Chapter 8: Design Patterns

Soumia Layachi

December 5, 2025

1 Introduction

In software development, some problems appear again and again: object creation, communication between modules, optimizing structure, etc. An experienced developer learns to recognize these recurring problems and apply proven solutions.

Design patterns, made popular by the famous Gang of Four (Gamma, Helm, Johnson, Vlissides), provide:

- reusable generic solutions,
- a clear and consistent architecture,
- reduced coupling,
- better modularity and maintainability.

2 What Is a Design Pattern?

A design pattern is not:

- an algorithm,
- ready-to-use code,
- an analysis method.

A design pattern **is**:

- a standardized solution to a common problem,
- an abstract model, independent of programming languages,

- a set of roles, responsibilities, and relationships between classes/objects,
- a guide to help structure software cleanly.

Definitions from authors:

- Coad (1992): a reusable abstraction of a set of classes.
- Appleton (1997): a rule connecting a context, a recurring problem, and a solution.
- Aarsten (1996): a group of cooperating objects that solve a design problem.

2.1 Benefits and Drawbacks

Benefits

- Reduced coupling between classes
- Faster development
- Easier maintenance
- A common language among developers
- Reliable solutions → fewer errors

Drawbacks

- Hard to recognize at first
- Require experience and practice
- The 23 GoF patterns can be a lot to learn
- Some patterns depend on others
- Using patterns everywhere is not always necessary

3 How to Use a Pattern Correctly?

To apply a pattern properly, you should document:

1. The Problem

- What recurring issue needs to be solved?
- What is the goal of the pattern?
- Give an illustrative example.

2. The Solution

- UML structure (classes, interfaces)
- Role descriptions
- Collaboration between objects

3. Application Domain

- When should it be used?
- What design issues does it avoid?
- How to recognize the situation?

4. Consequences

- Benefits
- Possible limitations

5. Implementation

- Practical details
- Language-related problems
- Example code

4 GoF Classification of Design Patterns

The 23 GoF patterns are divided into three categories:

4.1 Creational Patterns

They define how to create objects while hiding complexity. Goal: hide creation details and improve flexibility.

Examples:

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

4.2 Structural Patterns

They explain how to organize and combine classes and objects. Goal: create flexible and extensible structures.

Examples:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy
- Flyweight

4.3 Behavioral Patterns

They define how objects interact with each other. Goal: distribute responsibilities intelligently.

Examples:

- Strategy
- Observer

- State
- Command
- Chain of Responsibility
- Iterator
- Mediator
- Visitor
- Template Method

5 Patterns by Scope

5.1 Class Scope

Based on:

- static relationships,
- inheritance,
- class organization.

5.2 Object Scope

Based on:

- dynamic relationships between objects,
- composition,
- delegation.

6 Example of a Pattern

6.1 Singleton Pattern

Problem: We want to guarantee that a class has only one instance in the entire program (e.g., configuration, logs, database connections).

Solution:

- Private constructor

- Static method returning the unique instance
- Internal static attribute to store the instance

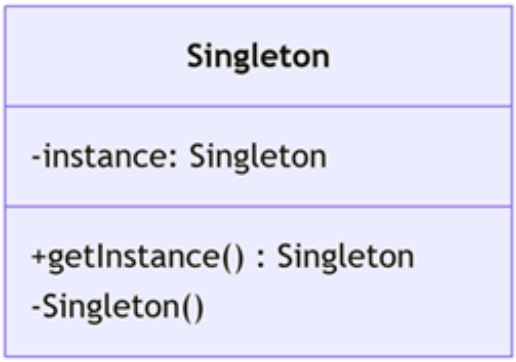

Figure 1: Singleton Pattern.

Code Example

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {} // Private constructor

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

Uses: configuration, logging, database connections.

6.2 Structural Pattern Example: Decorator

Problem: Add or remove features dynamically from an object without modifying its class.

Solution: Wrap the object inside a decorator that adds behavior (like nested Russian dolls).

