Template Method Design Pattern

Last Updated : 03 Jan, 2025

The Template Method Design Pattern is a behavioral design pattern that provides a blueprint for organizing code, making it flexible and easy to extend. With this pattern, you define the core steps of an algorithm in a method but allow subclasses to override specific steps without changing the overall structure. Think of it like setting up a recipe: the main steps stay the same, but you can tweak parts to add unique flavors.

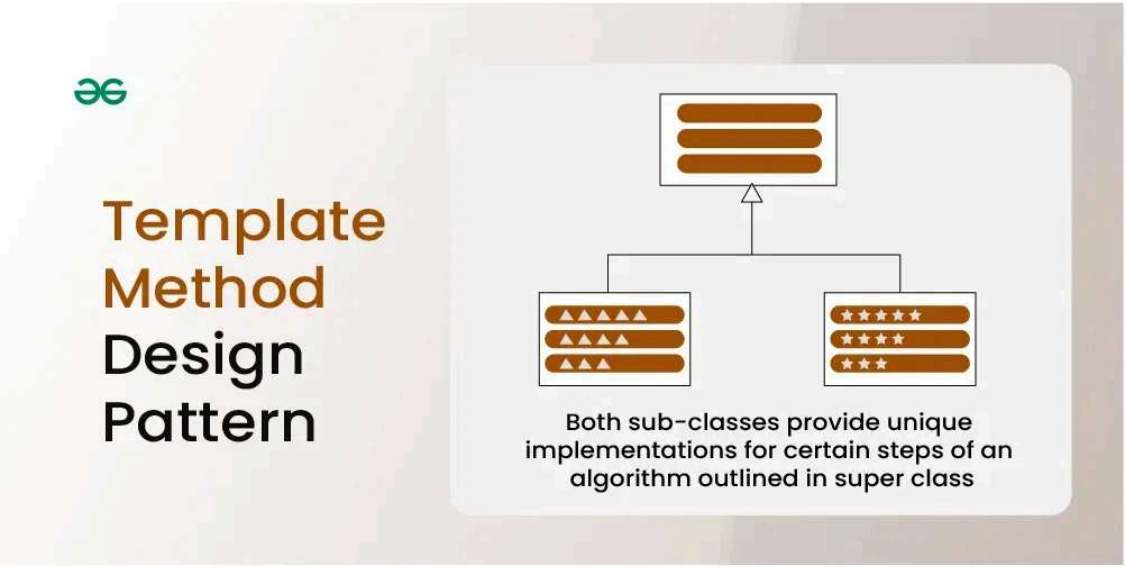


Table of Content

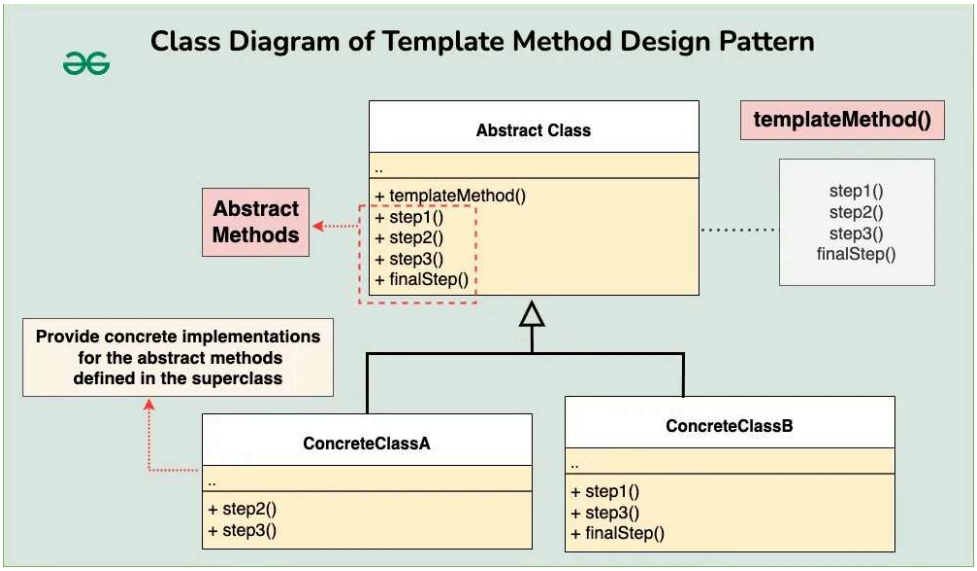
What is the Template Method Design Pattern?

