The 10 most popular types of design patterns in Java

In software development, it is essential to provide flexibility in the design in such a way that it is easy to maintain, efficient, and can work with changing technological or business conditions. Therefore, for our application code to meet such conditions, we must use proven practices, which include design patterns.

Java, being one of the most popular programming languages, has its own set of software design patterns that every Java programmer should know. In this article, we will learn about the most popular ones and analyze their use.

Table of Contents

What are software design patterns?

A design pattern is a reusable solution for a typical problem encountered in software design that can be applied to various situations. These are not finished designs or code, but proven practices of many software developers with ready-to-use reusable solutions for selected problems encountered in the design of object-oriented solutions.

A key role in the process of popularizing the concept of design patterns was played by the authors of the bestseller "Design Patterns: Elements of Reusable Object-Oriented Software": Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, collectively known as the "Gang of Four" (GoF).

Since 1994, dozens of other patterns have been discovered for object-oriented programming. The “pattern approach” has become very popular in many frameworks. As a result, there are now many more patterns outside object-oriented design. Now software developers use these patterns, sometimes even unconsciously. On the other hand, it makes sense to use them intentionally to talk to programmers using the same language elements and sharing the same source code.

Why are Java design patterns important?

Java design patterns gained popularity for several reasons. First, **they promote reusable object-oriented software,** allowing developers to write modular and extensible code. This makes it easier to maintain and update software systems long-term.

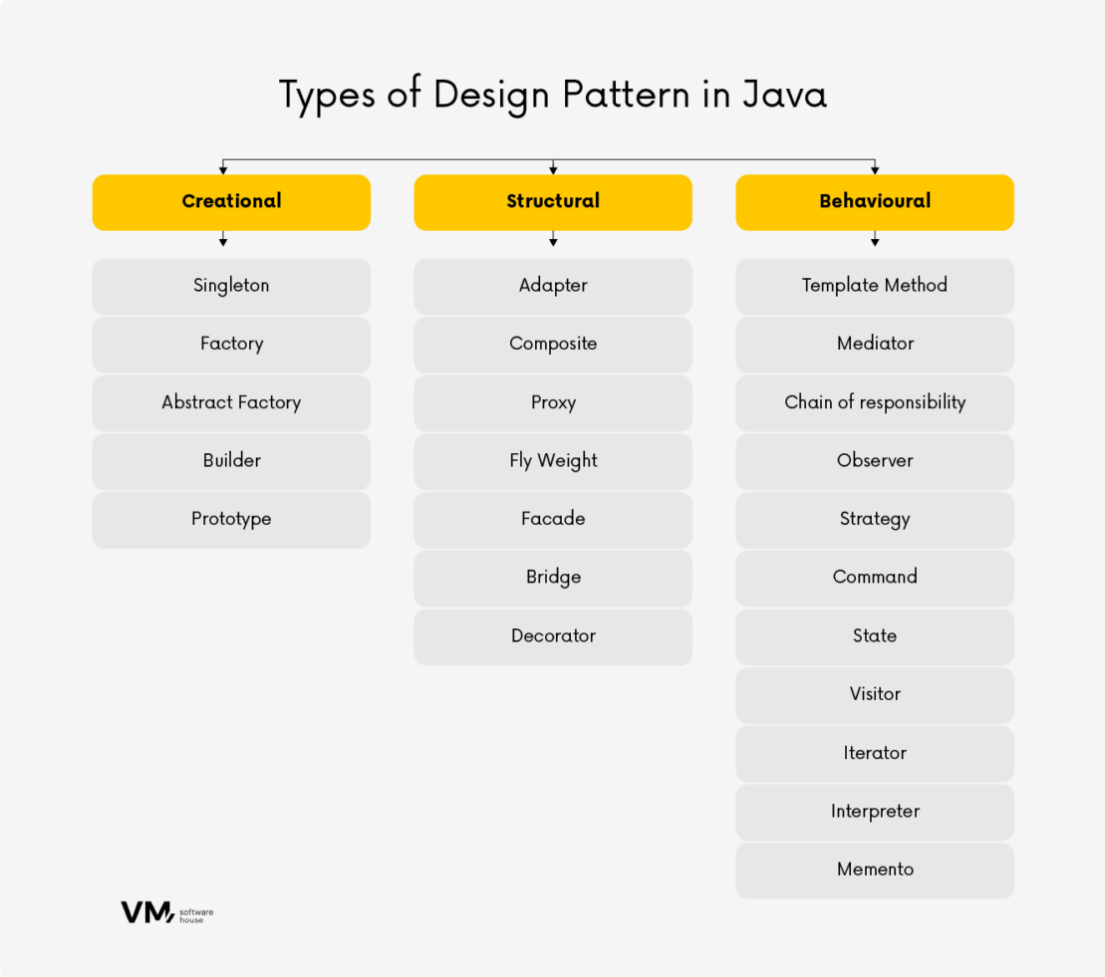
The Factory pattern, for example, is a software design pattern that provides an interface for creating objects but allows subclasses to decide which class to create an instance of. This promotes code reuse by allowing developers to create objects without specifying the exact class of object that will be created.

