DZone

Revisiting the Template Method Design Pattern in Java

by Grzegorz Piwowarek MVB · Nov. 12, 18 · Java Zone · Tutorial

The conciseness of Java 8 lambda expressions sheds a new light on classic GoF design patterns. By leveraging functional programming, we can get the same benefits with much less coupling and ceremony – the Template method is a great example of that.

Classic GoF Template Method Implementation

The template method design pattern is one of 23 design patterns described by the Gang of Four – utilizing it makes it easy to conform to the Open-Closed and Hollywood principles.

Simply put, it facilitates defining a skeleton (algorithm invariants) of a certain algorithm where users can fill-in-the-blanks, which is achieved by overriding abstract methods exposed by an abstract class defining the skeleton implementation.

Getting more practical, imagine scenarios like logging execution time of some action, running your code within a transaction, or a classical JUnit workflow where we're responsible for filling in the blanks in form of before/after/test methods — those are scenarios where the pattern shines.

Let's have a look at a fairly simple example that surrounds our code with naive execution time logging.

Classically-trained GoF design pattern practitioners would implement the idea using abstract classes:

```
1  package com.pivovarit.template_method.gof;
2
3  import java.time.Duration;
4  import java.time.LocalTime;
5
6  abstract class AbstractTimeLoggingMethod {
7
8      abstract void run();
9
10     public void runWithTimeLogging() {
11         var before = LocalTime.now();
12         run();
13         var after = LocalTime.now();
```

```
        System.out.printf(
          "Execution took: %d ms%n",
          Duration.between(before, after).toMillis());
    }
}
```

And then, if we want to wrap our piece of code with the logging logic, we just need to extend the class and then use the public method:

```
public static void main(String[] args) {
    var fetchAndLog = new AbstractTimeLoggingMethod() {
        @Override
        void run() {
            findById(42);
        }
    };

    fetchAndLog.runWithTimeLogging(); // Execution took: 1005 ms
}
```

However, since it relies on inheritance, this approach is quite invasive – it tightly couples classes together and screams with excessive boilerplate (not even mentioning the fact that each subclass becomes a new **.class* file eventually).

Simplifying Template Method With Functions

Having new tools in our toolbox, we can achieve a similar effect using a lightweight version of the above without using inheritance by utilizing functional programming ideas. Since we can now pass functions around, why not do the same here?

So, instead of defining a skeleton by using an abstract class, we can do that in a single method that accepts a functional interface:

```
final class TemplateMethodUtil {

    private TemplateMethodUtil() {
    }

    static void runWithExecutionTimeLogging(Runnable action) {
        var before = LocalTime.now();
        action.run();
        var after = LocalTime.now();
        System.out.printf(
          "Execution took: %d ms%n",
          Duration.between(before, after).toMillis());
    }
```

```
}
```

And now, whenever we want to opt-in, it's enough to just call the util method to leverage the compile-time aspect'ish semantics:

```
TemplateMethodUtil.runWithExecutionTimeLogging(() -> findById(42));
```

Want to orchestrate multiple calls? Not a problem:

```
static void orchestrate(Runnable step1, Runnable step2) {
    System.out.println("starting...");
    step1.run();
    step2.run();
    System.out.println("ending...");
}
```

And in action:

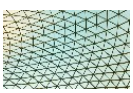
```
TemplateMethodUtil.orchestrate(
  () -> System.out.println("a"),
  () -> System.out.println("b"));

starting...
a
b
ending...
```

Summary

The GoF book is full of canonical ideas, but it's still worth revisiting them as new ways emerge that enable better implementations.

Code snippets can be found on GitHub.

Like This Article? Read More From DZone



Adapter Design Pattern With Scala

Java Lambda Expressions: Functions as First-Class Citizens

Topics: JAVA , TUTORIAL , TEMPLATE METHOD DESIGN , TEMPLATE METHOD , DESIGN PATTERNS , FUNCTIONAL PROGRAMMING , FP