The Template Method pattern is a behavioral design pattern that defines the skeleton of an algorithm or operations in a superclass (often abstract) and

leaves the details to be implemented by the child classes. It allows subclasses to customize specific parts of the algorithm without altering its overall structure.

- The overall structure and sequence of the algorithm are preserved by the parent class.
- Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps.
- These steps can be an abstract method that will be implemented by its subclasses.

Components of Template Method Design Pattern



- **Abstract Class (or Interface):**
    - This is the superclass that defines the template method. It provides a skeleton for the algorithm, where certain steps are defined but others are left abstract or defined as hooks that subclasses can override.
    - It may also include concrete methods that are common to all subclasses and are used within the template method.
- **Template Method:**
    - This is the method within the abstract class that defines the overall algorithm structure by calling various steps in a specific order.

    - It's often declared as final to prevent subclasses from changing the algorithm's structure.
    - The template method usually consists of a series of method calls (either abstract or concrete) that make up the algorithm's steps.
- **Abstract (or Hook) Methods:**
    - These are methods declared within the abstract class but not implemented.
    - They serve as placeholders for steps in the algorithm that should be implemented by subclasses.
    - Subclasses must provide concrete implementations for these methods to complete the algorithm.
- **Concrete Subclasses:**
    - These are the subclasses that extend the abstract class and provide concrete implementations for the abstract methods defined in the superclass.
    - Each subclass can override certain steps of the algorithm to customize the behavior without changing the overall structure.

How to Implement Template Method Design Pattern?

Let us see how to implement the Template Method Design Pattern in these simple steps:

- **Step 1: Create an Abstract Class**: Start by making a base or abstract class that defines the overall structure of the algorithm. This class will have a template method to outline the steps.
- **Step 2: Define the Template Method**: Inside the abstract class, create a method (the template method) that calls each step of the algorithm in a specific order.
- **Step 3: Implement Core Steps**: For each step of the algorithm, create individual methods in the abstract class. Some methods can have default implementations, while others can be abstract to allow customization.
- **Step 4: Create Subclasses**: Now, create subclasses that inherit from the abstract class. In each subclass, override the steps that need specific behavior, leaving the rest as they are.

- **Step 5: Use the Template Method**: When you run the template method on a subclass instance, it will execute all the steps in the defined order, with customizations in place from the overridden methods.