Second, design patterns **improve the code readability and maintainability**. By following a standard structure, patterns make existing code easier to understand, which is especially helpful when working in teams or when returning to the code base after a long time.

Three commonly used categories of design patterns

The most universal and high-level patterns that we will address today are architectural patterns. Developers can apply them to virtually any programming language, and they use them to design the architecture of an entire application.

According to the previously mentioned manual, 23 design patterns can be divided into three categories depending on the responsibility being carried out: **creative, structural, and behavioral** patterns. Let’s now take a closer look at some of the important design patterns used in Java.



Creational design patterns

These design patterns address how to create classes, methods, and data types, focusing on the process of creating objects, and trying to compose objects in a way that is appropriate to the situation. Creational patterns aim to make object creation more flexible and separate from client code, promoting code reusability and maintainability.

- **Factory**

The Factory pattern **provides an interface for creating objects in a superclass but allows subclasses to change the types of objects that will be created.** It is used when a system needs to be independent of object creation, composition, and representation. This allows a class to delegate the responsibility for creating instances to its subclasses.

This creational design pattern is commonly used in scenarios where there are multiple classes implementing a common interface or extending a common base class. There are different variations of the class creation patterns, such as Simple Factory, Factory Method and Abstract Factory. Each variation has its advantages and use cases, allowing you to choose the most suitable one for your specific requirements.

- **Builder**

The Builder pattern is a creative design pattern that **separates the construction of a complex object from its representation, allowing the same construction process to create different representations**. It includes a director class, which coordinates the construction process, and a builder interface with specific classes implementing the construction steps.

The Builder pattern is often used by developers using the Lombok library, which generates Java code for us and, among other things, also provides the creation of methods that are used in the Builder. That is, by creating classes and creating new objects, we can use them effortlessly without writing a lot of code. It is used when an object needs to be constructed with many possible configurations or when the construction process involves many steps.

- **Singleton**

The Singleton design pattern ensures that a **class has only one instance, which is also the global access point to that instance**. It is extremely useful when dealing with service implementations that we use only once in the entire program, regardless of where they are created. The singleton pattern is a solution especially used for connections to the database or to devices that use serial port communication. By having only one instance of the class, you can ensure efficient use of resources and avoid unnecessary load.

- **Prototype**

The Prototype pattern is usually used when w**e have an instance of a class (prototype) and want to create new objects by simply copying the prototype**. This pattern is particularly useful when the cost of creating a new object is more expensive or complicated than copying an existing one. It is particularly useful in scenarios where object creation must be dynamic and flexible.

Structural patterns

The structural design patterns deal with the r**elationships of related objects to form larger structures**. These patterns help define relationships between entities and simplify the design of complex systems.

- **Decorator**

The Decorator pattern allows you to **add behavior to a single object, either statically or dynamically, without affecting the behavior of other objects in the same class**. It is a structural pattern involving a set of decorator classes that are used to wrap specific components. It is most often used when we want to extend the behavior of individual objects without modifying their code.

- **Facade**

The Facade pattern provides a **simplified (but limited) interface to a complex system of classes, a library, or a framework, reduces the overall complexity of the application, and helps move unwanted dependencies to a single place**. The main goal of the Facade pattern is to simplify and standardize the set of interfaces, making the subsystem more accessible and easier to use for customers. Facade patterns are most often used in applications written in Java when working with complex libraries and APIs.

- **Proxy**

The proxy design pattern is a representative or proxy of another object to gain supervised access to the object it represents. Using this pattern, we can minimize the load and increase performance. If we need to use some library that we know will cause a heavy load on the CPU server, we can, as it were, postpone the operation until the final moment when we need to use it. In the case of applications, we save memory and hardware resources for computationally heavy operations and load complex elements only when they are needed.

Behavioral patterns

They concern the behavior of cooperating related or dependent objects and how classes and objects interact, communicate, and cooperate. **Behavioral patterns are primarily concerned with the delegation of responsibility between existing objects and the patterns of communication between them**. They are very commonly used in enterprise projects.

This is because by using third-party library providers, we may have to deal with memory leaks, which can cause us problems that we will only find out about in a few weeks or months. In such situations, it is useful to use different design patterns to monitor the performance of the application, for example.

- **Strategy**

The strategy design pattern defines a family of algorithms, encapsulates each of them, and allows them to change. The strategy pattern **allows an algorithm to change independently of the clients that use it.** It allows the client to choose from the family of algorithms at the time of execution. The key idea of the Strategy pattern is to separate the algorithms from the clients who use them, promoting flexibility and extensibility.

- **Visitor**

The visitor pattern is responsible for performing some specific operation on a complex data structure. **It allows you to define a new operation without changing the classes of elements on which it operates**. That is, for example, it can log for us the times of the operation or tell us who when, and how certain functions were run.

- **Observer**

Observer design pattern **defines a one-to-many relationship between objects** so that when one object changes state, all its dependent objects are automatically notified and updated. On the one hand, the observer pattern ensures that we have a module that publishes events to us, for example, a user has requested certain data, and then we create an event, which in turn triggers a reaction in the other part of the system, generating the corresponding data.

Three examples of using design patterns in software development

Here you'll find the code examples of using design patterns in the software development process.

Example #1

In the first case, we will use four types of software design patterns to solve the problem of creating different Vehicle objects that have specific behaviors.

- **The Builder design pattern** obtained with the @SuperBuilder annotation is used to create Builder methods during inheritance. Lombok's regular @Builder does not apply here.
- **Factory design pattern** used in the VehicleFactory class to produce object types depending on the name.
- **Factory Methods** (manufacturing method) defined with VehicleCreator classes contain details of manufacturing individual vehicles.
- **Strategy design pattern** used in the form of EconomicDriving and AggressiveDriving defines the driving styles of individual vehicles.

Source: Github: https://encr.pw/6jREj

```
package org.example;


import lombok.Getter;
import lombok.experimental.SuperBuilder;


// Builder pattern using Lombok in class hierarchy
@SuperBuilder
@Getter
abstract class Vehicle {
    // Common properties for all vehicles
    private int wheels;
    private int seats;
    private String engine;
    private VehicleBehavior behavior;
```

```
    // Method to drive the vehicle, delegates behavior execution
    public void drive() {
        behavior.perform();
    }
}

// Truck class extending Vehicle, utilizing Lombok's SuperBuilder
@SuperBuilder
class Truck extends Vehicle {
    @Override
    public void drive() {
        // Prints engine type followed by "truck" before executing common drive logic
        System.out.print(getEngine() + " truck ");
        super.drive();
    }
}

// Bus class extending Vehicle, utilizing Lombok's SuperBuilder
@SuperBuilder
class Bus extends Vehicle {
    @Override
    public void drive() {
        // Prints engine type followed by "bus" before executing common drive logic
        System.out.print(getEngine() + " bus ");
        super.drive();
    }
}
```

```
// Car class extending Vehicle, utilizing Lombok's SuperBuilder
@SuperBuilder
class Car extends Vehicle {
  @Override
  public void drive() {
    // Prints engine type followed by "car" before executing common drive logic
    System.out.print(getEngine() + " car ");
    super.drive();
  }
}

// Factory pattern implementation for creating Vehicle instances
class VehicleFactory {
  public static Vehicle getVehicle(String type) {
    switch (type) {
      case "truck":
        // Returns a Truck instance using TruckCreator
        return new TruckCreator().buildVehicle();
      case "bus":
        // Returns a Bus instance using BusCreator
        return new BusCreator().buildVehicle();
      case "car":
        // Returns a Car instance using CarCreator
        return new CarCreator().buildVehicle();
      default:
        // Throws an exception for an invalid vehicle type
        throw new IllegalArgumentException("Invalid vehicle type: " + type);
```

```
    }
  }
}
```

After starting the program, you will get on the screen:

- 24.0L V12 diesel truck economic driving
- 6.0L V6 diesel bus economic driving
- 2.0L R4 petrol car aggressive driving

Example #2

In the second example, we will use the **Memento pattern as a behavioral design pattern** (it lets you save and restore the previous state of an object without revealing the details of its implementation) in a program like Paint to get the possibility of a 10-level *'Undo'* or backward function.

We can draw shapes on the canvas (Canvas) and fill it with colors. Each time we press Save, Caretaker creates and saves the current state of the canvas as a CanvasMemento object. If we press Undo, we return to the previous state.

After running the program twice, save the changes, modify once more, and then use undo twice to restore the previous canvas values.

Source: Github https://l1nq.com/8PRm1

```
package org.example;


import java.util.LinkedList;
import java.util.List;


// Memento class to store the state of the Canvas
class CanvasMemento {
   private final String canvasState;
```

```
    // Constructor to initialize the state
    public CanvasMemento(String state) {
        this.canvasState = state;
    }

    // Getter method to retrieve the stored state
    public String getState() {
        return canvasState;
    }
}

// Originator class that represents the Canvas
class Canvas {
    private String content = "";

    // Method to draw a shape on the canvas
    public void draw(String shape) {
        content += " draw:" + shape;
    }

    // Method to fill the canvas with a color
    public void fill(String color) {
        content += " fill:" + color;
    }

    // Creates a new Memento with the current state
    public CanvasMemento save() {
        return new CanvasMemento(content);
```

```
    }

    // Restores the state from the provided Memento
    public void undoToLastSave(CanvasMemento memento) {
        content = memento.getState();
    }

    // Returns the current state of the canvas as a string
    @Override
    public String toString() {
        return content;
    }
}

// Caretaker class that manages the saved states
class Caretaker {
    private final List<CanvasMemento> saveStates = new LinkedList<>();
    private final Canvas canvas;

    // Constructor to initialize the caretaker with a canvas
    public Caretaker(Canvas canvas) {
        this.canvas = canvas;
    }

    // Saves the current state of the canvas
    public void save() {
        saveStates.add(canvas.save());
    }
```

```
    // Restores the last saved state
    public void undo() {
        if (!saveStates.isEmpty()) {
            canvas.undoToLastSave(saveStates.remove(saveStates.size() - 1));
        }
    }
}
```

On the screen, we get:

- Canvas after all operations: draw:Circle fill:Red draw:Triangle
- Canvas after undo draw triangle: draw:Circle fill:Red
- Canvas after undo fill red: draw:Circle

Example #3

The third example is the application of the **Visitor design pattern** to a building inspector, who is tasked with performing actions known only to him while visiting successive rooms of a house.

Once we accept the inspector and let it into the house *(house.accept(inspector))*, BuildingInspector starts inspecting the rooms defined at the house construction stage. It performs actions specific to each room and finally checks the structure of the entire house when we call '*visitor.visit(this)'* in the accept method of the House class

Visitor pattern separates algorithms from the objects on which they operate-better object composition

```
package org.example;

// Visitor Design Pattern Implementation for Building Inspection
public class BuildingInspection {
```

```
// BuildingPart interface representing different parts of a building
interface BuildingPart {
    void accept(BuildingPartVisitor visitor);
}

// Concrete class representing a Kitchen
static class Kitchen implements BuildingPart {
    public void accept(BuildingPartVisitor visitor) {
        visitor.visit(this);
    }
}

// Concrete class representing a Living Room
static class LivingRoom implements BuildingPart {
    public void accept(BuildingPartVisitor visitor) {
        visitor.visit(this);
    }
}

// Concrete class representing a Bathroom
static class Bathroom implements BuildingPart {
    public void accept(BuildingPartVisitor visitor) {
        visitor.visit(this);
    }
}

// Concrete class representing a House containing multiple BuildingParts
```

```
static class House implements BuildingPart {
    BuildingPart[] parts;

    public House() {
        parts = new BuildingPart[]{new Kitchen(), new LivingRoom(), new Bathroom()};
    }

    public void accept(BuildingPartVisitor visitor) {
        // Let each part accept the visitor
        for (BuildingPart part : parts) {
            part.accept(visitor);
        }
        // Visit the house itself
        visitor.visit(this);
    }
}

// Visitor interface defining methods for visiting different building parts
interface BuildingPartVisitor {
    void visit(Kitchen kitchen);
    void visit(LivingRoom livingRoom);
    void visit(Bathroom bathroom);
    void visit(House house);
}

// Concrete Visitor that performs inspection on each building part
static class BuildingInspector implements BuildingPartVisitor {
    public void visit(Kitchen kitchen) {
```

```
        System.out.println("Inspecting the kitchen: Checking appliances and
ventilation.");
    }

    public void visit(LivingRoom livingRoom) {
        System.out.println("Inspecting the living room: Checking furniture and wiring.");
    }

    public void visit(Bathroom bathroom) {
        System.out.println("Inspecting the bathroom: Checking plumbing and
sanitation.");
    }

    public void visit(House house) {
        System.out.println("Finalizing house inspection: Ensuring all parts are in order.");
    }
  }

  // Main method to demonstrate the Visitor Pattern
  public static void main(String[] args) {
    House house = new House();
    BuildingInspector inspector = new BuildingInspector();
    house.accept(inspector);
  }
}
```

Source: Github https://encr.pw/TZhQq

On the screen after the completion of the program according to the order of the rooms during the creation of the house:

1. Inspecting the quality of kitchen appliances and safety.
2. Checking living room space and ventilation.
3. Inspecting plumbing and hygiene conditions in the bathroom.
4. Performing an overall structural integrity check of the house.

What to consider when choosing design patterns?

When implementing design patterns, it is important **not to overcomplicate our project**. It is better to start with simple solutions, while design patterns are introduced to solve the problem of complexity.

Another important thing is to s**tick to coding standards and best practices**. Consistent coding styles make it easier for the team to understand and maintain the code, even when using design patterns.

To guide team members and other future code reviewers, it is necessary to **clearly document design decisions**. Although design patterns can make maintenance easier, improper implementation can introduce errors. Therefore, we should always use comprehensive testing to ensure the correct implementation of our design patterns.

Design patterns in code refactoring – how to use them wisely?

If the chosen design pattern does not provide the expected benefits or if the requirements change, you need to be ready to refactor the code because design patterns should adapt to the changing needs of the project.

When approaching refactoring, **it is certainly useful to understand the entire software engineering system holistically so that we can consciously decide whether we want to rewrite the system and at what scale**, e.g., are there any performance issues? Is the problem simply that, for example, we would like to use a new version of a library that has an entirely different interface?

If the decision is made, for example, that we are rewriting the module considering the current structure, and the patterns are used thoughtfully, it is worth using them. They provide an opportunity to extend and expand the project without too much time investment.

On the other hand, introducing patterns from scratch into a project that is written haphazardly and incomprehensibly can result in us essentially starting a new project from scratch. That is, we first spend several months understanding an older version of the project and then determine that sometimes 3 or 4 software design techniques like design patterns could solve the underlying problems. This means that from the very beginning, when we create the first stage of the project, the MVP with the most important functions for the customer, we use several patterns, and implement the next modules according to them.

Summary

In this article, we have discussed just some of the popular Java design patterns, such as creative, structural, and behavioral, and explained how they can be applied to Java programming, but there are also many other patterns. The better you understand and use these software design patterns in your daily work, you can improve the quality of your Java code, make it easier to maintain, and become a more skilled and efficient programmer.

If you would like to talk to experienced software engineers about their coding, migration, or refactoring practices, contact us. We will be happy to discuss our experiences and advise on effective solutions for your project.