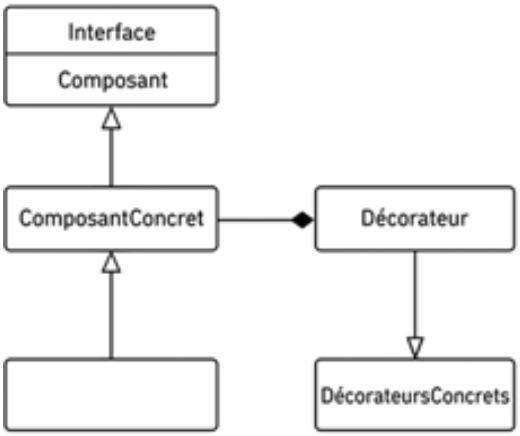

Figure 2: decorator Pattern.

Component Interface

```
public interface Boisson {
    String getDescription();
    double getPrix();
}
```

Concrete Component

```
public class Cafe implements Boisson {
    public String getDescription() {
        return "Caf ";
    }

    public double getPrix() {
        return 5.0;
    }
}
```

Abstract Decorator

```
public abstract class BoissonDecorateur implements
   Boisson {
    protected Boisson boisson;
```

```
    public BoissonDecorateur(Boisson boisson) {
        this.boisson = boisson;
    }
}
```

Concrete Decorators

Milk:

```
public class Lait extends BoissonDecorateur {
    public Lait(Boisson boisson) {
        super(boisson);
    }
    public String getDescription() {
        return boisson.getDescription() + ", lait";
    }
    public double getPrix() {
        return boisson.getPrix() + 1.0;
    }
}
```

Sugar:

```
public class Sucre extends BoissonDecorateur {
    public Sucre(Boisson boisson) {
        super(boisson);
    }
    public String getDescription() {
        return boisson.getDescription() + ", sucre";
    }
    public double getPrix() {
        return boisson.getPrix() + 0.5;
    }
}
```

Dynamic Composition

```
public class Test {
    public static void main(String[] args) {
        Boisson b = new Cafe();     // Simple coffee
        b = new Lait(b);            // Add milk
        b = new Sucre(b);           // Add sugar
        b = new Sucre(b);           // Add more sugar
```

```
        System.out.println(b.getDescription());
        System.out.println("Prix : " + b.getPrix());
    }
}
```

6.3 Behavioral Pattern Example: Observer

Problem: When an object's state changes, other objects must be notified automatically, without tight coupling.

Solution: Create a Subject → Observers relationship.

```
interface Observateur {
    void mettreAJour(int nouvelEtat);
}

class Sujet {
    private List<Observateur> obs = new ArrayList<>();
    private int etat;

    void ajouter(Observateur o) { obs.add(o); }
    void retirer(Observateur o) { obs.remove(o); }

    void setEtat(int e) {
        etat = e;
        notifier();
    }

    void notifier() {
        for (Observateur o : obs)
            o.mettreAJour(etat);
    }
}
```

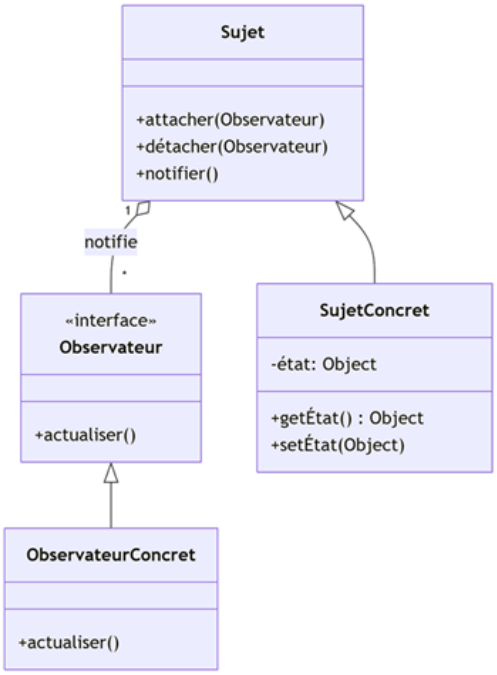

Figure 3: Observer Pattern.

7 Conclusion

Design patterns are essential tools for:

- improving software quality,
- reducing coupling,
- increasing flexibility,
- making systems easier to maintain and evolve.

They are not mandatory, but they are crucial in professional software development.