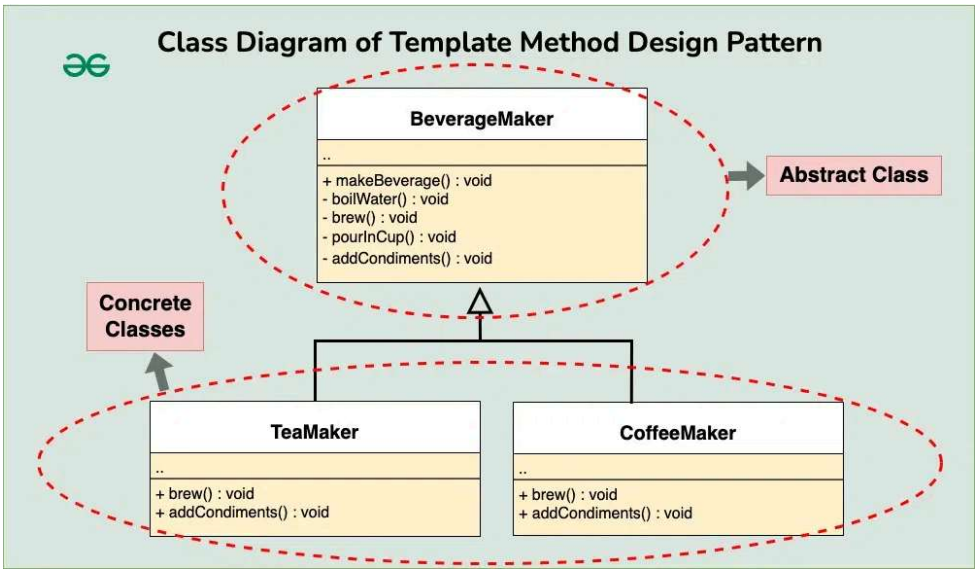
Template Method Design Pattern example (with implementation)

Problem Statement:

Let's consider a scenario where we have a process for making different types of beverages, such as tea and coffee. While the overall process of making beverages is similar (e.g., boiling water, adding ingredients), the specific steps and ingredients vary for each type of beverage.

Benefits of using Template Method Design Pattern in this scenario

- Using the Template Method pattern in this scenario allows us to define a common structure for making beverages in a superclass while allowing subclasses to customize specific steps, such as adding ingredients, without changing the overall process.
- This promotes code reuse, reduces duplication, and provides a flexible way to accommodate variations in beverage preparation.



Below is the implementation for the above problem statement using Template Design Pattern

Let's break down into the component wise code:

1. Abstract Class

```
// Abstract class defining the template method
abstract class BeverageMaker {
    // Template method defining the overall process
    public final void makeBeverage() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    // Abstract methods to be implemented by subclasses
    abstract void brew();
    abstract void addCondiments();

    // Common methods
    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

2. Concrete Class (TeaMaker)

```
// Concrete subclass for making tea
class TeaMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }

```

```
    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}
```

3. Concrete Class (CoffeeMaker)

```
// Concrete subclass for making coffee
class CoffeeMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}
```

Complete code for the above example

Below is the complete code for the above example:

```
// Abstract class defining the template method
abstract class BeverageMaker {
    // Template method defining the overall process
    public final void makeBeverage() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

```

```
    // Abstract methods to be implemented by subclasses
    abstract void brew();
    abstract void addCondiments();

    // Common methods
    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

// Concrete subclass for making tea
class TeaMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}

// Concrete subclass for making coffee
class CoffeeMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void brew() {
        System.out.println("Dripping coffee through
filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

```

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Making tea:");
        BeverageMaker teaMaker = new TeaMaker();
        teaMaker.makeBeverage();

        System.out.println("\nMaking coffee:");
        BeverageMaker coffeeMaker = new CoffeeMaker();
        coffeeMaker.makeBeverage();
    }
}
```

```
Making tea:
Boiling water
Steeping the tea
Pouring into cup
Adding lemon

Making coffee:
Boiling water
Dripping coffee through filter
Pouring into cup
Adding sugar and milk
```

Communication flow of the above example implementation:

- **Client Interaction**: Imagine you're the person who wants to make a hot beverage, so you decide whether you want tea or coffee.
- **Template Method Execution**: You follow a predefined set of steps to make your chosen beverage. These steps are outlined in a recipe book (abstract class) that you have.
- **Execution Flow within Template Method**: You start by boiling water, pouring it into a cup, then you add your specific ingredients depending on whether you're making tea or coffee. These steps are part of the recipe (template method).
- **Subclass Implementation**: You decide to make tea, so you follow the tea recipe (subclass). In this recipe, instead of adding coffee grounds, you steep

a tea bag and add lemon.

