Software Design Pattern: Template Method

Anton Seitz
INF22B
DHBW
Stuttgart, Germany
inf22052@lehre.dhbw-stuttgart.de

Jannis Gehring
INF22B
DHBW
Stuttgart, Germany
inf22115@lehre.dhbw-stuttgart.de

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.**

Contents

I. Motivation

In software development, many problems arise from the need to manage processes that share a common structure but vary in detail. For example, consider scenarios where multiple algorithms or workflows follow the same sequence of steps but require customization for specific use cases. Ensuring consistency in such processes often leads to code duplication, which complicates maintenance and increases the likelihood of errors.

When developers attempt to address these issues using conditional logic (e.g., `if` or `switch` statements), the result is often code that is harder to understand, modify, and extend. As the number of variations grows, these conditionals become unwieldy, making the codebase fragile and less reusable.

Another challenge lies in maintaining *flexibility* without sacrificing *control*. Certain processes demand a fixed order of operations - for instance, initializing resources, executing a task, and cleaning up afterward - but may require variation in how individual steps are implemented. Developers must balance enforcing a standardized sequence with providing enough adaptability to accommodate different requirements.

How can we structure code to manage such recurring problems effectively? The solution lies in a design approach that enforces the overall sequence of operations while allowing variation in specific steps - a balance that promotes reusability, consistency, and scalability.

II. The Template Method Design Pattern (TMDP)

With the Template Method Design Pattern (TMDP), the algorithm is divided into several abstract steps, which are called in sequence in the *template method*. This method implements the solution to the problem on a high level, allowing the different, specific step implementations to deal with the differing contexts. The steps as well as the template method are aggregated in an abstract class.

When the template method is to be applied to a new context, a sub-class of the abstract class is created. Then, at minimum all required functions declared in the abstract class have to be implemented, before the template method can be performed for the new class.

A. Different types of abstract steps

One differentiates between different types of methods in the template method. The number of different types of methods and the naming varies ([1] and [2] count 2, [3] and [4] 3), but the underlying concepts of *abstraction* and *default implementation*. We count three types of abstract steps in the template method [[3]]:

1) **Required abstract steps**
   These steps are defined as abstract methods in the abstract class. Therefore, they have to be implemented by every concrete class, otherwise the template method cannot be performed.
2) **Optional abstract steps**
   For these steps, a default implementation exists in the abstract class. Therefore, they do not have to be implemented by the concrete class but can be if the context requires so. This can be useful when steps are *exactly* the same for multiple contexts.
3) **Hook methods**
   For these steps, no implementation is given in the abstract class, but they are not abstract classes either. Instead, they can be implemented in concrete classes for them to *hook into* the template method. One example would be the allowance of logging in between of different steps of the Template Method.

B. Class diagram

The following class diagram displays the different classes and the different types of abstract steps. Whilst the

functions `required_step1` and `required_step2` are implemented by every concrete classes, `optional_step` and `hook_method` are only implemented by some of them.

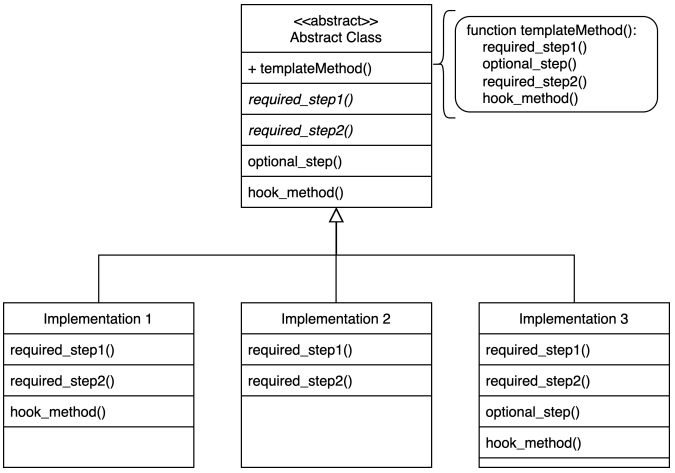

Fig. 1: Class diagram showcasing the different concepts of Template Method Design Pattern (TMDP)

III. Example

A. General project information

For showcasing TMDP, a context had to be chosen which fits the situation of similar algorithms with differing concrete implementations. The implemented application, in short, fetches, processes and displays data of differing source and contexts. The different contexts are presented subsequently:

1) **`CryptoVisualize` - Price of Bitcoin**
   The price of the wide-spread crypto currency Bitcoin is retrieved for every day since the first of July 2024. The data is then being transformed and visualized as a plotly line-chart.
2) **`DogVisualize` - Picture of Dog**
   A random picture of a dog is being fetched and displayed.
3) **`AutobahnVisualize` - German Highway lorries**
   Informations regarding different lorries of the notorious german highways ('Autobahn') are fetched. Then, the lorries are displayed on a plotly map-chart. They are color-coded according to the highway they belong to, which allows the retracing of individual highways.

