Template Method Design Pattern

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing the overall structure.

Customer Requirement:

Imagine you are developing a data processing framework where different types of data need to be processed. The steps to process the data are similar across different types, but some specific steps vary based on the data type. The customer wants to ensure that the common steps are handled consistently, while allowing for customization of specific steps.

Solution: Template Method Pattern

The Template Method Pattern fits perfectly here. It allows you to define the structure of the data processing algorithm in a base class, while letting subclasses customize specific steps. This ensures that the overall process remains consistent, while also being flexible.

Components of the Template Method Pattern:

1. Abstract Class (DataProcessor): Defines the template method (processData) that outlines the steps of the algorithm. Implements the steps that are common across all types of data.
2. Concrete Classes (TextDataProcessor, CSVDataProcessor): Implement the abstract methods defined by the abstract class. Provide specific implementations for the steps that vary.

Example: Data Processing Framework

1. Abstract Class (DataProcessor):

```
public abstract class DataProcessor {
    public final void processData() {
        readData();

        processData();

        writeData();

    }

    private void readData() {

        System.out.println('Reading data...');

    }

    private void writeData() {

        System.out.println('Writing data...');

    }

    protected abstract void processData();

}
```

2. Concrete Classes (TextDataProcessor, CSVDataProcessor):

```
public class TextDataProcessor extends DataProcessor {

    @Override

    protected void processData() {

        System.out.println('Processing text data...');

    }

}



public class CSVDataProcessor extends DataProcessor {

    @Override

    protected void processData() {

        System.out.println('Processing CSV data...');

    }

}
```

3. Client Code:

```
public class TemplateMethodPatternDemo {
    public static void main(String[] args) {
        DataProcessor textProcessor = new TextDataProcessor();
        textProcessor.processData();

        DataProcessor csvProcessor = new CSVDataProcessor();
        csvProcessor.processData();
    }
}
```

How It Works:

- Template Method (processData): This method in the DataProcessor class defines the skeleton of the data processing algorithm. It consists of three steps: reading data, processing data, and writing data.

Concrete Steps (TextDataProcessor, CSVDataProcessor): These classes provide specific implementations of the processData method. The readData and writeData steps are common and handled by the abstract class, while the processData step is customized by each subclass.

Why It Is a Behavioral Design Pattern:

The Template Method Pattern is a behavioral pattern because it defines the behavior of the algorithm in the form of a method template. It allows different behaviors (the concrete implementations) to be defined by subclasses while ensuring that the overall process remains consistent.

Benefits of the Template Method Pattern:

1. Code Reusability: The common parts of the algorithm are implemented in the abstract class, promoting code reuse.

2. Flexibility: The pattern allows subclasses to override certain steps of the algorithm, providing flexibility in behavior.

3. Consistency: By defining the structure of the algorithm in a template method, the pattern ensures consistency in how the algorithm is executed across different subclasses.

When to Use the Template Method Pattern:

- When you have an algorithm with steps that need to be defined by different subclasses, but the overall structure of the algorithm should remain the same.
- When you want to enforce a certain order of execution for an algorithm's steps while allowing some of these steps to be customizable by subclasses.
- When you want to reduce code duplication by moving common behavior to a superclass and leaving the specifics to the subclasses.