- **Method Overrides**: When you add lemon to your tea, you're customizing that step of the recipe. In programming, this is similar to overriding a method, where you provide your own implementation of a step.
- **Inheritance and Polymorphism**: You can use the same recipe book (abstract class) to make different beverages (concrete subclasses), whether it's tea or coffee. This is because the recipes (methods) are inherited from the abstract class.

When to use the Template Method Design Pattern?

- **Common Algorithm with Variations**: When an algorithm has a common structure but differs in some steps or implementations, the Template Method pattern can help subclasses customize specific parts while encapsulating the common phases in a superclass.
- **Code Reusability**: By specifying the common steps in one place, the Template Method design encourages code reuse when you have similar tasks or processes that must be executed in several contexts.
- **Enforcing Structure**: It's useful when you want to provide some elements of an algorithm flexibility while maintaining a particular structure or set of steps.
- **Reducing Duplication**: By centralizing common behavior in the abstract class and avoiding duplication of code in subclasses, the Template Method pattern helps in maintaining a clean and organized codebase.

When not to use the Template Method Design Pattern?

- **When Algorithms are Highly Variable**: Using the Template Method pattern might not be appropriate if the algorithms you're dealing with have a lot of differences in their steps and structure and little in common. This is because it could result in inappropriate abstraction or excessive complexity.
- **Tight Coupling Between Steps**: The Template Method pattern might not offer enough flexibility if there is a close coupling between the algorithm's parts, so modifications to one step require modifications to other steps.
- **Inflexibility with Runtime Changes**: Because the Template Method pattern depends on predetermined structure and behavior, it might not be the best

option if you expect frequent changes to the algorithm's stages or structure at runtime.

Comment More info Advertise with us

Next Article

Visitor design pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented. Creational...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

3. Types of Behvioural Patterns

Chain of Responsibility Design Pattern

The Chain of Responsibility design pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass...

10 min read

Command Design Pattern

The Command Design Pattern is a behavioral design pattern that turns a request into a stand-alone object called a command. With the help of this pattern, you can capture each component of a request, including th...

10 min read

Interpreter Design Pattern

The Interpreter Design Pattern is a behavioral design pattern used to define a language's grammar and provide an interpreter to process statements in that language. It is useful for parsing and executing...

10 min read

Mediator Design Pattern

The Mediator Design Pattern simplifies communication between multiple objects in a system by centralizing their interactions through a mediator. Instead of objects interacting directly, they communicate via a mediato...

7 min read

Memento Design Pattern

The Memento Design Pattern is a behavioral pattern that helps save and restore an object's state without exposing its internal details. It is like a "snapshot" that allows you to roll back changes if something goes...

6 min read

Observer Design Pattern

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When one object (the subject) changes state, all its dependents (observers) are notified...

8 min read

State Design Pattern

The State design pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. It achieves this by encapsulating the object's behavior within different state...

11 min read

Strategy Design Pattern

The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to switch algorithms dynamically without altering the code structure. Tabl...

11 min read

Template Method Design Pattern

The Template Method Design Pattern is a behavioral design pattern that provides a blueprint for organizing code, making it flexible and easy to extend. With this pattern, you define the core steps of an algorithm in a...

8 min read

Visitor design pattern

An object-oriented programming method called the Visitor design pattern makes it possible to add new operations to preexisting classes without changing them. It improves the modularity and maintainability of...

7 min read

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages

GeeksforGeeks
Sanchhaya Education Private Limited

Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:
K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

Advertise with us

Software Testing

DevOps Roadmap

System Design

High Level Design

Low Level Design

UML Diagrams

Interview Guide

Design Patterns

OOAD

System Design Bootcamp

Interview Questions

Inteview Preparation

Competitive Programming

Top DS or Algo for CP

Company-Wise Recruitment Process

Company-Wise Preparation

Aptitude Preparation

Puzzles

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Commerce

World GK

GeeksforGeeks Videos

DSA

Python

Java

C++

Web Development

Data Science

CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved