Design Patterns

Mradul Singhal

April 11, 2025

Contents

1 Introduction **3**
1.1 Types of Design Patterns 3
1.1.1 Creational Design Patterns 3
1.1.2 Structural Design Patterns 3
1.1.3 Behavioral Design Patters 3

2 Creational Design Patterns **4**
2.1 Singleton Pattern 4
2.2 Factory Pattern 6

Introduction

Design patterns are **well-established resolutions for repetitive challenges in software design**. They are programming language-independent strategies for solving a common problem. Design patterns are templates for solving a common problem in programming. They are general solutions to problems faced during software development. By using design patterns, we can make our code more flexible, reusable, and maintainable.

Types of Design Patterns

Creational Design Patterns

Creational design patterns provide solutions to instantiate an Object in the best possible way for specific situations.

Structural Design Patterns

Structural design patterns provide different ways to create a Class structure (for example, using inheritance and composition to create a large Object from small Objects).

Behavioral Design Patters

Behavioral patterns provide a solution for better interaction between objects and how to provide loose-coupling and flexibility to extend easily.

Creational Design Patterns

Singleton Pattern

- The **Singleton design pattern** is a **creational design pattern** that ensures a class has only one instance and provides a global access point to that instance.
- This pattern is particularly useful in scenarios where a single object is needed to coordinate actions across the system, such as managing database connections or configuration settings.
- It restricts the instantiation of a class to a single instance. By doing so, it helps maintain a controlled access point to the instance throughout the application's lifecycle.
- **Example:**

```
class Singleton {

  private static Singleton instance;

  private Singleton() {}

  public static Singleton getInstance() {
      if (instance == null) {
          instance = new Singleton();
      }
      return instance;
  }
}
```

- The lifetime of a singleton object is equal to the runtime of the application. The object may be configured to be lazily loaded.
- **Uses:**
  - If we need only one instance of a class throughout the application.
  - If we need to use to inject a single instance of class *a lot of* times as a dependency.

- **Pros:**
  - Provides a single point of access to a shared resource.
  - Reduces memory footprint when the object is reused across multiple components.
- **Cons:**
  - May cause global state issues, which contribute to hiding actual dependencies.
  - May result in non-deterministic behavior if the attributes of the singleton are mutable.
    Example:

```
class Singleton {

  int x;

  ...
}
```

```
class A {

  public static void main(String[] args) {

      var singleton = Singleton.getInstance();
      System.out.println(singleton.x++);
  }
}
```

```
class B {

  public static void main(String[] args) {

      var singleton = Singleton.getInstance();
      System.out.println(singleton.x--);
  }
}
```

- It persists throughout the lifecycle of the application, even when it's not being used.
- It is difficult to unit test and extend since there is no public constructor, and the only way to instantiate it is through a static method.

- **Tips:**
  - Avoid using mutable state inside a singleton unless absolutely necessary.
  - If the singleton is resource-heavy and not always needed, implement lazy loading to delay instantiation.
  - Ensure thread safety in multi-threaded environments by using synchronization or other concurrency-safe mechanisms.
  - Use dependency injection frameworks (like Spring) to manage singleton scope more cleanly and transparently.
  - If we do use Singletons, try to use dependency injection instead of calling `getInstance()` from the constructor.

```
public MyConstructor(Singleton singleton) {
    this.singleton = singleton;
}
```

instead of

```
public MyConstructor() {
    this.singleton = Singleton.getInstance();
}
```

At the very least, using dependency injection allows us to do some unit testing of the class by adhering to good encapsulation principles.

Factory Pattern

- The **Factory design pattern** is a **creational design pattern** that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- It helps in promoting loose coupling by eliminating the need to bind application-specific classes into the code.

- This pattern delegates the responsibility of instantiating a class to its subclasses or factory methods.
- It is particularly useful when the exact type of the object is not known until runtime or when the object creation involves complex logic.
- **Example:**

```
// Product interface
interface Shape {
  void draw();
}

// Concrete Product 1
class Circle implements Shape {
  public void draw() {
      System.out.println("Drawing a Circle");
  }
}

// Concrete Product 2
class Rectangle implements Shape {
  public void draw() {
      System.out.println("Drawing a Rectangle");
  }
}

// Factory class
class ShapeFactory {
  public Shape getShape(String shapeType) {
      if (shapeType == null) return null;
      if (shapeType.equalsIgnoreCase("CIRCLE")) {
          return new Circle();
      } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
          return new Rectangle();
      }
      return null;
  }
}

// Usage
```

```
class Main {
  public static void main(String[] args) {
      ShapeFactory factory = new ShapeFactory();

      Shape shape1 = factory.getShape("CIRCLE");
      shape1.draw();

      Shape shape2 = factory.getShape("RECTANGLE");
      shape2.draw();
  }
}
```

- The Factory pattern promotes the use of interfaces or abstract classes and defers the instantiation to child classes or factory methods.
- **Uses:**
  - If we have to know more than the product to construct A, B or C, and we can't have direct access to that knowledge, then it is useful. Then the factory can act as a knowledge center for producing needed objects.
  - If objects that needs to be instantiated need a reference to some object X, which the factory can provide, but our code in the place where we want to construct A, B or C can't or shouldn't have access to X.
  - If the case is, when we have X we create A and B but if we have Y type then we create C.
  - Also useful when some objects need a lot of dependencies to create. Hunting for those dependencies in a place where they should not be accessible might be problematic.
- **Pros:**
  - Promotes loose coupling and enhances code maintainability.
  - Makes the code more scalable as new product types can be added with minimal changes.
  - Encapsulates object creation logic in one place.
- **Cons:**

  - May introduce unnecessary complexity if the number of products is small or object creation is simple.

- **Tips:**
  - Keep the factory class focused on object creation only — don't mix with unrelated logic.
  - Avoid using factory pattern when object creation is not complicated enough, or not many variants of object.