### Template Design Pattern [Layman’s View]

**Layman's Explanation**

Imagine you are writing a cookbook for making different types of cakes. Each recipe has steps that are common to all recipes, such as preheating the oven, mixing ingredients, and baking. However, the specific ingredients and some steps vary depending on the cake. Instead of writing a complete recipe for each cake from scratch, you write a general recipe (template) with placeholders for the varying ingredients and steps. This way, the general process remains the same, but you can easily create new recipes by filling in the blanks.

The Template Design Pattern works similarly. It defines the step-by-step structure of an algorithm, allowing the specific steps to be altered or extended by subclasses, without modifying the overall structure. Template Design Pattern is about avoiding boilerplate code, among other benefits. Boilerplate code refers to sections of code that are repeated in multiple places with little to no variation.

The Template Design Pattern is a fundamental concept in software engineering, particularly in object-oriented programming. It is a part of the behavioral design patterns family, which is concerned with the assignment of responsibilities between objects. The Template Pattern provides a blueprint for defining the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing the algorithm's structure.

### Code Example

Let's consider a simple example in Java that demonstrates the Template Design Pattern. We'll create a template for a cooking process, with two subclasses for making specific dishes.

public abstract class CookingTemplate {

    // Template method

    public final void cookDish() {

        prepareIngredients();

        cook();

        serve();

    }

    // Steps defined as abstract methods to be implemented by subclasses

    protected abstract void prepareIngredients();

    protected abstract void cook();

    // Common step implemented in the template

    protected void serve() {

        System.out.println("Dish is ready to be served!");

    }

}

Concrete Class 1:

public class PastaDish extends CookingTemplate {

    @Override

    protected void prepareIngredients() {

        System.out.println("Preparing pasta, sauce, and cheese.");

    }

    @Override

    protected void cook() {

        System.out.println("Cooking pasta and mixing it with sauce.");

    }

}

Concrete Class 2:

public class SaladDish extends CookingTemplate {

    @Override

    protected void prepareIngredients() {

        System.out.println("Chopping vegetables and preparing dressing.");

    }

    @Override

    protected void cook() {

        System.out.println("Mixing vegetables with dressing.");

    }

}

Client Code:

public class Main {

    public static void main(String[] args) {

        CookingTemplate pasta = new PastaDish();

        pasta.cookDish();

        System.out.println();

        CookingTemplate salad = new SaladDish();

        salad.cookDish();

    }

}

### When to Use

* When you have a series of steps forming an algorithm, and you expect that these steps could be changed or extended by subclasses.
* When you want to avoid code duplication, by pulling common behavior into a single superclass.
* To emphasize the common algorithm structure and hide certain steps from the client code.

### When Not to Use

* When changes to the algorithm do not vary between different implementations. In such cases, Strategy or State patterns might be more appropriate.
* When subclasses would only be implementing the same behavior, leading to unnecessary complexity.

**Pitfalls**

* **Limited Flexibility:** While the template pattern is great for algorithms with a fixed set of steps, it might be too restrictive if the steps in the algorithm need to change dynamically or if the algorithm's structure needs to be modified.
* **Overhead:** For simple algorithms, using the template pattern might introduce unnecessary complexity and overhead.
* **Inheritance Overuse:** The template pattern relies on inheritance, which can lead to a tightly coupled hierarchy. This might make the system harder to understand and maintain.