Creational Design Patterns

Contents

Overview

Creational design patterns are designed to make object-creation easier, while hiding object-creation logic from view.

The creational patterns are:

- Factory
- Builder
- Singleton
- Prototype
- Abstract Factory

Tutorial

Builder Pattern

The **Builder** pattern allows for us to build more complex objects through subsets of simpler objects.

This is a creational pattern as it is concerned with the creation of objects. The Builder is independent of other objects and will receive each parameter step-by-step before returning the resulting constructed object.

Most commonly, it is used to prevent the issue of creating multiple objects from a class which contains a large number of parameters.

Without using the pattern, a class might contain several constructors, all using different permutations of parameters for every use case. This results in difficult-to-maintain code, as the class would become inundated with constructors.

In the example below, we have a Trainee `class` with an undefined large number of parameters:

▶ Trainee

Let's try using a `TraineeBuilder` class to alleviate this issue:

▶ Builder Pattern

In **method 1**, `TraineeBuilder()` will be used to create `Trainee` objects.

In **method 2**, we then create the actual `Trainee` object, which is fully instantiated when returned.

Finally, in **methods 3, 4, and 5**, we configure whichever parameters from the `Trainee` class we want, which will then be used to create `Trainee` objects.

Now, when we're creating `Trainee` objects, we can use the `TraineeBuilder()` to do this with as many of the attributes as we wish, chaining each method for each parameter as we go along:

▶ Building Trainee objects
The default fields will auto-populate with whichever values we set in the `TraineeBuilder` class when we haven't passed anything to the parameters.

This allows for a more verbose and clear method of creating objects, whilst keeping it concise to write!

Builder Pattern Advantages

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

Builder Pattern Disadvantages

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Singleton Pattern

The **Singleton** pattern is another creational pattern which lets us ensure that a class has exactly *one* instance in the JVM, while also providing a global access point to that instance.

Generally, it is used for single-point-of-operation modules, such as logging, driver objects, caching, and thread pools, though it is occasionally used within the implementations of other design patterns (e.g. Abstract Factory, Builder, Prototype, Façade).

All implementations of the Singleton pattern will have these two behaviours in common:

1. A `private` constructor, to prevent other objects from using the new operator with the Singleton class.
2. A `static` creation method that acts as a constructor. Under the hood, this method calls the `private` constructor to create an object and saves it in a `static` field. All following calls to this method return the cached object.

While the Singleton pattern solves some issues, it violates the Single Responsibility principle. As a result, it is occasionally called an 'anti-pattern' because it introduces global state to an application. It is also frequently misused, so use it with caution!

The below `ClassSingleton` shows it in action:

▶ Singleton Pattern
To use our `ClassSingleton` as a Singleton, we must make a `static` instance of it:

▶ Using Singleton
- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Use the Singleton pattern if you need to have stricter control over global variables.

Singleton Pattern Advantages

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it's requested for the first time.

Singleton Pattern Disadvantages

- Violates the Single Responsibility Principle.
- It can mask bad design, for instance, when the components of the program know too much about each other.
- It requires special treatment in a multi-threaded environment so that multiple threads won’t create a Singleton object several times.
- It can be difficult to unit test the client code of the Singleton, as many test frameworks rely on inheritance when producing mock objects. Since the constructor of the Singleton class is `private`, and overriding `static` methods is impossible in Java (and most other languages), you would need to think of a creative way to mock the Singleton. (Or just don’t write the tests. Or don’t use the Singleton pattern.)

Exercises

Research and implement a Builder pattern for the following class:

```java
public class BankAccount {
    private long accountNumber;
    private String owner;
    private String branch;
    private double balance;
    private double interestRate;

    public BankAccount(long accountNumber, String owner, String branch, double
balance, double interestRate) {
        this.accountNumber = accountNumber;
        this.owner = owner;
        this.branch = branch;
        this.balance = balance;
        this.interestRate = interestRate;
    }

    // getters and setters

}
```

▶ Solution