Design Patterns in Java

Marcin Kurzyna

January 21, 2025

Contents

1 Introduction

Design patterns are proven solutions to common software design problems. They provide templates or guidelines for writing code that is modular, reusable, and maintainable. Originating from the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides, these patterns have become fundamental to modern software engineering practices.

2 Why Use Design Patterns?

Design patterns are useful because they:

- Provide standardized and proven solutions to recurring problems.
- Enhance code readability and maintainability.
- Promote best practices in object-oriented design.
- Facilitate communication among developers by providing a shared vocabulary.

3 Types of Design Patterns

Design patterns are broadly classified into three categories:

1. **Creational Patterns**: Deal with object creation mechanisms. Examples: Singleton, Factory Method, Builder.
2. **Structural Patterns**: Concerned with object composition and relationships. Examples: Adapter, Composite, Proxy.
3. **Behavioral Patterns**: Focus on communication between objects. Examples: Observer, Strategy, Command.

4 Example 1: Singleton Pattern

4.1 Definition

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.

4.2 Implementation in Java

```
class Singleton {
    // Private static variable of the same class that is the only
        instance
    private static Singleton instance;

    // Private constructor to restrict instantiation of the class
    private Singleton() {}
```

```
7
8       // Public static method that returns the instance of the
            class
9       public static Singleton getInstance() {
10          if (instance == null) {
11              instance = new Singleton();
12          }
13          return instance;
14      }
15
16      public void showMessage() {
17          System.out.println("Hello, Singleton Pattern!");
18      }
19  }
20
21  public class Main {
22      public static void main(String[] args) {
23          Singleton singleton = Singleton.getInstance();
24          singleton.showMessage();
25      }
26  }
```

Listing 1: Singleton Pattern in Java

5 Example 2: Factory Method Pattern

5.1 Definition

The Factory Method Pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.

5.2 Implementation in Java

```
1   abstract class Shape {
2       abstract void draw();
3   }
4
5   class Circle extends Shape {
6       @Override
7       void draw() {
8           System.out.println("Drawing a Circle");
9       }
10  }
11
12  class Rectangle extends Shape {
13      @Override
14      void draw() {
15          System.out.println("Drawing a Rectangle");
16      }
17  }
```

```
18
19 class ShapeFactory {
20     public static Shape getShape(String shapeType) {
21         if (shapeType.equalsIgnoreCase("CIRCLE")) {
22             return new Circle();
23         } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
24             return new Rectangle();
25         }
26         return null;
27     }
28 }
29
30 public class Main {
31     public static void main(String[] args) {
32         Shape shape1 = ShapeFactory.getShape("CIRCLE");
33         shape1.draw();
34
35         Shape shape2 = ShapeFactory.getShape("RECTANGLE");
36         shape2.draw();
37     }
38 }
```

Listing 2: Factory Method Pattern in Java

6 Example 3: Decorator Pattern

6.1 Definition

The Decorator Pattern allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. It is often used for adding responsibilities to objects in a flexible and reusable way.

6.2 Implementation in Java

```
1  // Base interface
2  interface Pizza {
3      String getDescription();
4      double getCost();
5  }
6
7  // Concrete implementation of a plain pizza
8  class PlainPizza implements Pizza {
9      @Override
10     public String getDescription() {
11         return "Plain Pizza";
12     }
13
14     @Override
15     public double getCost() {
16         return 5.00;
```

```
    }
}

// Abstract decorator class
abstract class PizzaDecorator implements Pizza {
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDescription() {
        return pizza.getDescription();
    }

    @Override
    public double getCost() {
        return pizza.getCost();
    }
}

// Concrete decorators
class Ham extends PizzaDecorator {
    public Ham(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", Ham";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 2.00;
    }
}

class Pineapple extends PizzaDecorator {
    public Pineapple(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", Pineapple";
    }

    @Override
    public double getCost() {
```

```
68          return pizza.getCost() + 1.50;
69      }
70  }
71
72  class Sauce extends PizzaDecorator {
73      public Sauce(Pizza pizza) {
74          super(pizza);
75      }
76
77      @Override
78      public String getDescription() {
79          return pizza.getDescription() + ", Sauce";
80      }
81
82      @Override
83      public double getCost() {
84          return pizza.getCost() + 0.75;
85      }
86  }
87
88  // Test the decorator pattern
89  public class Main {
90      public static void main(String[] args) {
91          Pizza pizza = new PlainPizza();
92          System.out.println(pizza.getDescription() + " - $" +
                pizza.getCost());
93
94          pizza = new Ham(pizza);
95          System.out.println(pizza.getDescription() + " - $" +
                pizza.getCost());
96
97          pizza = new Pineapple(pizza);
98          System.out.println(pizza.getDescription() + " - $" +
                pizza.getCost());
99
100         pizza = new Sauce(pizza);
101         System.out.println(pizza.getDescription() + " - $" +
                pizza.getCost());
102     }
103 }
```

Listing 3: Decorator Pattern with Pizza in Java

7 Conclusion

Design patterns are an essential part of software development. They help developers solve complex design problems efficiently by providing reusable and time-tested solutions. By understanding and applying design patterns, developers can create robust and maintainable applications in Java.