TutorialsPoint

In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behavior pattern category.

Implementation

We are going to create a *Game* abstract class defining operations with a template method set to be final so that it cannot be overridden. *Cricket* and *Football* are concrete classes that extend *Game* and override its methods.

*TemplatePatternDemo*, our demo class, will use *Game* to demonstrate use of template pattern.



Step 1

Create an abstract class with a template method being final.

*Game.java*

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

//template method

public final void play(){

//initialize the game

initialize();

//start game

startPlay();

//end game

endPlay();

}

}

Step 2

Create concrete classes extending the above class.

*Cricket.java*

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

*Football.java*

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

Step 3

Use the *Game*'s template method play() to demonstrate a defined way of playing game.

*TemplatePatternDemo.java*

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

System.out.println();

game = new Football();

game.play();

}

}

Step 4

Verify the output.

Cricket Game Initialized! Start playing.

Cricket Game Started. Enjoy the game!

Cricket Game Finished!

Football Game Initialized! Start playing.

Football Game Started. Enjoy the game!

Football Game Finished!

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////DIFFERENT SOURCE///////////////////////////////////////////////////

### When Would I Use This Pattern?

The Template Method pattern is used when

* When behaviour of an algorithm can vary, you let subclasses implement the behaviour through overriding
* You want to avoid code duplication, implementing variations of the algorithm in subclasses
* You want to control the point that subclassing is allowed.

Template Method may not be an obvious choice in the beginning, but the usual sign that you should use the pattern is when you find that you have two almost identical classes working on some logic. At that stage, you should consider the power of the template method pattern to clean up your code.

As you can imagine, use of the Template Method is fairly common. You'll find it used in the Arrays class uses it for sorting. JFrame uses update() as a template method, subclasses of the JFrame use paint(Graphics g) as their hook method.

### So How Does It Work In Java?

For our Java example, we'll use a cross compiler as an example. First, we'll create a generic cross compiler base class, with it's **crossCompile()**method being the glue for the whole algorithm to run.

public abstract class CrossCompiler {

public final void crossCompile() {

collectSource();

compileToTarget();

}

//Template methods

protected abstract void collectSource();

protected abstract void compileToTarget();

}

Next we'll create two specific implementations of our cross compiler, for iPhone and for Android:

public class IPhoneCompiler extends CrossCompiler {

protected void collectSource() {

//anything specific to this class

}

protected void compileToTarget() {

//iphone specific compilation

}

}

public class AndroidCompiler extends CrossCompiler {

protected void collectSource() {

//anything specific to this class

}

protected void compileToTarget() {

//android specific compilation

}

}

To complete this example, here is how you would use your cross compilers

public class Client {

public static void main(String[] args) {

CrossCompiler iphone = new IPhoneCompiler();

iphone.crossCompile();

CrossCompiler android = new AndroidCompiler();

android.crossCompile();

}

}

### Watch Out for the Downsides

There are some downsides to the template method pattern. Firstly, your base classes tend to get cluttered up with a lot of seemingly unrelated code. Program flow is a little more difficult to follow - without the help of stepping through the code with a debugger. Alex Miller provides a [detailed rundown](http://tech.puredanger.com/2007/07/03/pattern-hate-template/) of the reasons he hates the template method pattern in his blog.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////DIFFERENT SOURCE///////////////////////////////////////////////////

WIKIPEDIA

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the **template method pattern** is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_pattern" \o "Behavioral pattern) [design pattern](https://en.wikipedia.org/wiki/Software_design_pattern" \o "Software design pattern) that defines the [program skeleton](https://en.wikipedia.org/wiki/Program_skeleton" \o "Program skeleton) of an [algorithm](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm) in an operation, deferring some steps to [subclasses](https://en.wikipedia.org/wiki/Subclass_(computer_science)" \o "Subclass (computer science)).[[1]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:0-1) It lets one redefine certain steps of an algorithm without changing the algorithm's structure.

## Introduction

In the template method of this design pattern, one or more algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed.