B. Project Setup

For implementing TMDP we used the programming language Python. Python is easy to read and allows to focus on the main concepts of this demonstration. The module `abc` (Abstract Base Classes) provides the necessary classes/decorators for implementing TMDP. The project is structured in three folders:

- `src` contains the source code for the TMDP. `model.py` implements the class structure, including the concrete classes. `args.py` deals with the command-line arguments and `main.py` brings both together to a working program
- `tests` contains the tests
- `doc` contains the files for this documentation.

The project can be setup by the following steps:

1) `sh python3 -m venv .venv`
2) `sh source .venv/bin/activate`
3) `sh pip install -r requirements.txt`

C. Usage

To execute the program, run `src/main.py`. The kind of visualization displayed can be configured by a command line argument:

```
python3 src/main.py -c [specifier]
```

These are the specifiers for the different classes:

context	specifier
`CryptoVisualize`	`crypto`
`DogVisualize`	`dog`
`AutobahnVisualize`	`autobahn`

Example outputs with their corresponding commands are presented subsequently.

Fig. 2: Command: `python3 src/main.py -c crypto`

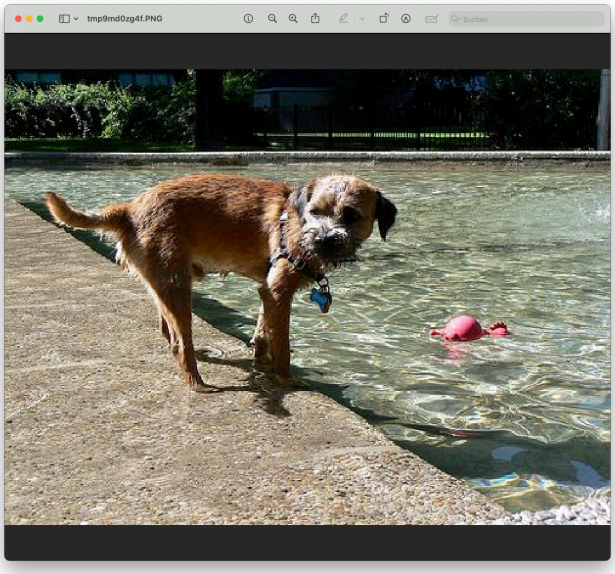

Fig. 3: Command: `python3 src/main.py -c dog`

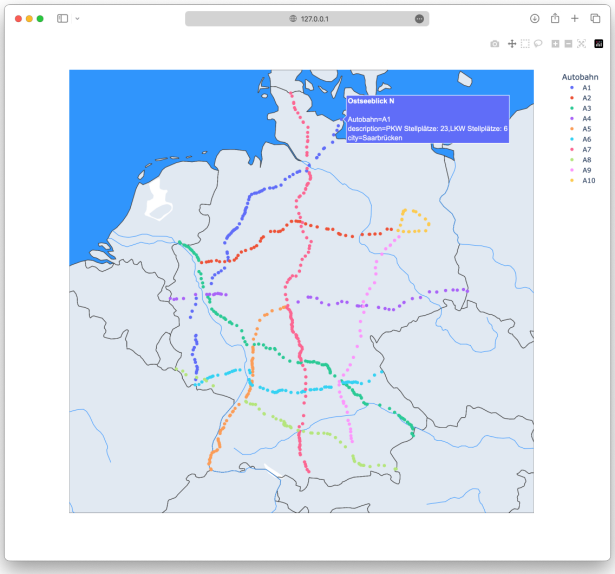

Fig. 4: Command: `python3 src/main.py -c autobahn`

a) *Tests:*
Tests of the program have been written using the python package `pytest`. They can be found in `tests/test.py` and run by

```
pytest tests/test.py -v
```

.

D. The TMDP in the example

a) *template method:*
The template method implements the basic fetch-process-visualize algorithm:

```
# template method
def show_me_stuff(self) -> None:
    api_url = self.get_api_url()
    content = self.api_requests(api_url)
    data = self.process_content(content)
    self.print_report(data)
    self.visualize_data(data)
    return
```

Listing 3: The example template method.

b) *Required abstract steps:*
`get_api_url` and `visualize_data` are different for every class and therefore implemented as abstract methods. In general, default implementations of functions should be given if possible to increase ease of use and standardization. Both functions, however, have no default implementation because they differ *for every concrete class*. In python, this can be implemented by a decorator imported from `abc` and an empty body:

```
@abstractmethod
def get_api_url(self) -> str:
    pass
```

Listing 4: Example of required abstract step in implementation.

c) *Optional abstract steps:*
`api_requests` and `process_content` have default implementations. For `api_requests`, it is just performing one api request and returning the content; for `process_content`, it is returning the input. These functions are implemented without additional decorators and with a proper function body:

```
def process_content(self, content):
    return content
```

Listing 5: Example of optional abstract step in implementation.

d) *Hook methods:* `print_report` is a hook method allowing the developer to print informations before visualizing the data. `CryptoVisualize` is the only class utilizing this method. It is implemented as a normal function with an empty body:

```
def print_report(self, data):
    pass
```

Listing 6: Example of hook method in implementation.

E. Further concepts of TMDP in the example

The implementation of `api_requests` in `AutobahnVisualize` is to be highlighted, because it is the only class overwriting the default implementation. This is due to the fact, that the first API request only returns the names of the highways and not the lorries itself. The default implementation is rather general, which is why `AutobahnVisualize.api_requests()` can refer to it when making requests. This approach adapts to the API

endpoints whilst still ensuring a standardized handling of API(-error)s:

```
  def api_requests(self, api_url):
     content_highways = (
        super()
        .api_requests(api_url)
        ["entries"][:10]
     )
     all_lorries = []
     for highway in content_highways:
           url = f"https://api.deutschland-
api.dev/autobahn/{highway}/parking_lorry?
field"
        lorries = (
           super()
           .api_requests(url)
           ["entries"]
        )
        all_lorries.append(lorries)
     return all_lorries
```

Listing 7: The overwritten version of `api_requests` in `AutobahnVisualize`.

IV. Real-World examples

1) **Testing Frameworks (JUnit)**
   JUnit uses template method principles to enforce a structured flow in test execution. Methods such as `setUp()` and `tearDown()` serve as hooks, while `executeTest()` is a required step for specific test logic.
2) **Game Development (Unreal Engine)**
   In Unreal Engine, the `Actor` class exemplifies the template method pattern. Hooks such as `BeginPlay()` and `Tick()` allow developers to inject custom logic while adhering to the engine's game loop.
3) **Oatmeal Preparation (Non-programming)**
   [2] provides an analogy of oatmeal preparation, illustrating how a fixed sequence of steps (gather ingredients, prepare, cook, and serve) can have flexibility in implementation depending on factors like the type of oatmeal and the cooking method (e.g., stovetop or microwave). This mirrors the template method's ability to define an overarching structure while allowing subclasses to handle details [[2], pp. 247-251].

V. Pro/Con of template method

The template method pattern offers both benefits and drawbacks:

A. *Advantages*

- **Reusability**
  Encourages code reuse by defining a reusable algorithmic structure applicable across various contexts [[2], pp. 249-250].
- **Scalability**
  New behaviors can be introduced by subclassing without altering the existing template, adhering to the open-closed principle [[2], p. 249].
- **Consistency**
  Enforces a standardized process flow, enhancing readability and maintainability [[2], pp. 250-251].
- **Reduced Duplication**
  Minimizes repeated code by encapsulating shared behavior in the abstract base class [[2], pp. 247-248].

B. *Disadvantages*

- **Inheritance Dependency**
  Relying on inheritance can limit flexibility and lead to tightly coupled designs, which [2] notes is a potential drawback of class-based patterns [[2], p. 252].
- **Complexity for Simple Use Cases**
  For straightforward problems, introducing an abstract template may introduce unnecessary complexity [[2], pp. 252-253].
- **Global Impact of Changes**
  Modifications in the base class can inadvertently propagate to all subclasses, introducing potential bugs [[2], p. 253].

VI. Differentiation to Other Patterns

While the template method shares similarities with other design patterns, its unique characteristics set it apart:

1) **Strategy Pattern**
   The Strategy pattern encapsulates algorithms as interchangeable objects, offering runtime flexibility. In contrast, the template method fixes the algorithm's structure at compile time, delegating specific steps to subclasses [[2], pp. 250-251].
2) **Factory Method**
   Both patterns rely on abstract methods, but their goals differ: the Factory Method creates objects, while the template method defines an algorithmic workflow [[2], pp. 249-250].

VII. Outlook

While the template method remains a robust tool for algorithm design, its reliance on inheritance may be less favored in modern design paradigms emphasizing composition and functional programming [[2], pp. 253-254]. Nonetheless, its ability to enforce structure in repetitive workflows ensures its continued relevance in certain contexts.

References

[1] "Template method pattern." [Online]. Available: https://en.wikipedia.org/wiki/Template_method_pattern
[2] J. E. McDonough, "Template Method Design Pattern," in *Object-Oriented Design with ABAP: A Practical Approach*, Berkeley, CA: Apress, 2017, pp. 247–254. doi: 10.1007/978-1-4842-2838-8_19.
[3] "Template Method." [Online]. Available: https://refactoring.guru/design-patterns/template-method
[4] J. W. Cooper, "Java design patterns: a tutorial," 2000.