kinds of methods in a bit...

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 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!

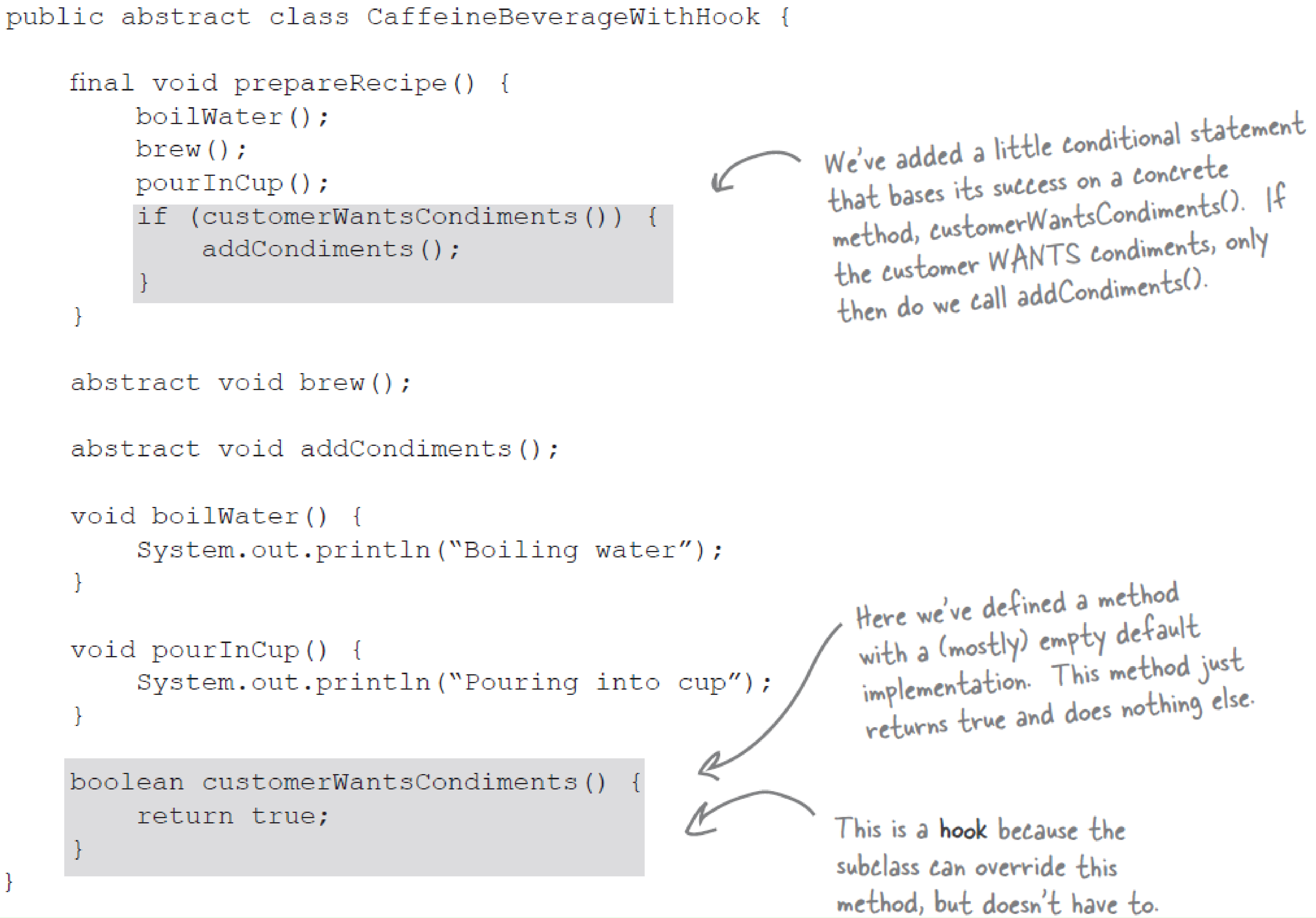
We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Hook 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.

CaffeineBeverageWithHook



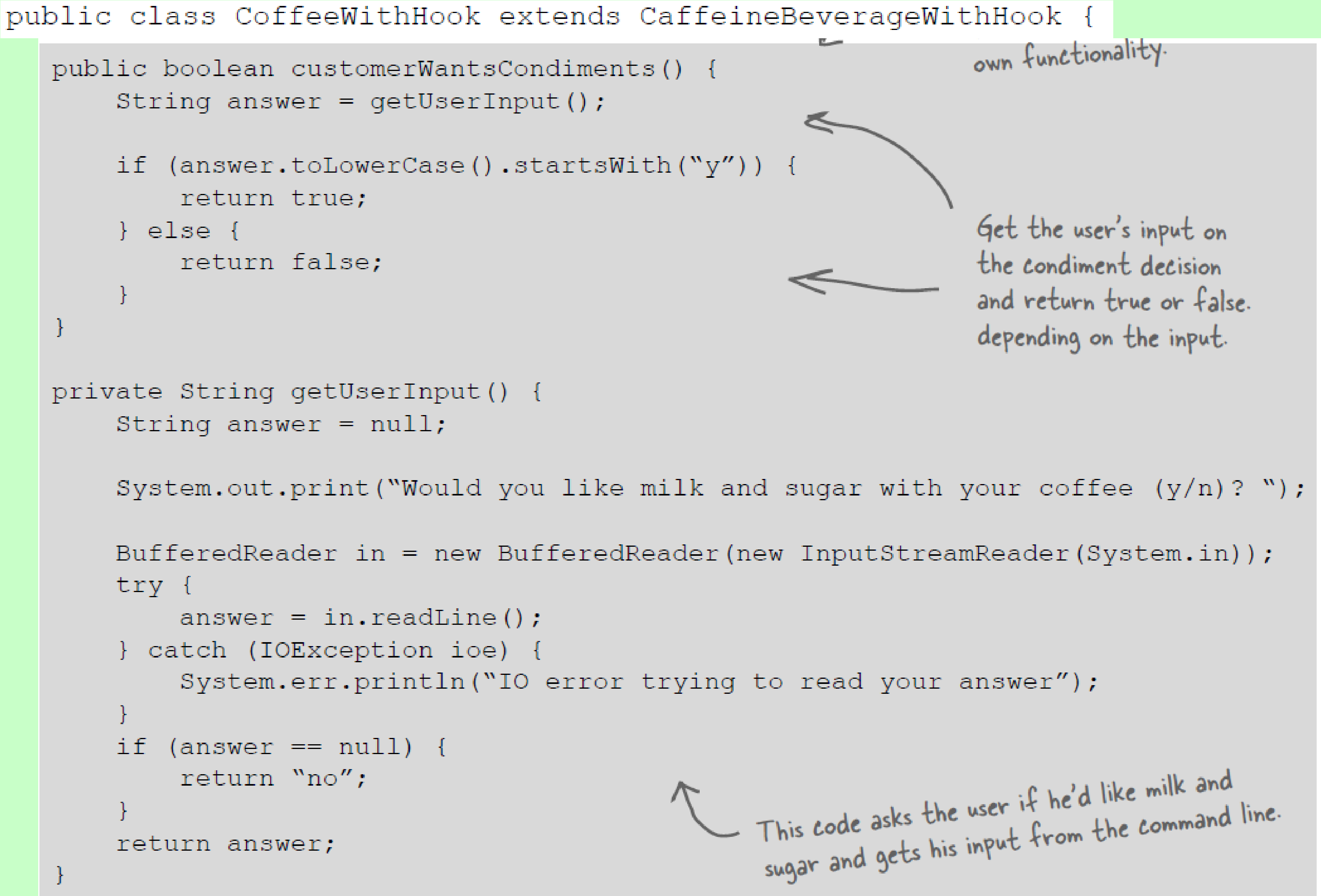
```
public abstract class CaffeineBeverageWithHook {

    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.

CoffeeWithHook



```
public class CoffeeWithHook extends CaffeineBeverageWithHook {
    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;
    }
```

own functionality.

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

This code asks the user if he'd like milk and sugar and gets his input from the command line.

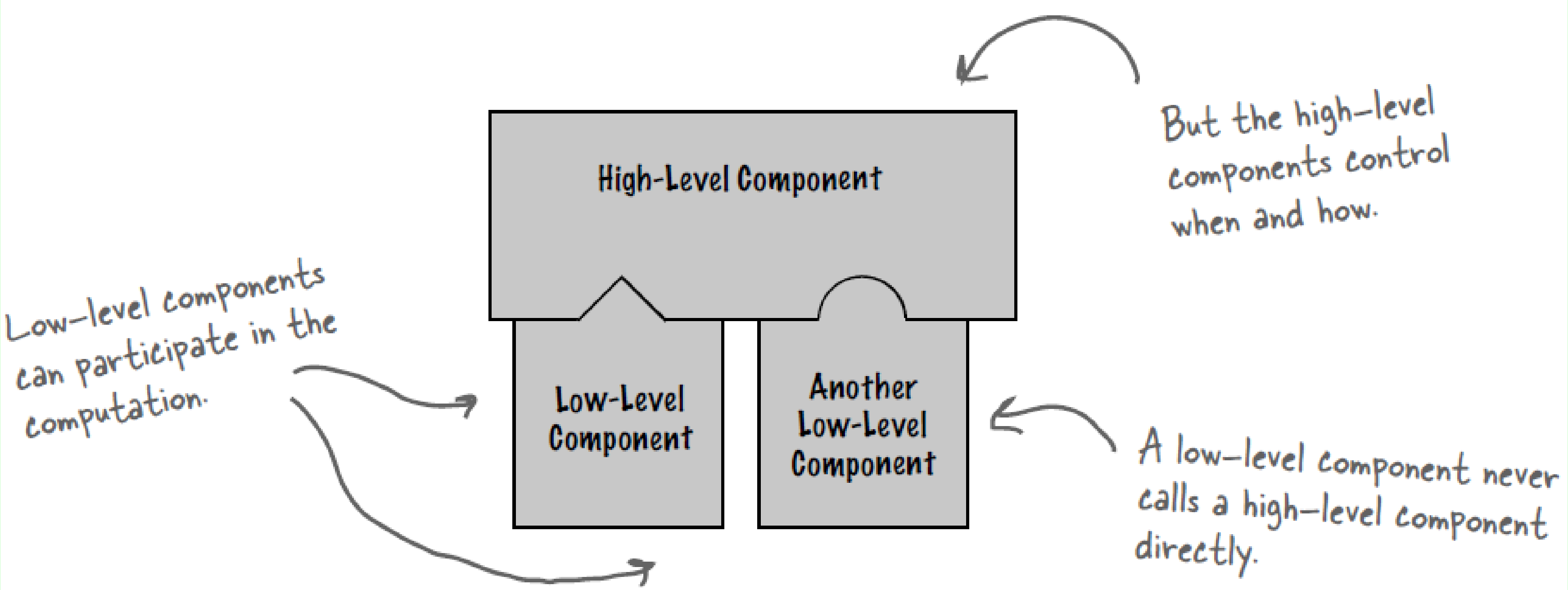
The Hollywood Principle

The Hollywood Principle

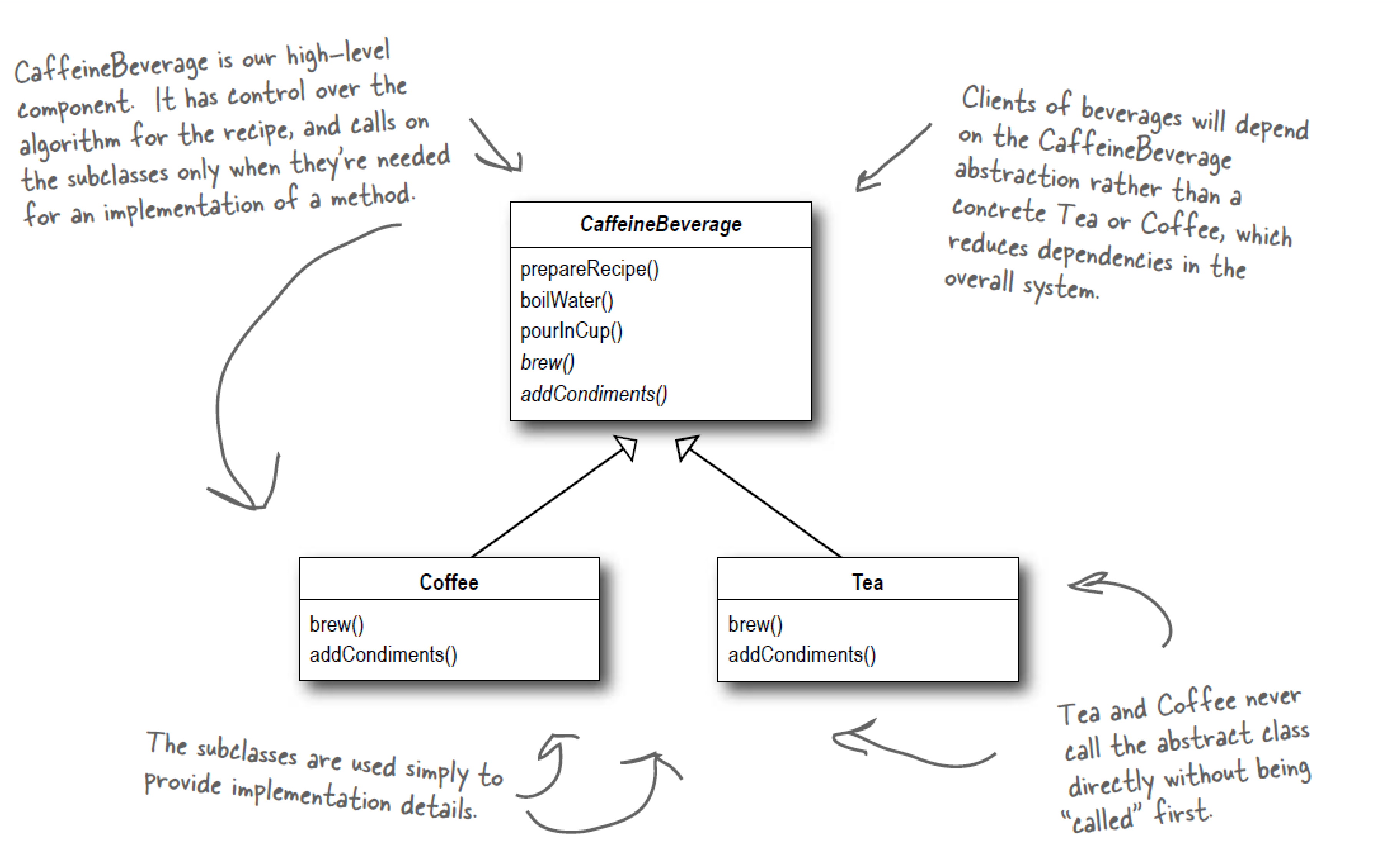
Don’t call us, we’ll call you.

- 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 a “don’t call us, we’ll call you” treatment.

The Hollywood Principle


High-Level Component
Low-Level Component
Another Low-Level Component
But the high-level components control when and how.
Low-level components can participate in the computation.
A low-level component never calls a high-level component directly.

The Hollywood Principle



CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.

CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()
brew()
addCondiments()

Coffee

brew()
addCondiments()

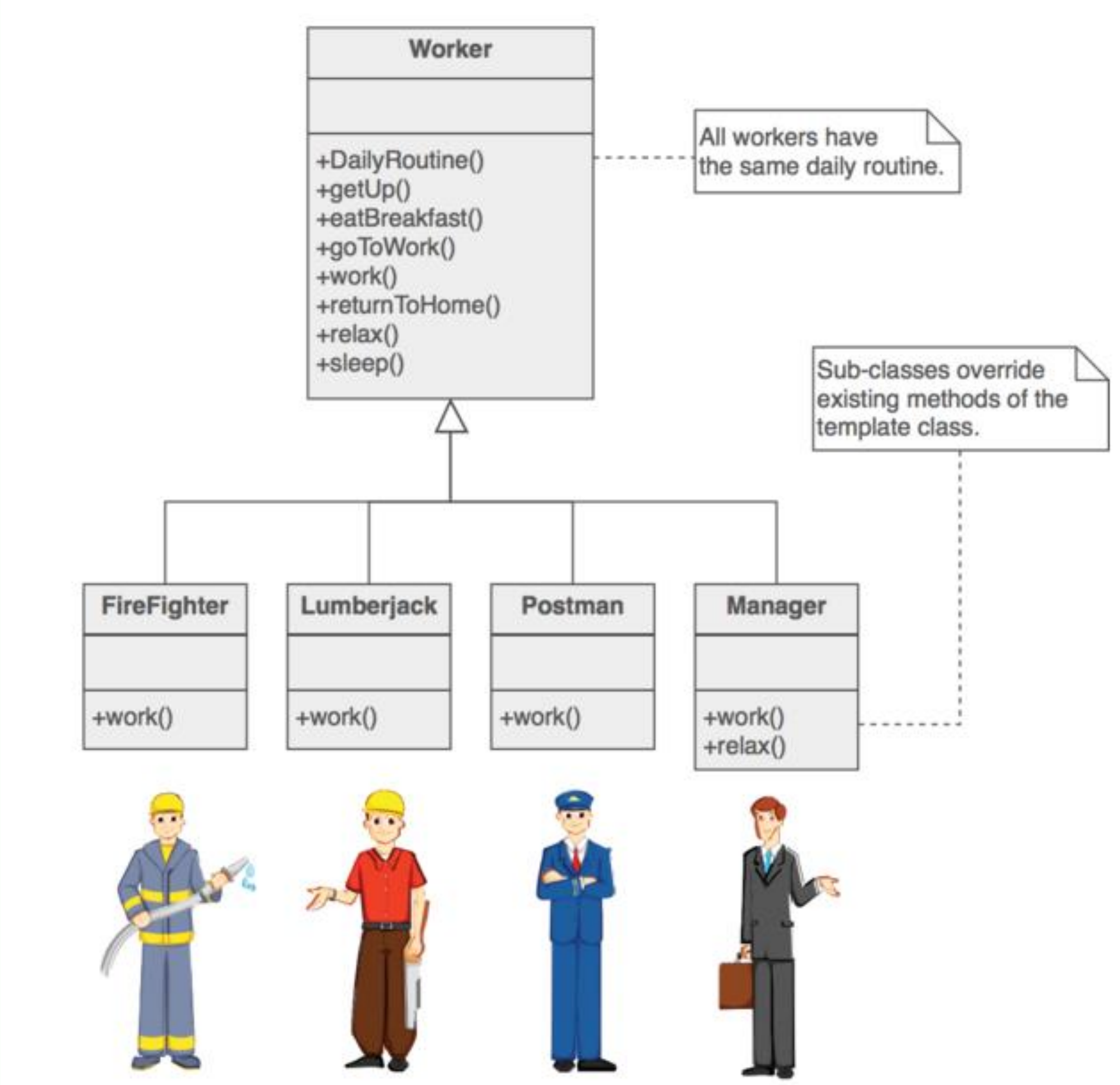
Tea

brew()
addCondiments()

The subclasses are used simply to provide implementation details.

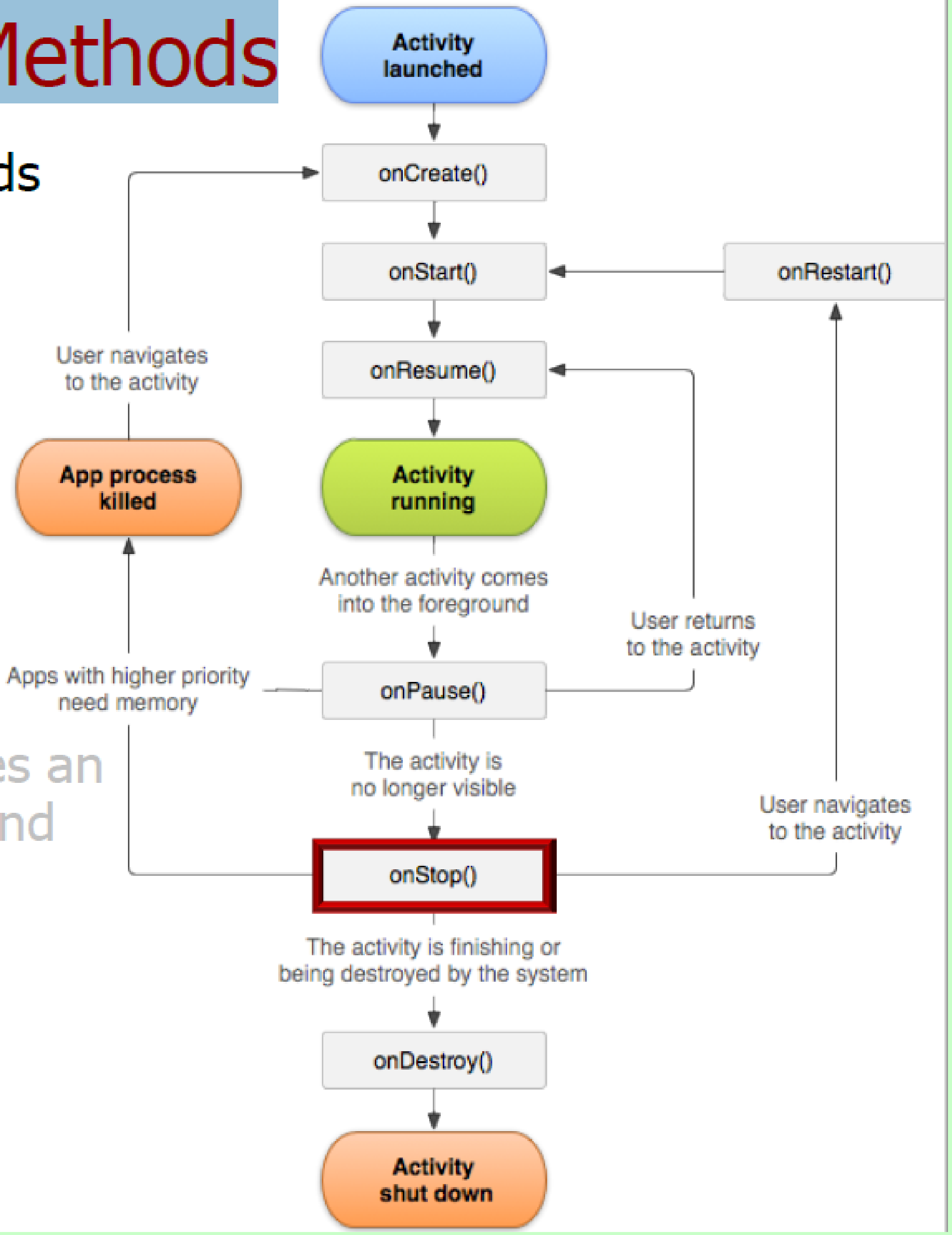
Tea and Coffee never call the abstract class directly without being "called" first.

Another Example



