Template Method

4

The pattern “Template Method” belongs to the behavior patterns and is quite catchy. It helps you when you need to implement an algorithm differently in subclasses, but on the other hand must not change it.

4.1 How Template Method Works

Take the classic order of input, processing, and output as an example. You want to input a string, convert it optionally to upper or lower case, and then output it to the console.

4.1.1 A First Approach

At first glance, the task seems to be solved quickly if you define two classes: `Uppercase` and `Lowercase`. I print the source code of `Uppercase` below. The rest can be found in the sample project Template_1. The method `run()` defines the algorithm: input, convert, output. The source code of the `Lowercase` class is identical. Only the bold part differs.

```
public class Uppercase {
    public final void run() {
        final var input = textEnter();
        final var converted = convert(input);
        print(converted);
    }
```


O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_4

```
    private String textEnter() {
        final var message = "Please enter the text:";
        return JOptionPane.showInputDialog(message);
    }

    private String convert(String input) {
        return input.toUpperCase();
    }

    private void print(String text) {
        System.out.println(text);
    }

    public static void main(String[] args) {
        new Uppercase().run();
    }
}
```

When a project has source code that is copy-and-pasted, it is usually an indication of poor design. It makes sense to move duplicate code to a common superclass-and that's exactly what you do in the following section.

4.1.2 The Second Approach

Now we build an abstract superclass that defines the algorithm on the one hand, and the methods that are identical in both classes on the other. Since the definition of the algorithm should be binding for all subclasses, it makes sense to declare the method `run()` final. You can find this code in the sample project Template_2.

```
public abstract class Input {
    public final void run() {
        var input = textEnter();
        var converted = convert(input);
        print(converted);
    }

    private final String textEnter() {
       return JOptionPane.showInputDialog("Please enter the text:");
    }

    protected abstract String convert(String input);

    private final void print(String text) {
        System.out.println(text);
    }
}
```

The subclasses `UppercaseConverter` and `LowercaseConverter` only override the abstract method `convert()`. As an example, I show the class `LowercaseConverter` here.

```
public class LowercaseConverter extends Input {
    @Override
    protected String convert(String input) {
        return input.toLowerCase();
    }
}
```

In the next section we will put this project into operation.

4.1.3 The Hollywood Principle

The client can now easily select the desired converter by instantiating it from the appropriate subclass:

```
public class Client {
    public static void main(String[] args) {
        Input input = new LowercaseConverter();
        input.run();
        Input newInput = new UppercaseConverter();
        newInput.run();
    }
}
```

The call is always made from the superclass. The algorithm is defined in the superclass and the superclass calls the corresponding actual functionality in the subclass. This procedure is what GoF calls the Hollywood principle: "Don't call us – we'll call you!"

Also note that at this point (highlighted in bold above), the declaration of the variable `input does` NOT work with a `var.` We want to have a local variable of the superclass type here, so that we can then assign instances of the various subclasses as we wish.

If you use var. at this point, the compiler recognizes the type LowerCaseConverter and assigns it to the variable. Until then everything is ok. But if you later want to build an UppercaseConverter with the same variable, you run into an error. Fortunately, however, already at compile time.

4.1.4 Introducing Hook Methods

You can extend the project with hook methods. A hook method is a method that can optionally be overridden. A default behavior – or no behavior at all – is defined in the superclass. The subclasses can – but do not have to – adopt this behavior. In the Template_3 sample project, there is such a hook method. The `save()` method returns whether the entered text should be saved to disk. In this case the method `saveToDisk()` is called. By default `false` is returned.

```
public abstract class Input {
    public final void run() {
        var input = textEnter();
        var converted = convert(input);
        print(converted);
        if (save())
            saveToDisk();
    }

    // … abridged

    protected boolean save() {
        return false;
    }

    private void saveToDisk() {
        System.out.println("Input saved");
    }
}
```

If the subclasses feel that the default behavior doesn't fit this way, they are free to override it. For example, the `UppercaseConverter` wants the text to always be saved – so it returns `true`:

```
public class UppercaseConverter extends Input {
    // … abridged

    @Override
    protected boolean save() {
        return true;
    }
}
```

The LowercaseConverter wants to let the user decide whether to save the text:

```
public class LowercaseConverter extends Input {
    // … abridged

    @Override
    protected boolean save() {
        var question = "Should the text be saved?";
        var answer =
           JOptionPane.showConfirmDialog(null, question);
        return answer == JOptionPane.YES_OPTION;
    }
}
```

Hook methods are also called insertion methods. They offer the possibility to influence the flow of the algorithm.