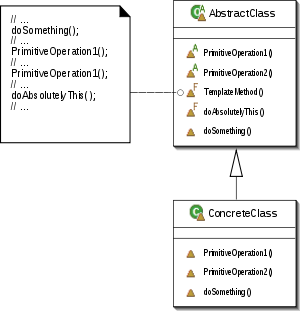
In object-oriented programming, a concrete class is created that provides the steps of an [algorithm design](https://en.wikipedia.org/wiki/Algorithm_design" \o "Algorithm design). Steps that are considered invariant are implemented inside the base class. The steps that are considered to be variant, are given a default implementation or none at all. These variant steps must be supplied by concrete derived subclasses.[[3]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:1-3) Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.

The template method pattern thus manages the larger picture of task [semantics](https://en.wikipedia.org/wiki/Semantics" \o "Semantics), and more refined implementation details of selection and sequence of methods. This larger picture calls abstract and non-abstract methods for the task at hand. The non-abstract methods are completely controlled by the template method, but the abstract methods, implemented in subclasses, provide the pattern's expressive power and degree of freedom. Template method's abstract class may also define hook methods that may be overridden by subclasses.[[2]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:2-2)

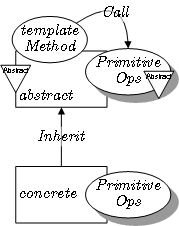
Some or all of the abstract methods can be specialized in a subclass, allowing the writer of the subclass to provide particular behavior with minimal modifications to the larger semantics. The template method (that is non-abstract) remains unchanged in this pattern, ensuring that the subordinate non-abstract methods and abstract methods are called in the originally intended sequence.

The template method pattern occurs frequently, at least in its simplest case, where a method calls only one abstract method when using object oriented languages. If a software writer uses a [polymorphic](https://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming" \o "Polymorphism in object-oriented programming) method at all, this design pattern may be a rather natural consequence. This is because a method calling an abstract or polymorphic function is simply the reason for being of the abstract or polymorphic method. The template method pattern may be used to add immediate present value to the software or with a vision to enhancements in the future.

## Structure[[edit](https://en.wikipedia.org/w/index.php?title=Template_method_pattern&action=edit&section=2" \o "Edit section: Structure)]

[](https://en.wikipedia.org/wiki/File:Template_Method_UML.svg)

Template method: [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) [class diagram](https://en.wikipedia.org/wiki/Class_diagram" \o "Class diagram).

[](https://en.wikipedia.org/wiki/File:Template_Method_pattern_in_LePUS3.gif)

Template Method in [LePUS3](https://en.wikipedia.org/wiki/Lepus3).[[4]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-4)

## Usage[[edit](https://en.wikipedia.org/w/index.php?title=Template_method_pattern&action=edit&section=3" \o "Edit section: Usage)]

The template method is used in frameworks, where each implements the invariant parts of a domain's architecture, leaving "placeholders" for customization options. This is an example of [inversion of control](https://en.wikipedia.org/wiki/Inversion_of_control" \o "Inversion of control). The template method is used for the following reasons:[[3]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:1-3)

* Let subclasses implement varying behavior (through [method overriding](https://en.wikipedia.org/wiki/Method_overriding_(programming)" \o "Method overriding (programming))).[[5]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:3-5)
* Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's [algorithm](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm), and necessary variations are implemented in the subclasses.[[5]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:3-5)
* Control at what point(s) [subclassing](https://en.wikipedia.org/wiki/Inheritance_(computer_science)" \o "Inheritance (computer science)) is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.[[5]](https://en.wikipedia.org/wiki/Template_method_pattern#cite_note-:3-5)

### Usage with code generators**[[edit](https://en.wikipedia.org/w/index.php?title=Template_method_pattern&action=edit&section=4" \o "Edit section: Usage with code generators)]**

The template pattern is useful working with auto-generated code. The challenge of working with generated code is that any refinement of the source material will lead to changes in the generated code, which could overwrite hand-written modifications. This may be solved using the Template pattern, by generating abstract code, and making hand-written modifications to a concrete subclass or implementation class. The abstract code may be in the form of an abstract class in C++, or an interface in Java or C#. The hand-written code would go into a subclass in C++, and an implementing class in Java or C#. When used with code generation, this pattern is sometimes referred to as the [Generation Gap](https://en.wikipedia.org/wiki/Generation_gap_(pattern)" \o "Generation gap (pattern)) pattern.