Activity Lifecycle Hook Methods

- The Android runtime calls hook methods on an Activity to control its lifecycle:
  - **onCreate()** – called to initialize an Activity when it is first created
  - **onStart()** – called when Activity is becoming visible to the user
  - **onResume()** – called when user returns to an Activity from another
  - **onPause()** – called when user leaves an Activity that's still visible in background
  - **onStop()** – called when user leaves an Activity for another


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

http://www.dre.vanderbilt.edu/~schmidt/cs282/PDFs/3-Activities.pdf

Template Method vs Strategy ?

Template Method vs Strategy

- Template Method uses inheritance to vary part of an algorithm.

- Strategy uses delegation to vary the entire algorithm.

When Would I Use This Pattern?

- The Template Method pattern is used when
  - When behaviour of an algorithm can vary, you let subclasses implement the behaviour through overriding
  - You want to avoid code duplication, implementing variations of the algorithm in subclasses
  - You want to control the point that subclassing is allowed.

When Would I Use This Pattern?

- Template Method may not be an obvious choice in the beginning, but the usual sign that you should use the pattern is when
  - You find that you have two almost identical classes working on some logic.
- At that stage, you should consider the power of the template method pattern to clean up your code.

Types of Methods in Parent Class

- When broken down, there are four different types of methods used in the parent class:
  - **Concrete methods** Standard complete methods that are useful to the subclasses. These methods are usually utility methods.
  - **Abstract methods** Methods containing no implementation that must be implemented in subclasses.
  - **Hook methods** Methods containing a default implementation that may be overidden in some classes. Hook methods are intended to be overridden, concrete methods are not.
  - **Template methods** A method that calls any of the methods listed above in order to describe the algorithm without needing to implement the details.

Summary

1. Examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes.
2. Define a new abstract base class to host the "don't call us, we'll call you" framework.
3. Move the shell of the algorithm (now called the "template method") and the definition of all standard steps to the new base class.
4. Define a placeholder or "hook" method in the base class for each step that requires many different implementations.

Summary

5. Invoke the hook method(s) from the template method.
6. Each of the existing classes declares an "is-a" relationship to the new abstract base class.
7. Remove from the existing classes all the implementation details that have been moved to the base class.
8. The only details that will remain in the existing classes will be the implementation details peculiar to each derived class.

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4