That's it – now you know the Template Method Pattern. Despite – or perhaps because of – its seemingly trivial formulation, the Template Method Pattern is extremely important for the development of frameworks such as the Java class library. You develop a basic framework, the algorithm, and build your extensions on it.

4.2 The "ListModel" Interface

You will be able to identify the Template Method Pattern wherever you encounter abstract classes. At this point, I'd like to briefly discuss the algorithm that the `ListModel` interface specifies; it describes what must happen and what data must be present for a `JList` to display data. A `JList` accepts any object as a data model that is of type `ListModel`. The `ListModel` interface prescribes four methods that are required to display data. A `JList` must be able to register and deregister with the model as an observer, a `ListDataListener`; to do this, the `addListDataListener()` and `removeListDataListener()` methods must be implemented. The `JList` must also be able to ask the database how many items to display; this is answered by the `getSize()` method. And finally, the `JList` must be able to query the element at a specific location; for this, there is the `getElementAt()` method, which you pass the index of the element you are looking for.

In the directory of this chapter, you will find the sample project ListModel_Example. This project demonstrates the procedure with a very simple example. If you analyze its source code, you will see that the methods for registering and deregistering listeners are so general that they can probably be used in all situations.

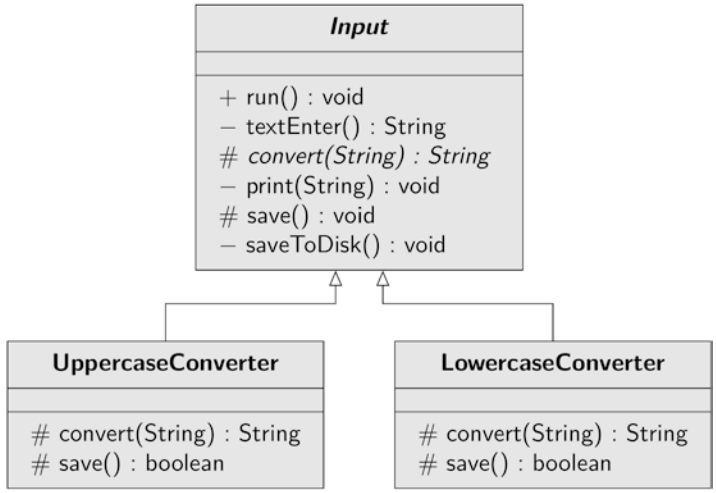

Fig. 4.1 UML diagram of the Template Method Pattern (example project Template_3)

In the class library there is the `AbstractListModel` class in which these two methods are implemented. If you need a different implementation than the default, you are free to override these methods. Methods that describe the database, that is, `getSize()` and `getElementAt()`, escape a standard implementation. You must always define these by yourself. Please have a look at the AbstractListModel_Example project again.

4.3 Template Method – The UML Diagram

The UML diagram of the Template Method Pattern for the example project Template_3 can be found in Fig. 4.1.

4.4 Summary

Go through the chapter again in key words:

- You define a specific algorithm,
- Parts of the algorithm can be executed by the class itself; other parts are prescribed to the subclasses that implement the non-abstract parts,
- You describe the algorithm in a final method of the abstract superclass,
- The subclass or subclasses override the abstract methods of the superclass,
- Optionally, hook methods – insertion methods – can be overwritten,
- Hook methods allow you to partially vary the algorithm.

4.5 Description of Purpose

The Gang of Four describes the purpose of the Template Method pattern as follows:

> Define the skeleton of an algorithm in an operation and delegate individual steps to subclasses. Using a template method allows subclasses to override specific steps of an algorithm without changing its structure.