Introduction to Design Patterns

Each **design pattern** describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem in a way such that the solution can be reused repeatedly without ever doing it the same way twice.

There are mainly three categories of design patterns:

* **Creational Patterns** – These deal with the process of object creation and class instantiation.
* **Structural Patterns** – These deal with the structure of classes and objects to form larger structures and provide new functionality.
* **Behavioural Patterns** – These deal with the dynamic interaction between classes and objects.

The complete list of design patterns is provided below:

* Creational Patterns
  + Abstract Factory
  + Builder
  + Factory Method
  + Prototype
  + Singleton
* Structural Patterns
  + Adapter
  + Bridge
  + Composite
  + Decorator
  + Façade
  + Flyweight
  + Proxy
* Behavioural Patterns
  + Chain of Responsibility
  + Command
  + Interpreter
  + Iterator
  + Mediator
  + Memento
  + Observer
  + State
  + Strategy
  + Template Method
  + Visitor

## Template Pattern

Suppose we have a scenario where multiple algorithms can be divided into several similar steps. For example, consider that we build simulations for two games, each of which have an initialize method, a startPlay method and an endPlay method.

public class Cricket {  
 public void initialize() {};  
 public void startPlay() {};  
 public void endPlay() {};  
}

JAVA

public class Football {  
 public void initialize() {};  
 public void startPlay() {};  
 public void endPlay() {};  
}

JAVA

Instead of having separate classes, we can have a single parent class that contains the implementations for all of these methods. Then we can use a single method call that will in turn call each of these methods one by one.

public abstract class Game {  
 public final void play() {  
 initialize();  
 startPlay();  
 endPlay();  
 }  
 public void initialize() {};  
 public void startPlay() {};  
 public void endPlay() {};  
}

JAVA

If the implementations of any of the methods being called in the play method changes based on which class is inheriting from the parent, it should be made abstract. Otherwise, the method should be made final. This is because we know for a fact that the implementation of the final methods are not supposed to change. Declaring them final will help preserve this logic in all future classes that inherit from this one.

The purpose of the template pattern is to define the **skeleton of an algorithm**, perhaps deferring some of the steps to the client subclasses. Thus, the template pattern allows the client to change certain steps of an algorithm without changing the algorithm structure. For example, a client subclass can override the startPlay method and define what exactly happens inside the method, but they cannot change the fact that endPlay will follow startPlay.

The template pattern solves the problem created when two components have significant similarities but demonstrate no reuse of common interfaces or implementations. It suggests that the components be broken down into a series of steps and each step executed in turn using a **template method** (the play method in the example above).

### Hooks

An interesting caveat of the template method is the use of conditional steps. What if we need to execute a certain step conditionally in the algorithm? One possibility is to simply include an if statement.

public void play() {  
 initialize();  
 startPlay();  
 if (gameHasBreak) {  
 takeBreak();  
 }  
 endPlay();  
}

JAVA

A more common implementation of this is to use a **hook**, a method which is empty and can be used by the client to add whatever code they wish in between two steps.

public void play() {  
 initialize();  
 startPlay();  
 takeBreak();  
 endPlay();  
}

JAVA

A prominent use case of hooks is in the React framework.

### The Hollywood Principle

The use of the Template Pattern brings us to another design principle.

💡**The Hollywood Principle**: High-level components should call low-level ones, not the other way around.

In the case of this pattern, the template method is the high-level component which is calling each of the low-level components in turn. We are not using the low-level components to modify or execute steps in the high-level component. By not doing this, we avoid **dependency rot**. If high level components were dependent on low level ones for their proper execution, over time, changes to the low-level components would cause the high-level ones to stop working and it would be difficult to understand why.