SOFTWARE DESIGN PATTERNS

A Comprehensive Guide to Creational, Structural, and Behavioral Patterns

Enterprise Software Architecture Series

Version 3.0 | January 2025

Documentation for Software Engineers

Best Practices and Implementation Guidelines

Page 1 of 15

Table of Contents

1. INTRODUCTION TO DESIGN PATTERNS

1.1 What Are Design Patterns?

Design patterns are **proven solutions** to recurring software design problems. They represent best practices evolved over time by experienced software developers.

> **Key Characteristics:**
>
> - **Reusable:** Applicable across multiple projects
> - **Language Agnostic:** Can be implemented in any programming language
> - **Tested Solutions:** Battle-tested approaches
> - **Communication Tools:** Provide common vocabulary for developers

1.2 Historical Context

The concept originated in architecture with Christopher Alexander's 1977 book "A Pattern Language." The software community adopted this concept, culminating in the seminal 1994 book "Design Patterns: Elements of Reusable Object-Oriented Software" by the **Gang of Four (GoF)**:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

1.3 Benefits and Applications

Accelerated Development

Reduce time spent on design decisions with proven solutions

Improved Code Quality

Promote clean, maintainable, and scalable code architecture

Enhanced Communication

Common vocabulary among team members speeds up collaboration

Reduced Risk

Use proven solutions instead of inventing new

Better Scalability

Patterns support system growth and future changes

Maintainability

Well-structured code is easier to debug and

ones		extend

2. PATTERN CLASSIFICATION

DESIGN PATTERNS RELATIONSHIP

```
┌──────────────────────────────────────────┐
│             DESIGN PATTERNS              │
├─────────────┬─────────────┬──────────────┤
│ Creational  │ Structural  │ Behavioral   │
│             │             │              │
│ • Singleton │ • Adapter   │ • Observer   │
│ • Factory   │ • Decorator │ • Strategy   │
│ • Builder   │ • Facade    │ • Command    │
│ • Prototype │ • Proxy     │ • Iterator   │
└─────────────┴─────────────┴──────────────┘
```

2.1 Creational Patterns

Focus: Object creation mechanisms

Core Principle: Decouple object creation from usage

Primary Patterns: Singleton, Factory Method, Abstract Factory, Builder, Prototype

2.2 Structural Patterns

Focus: Object composition and relationships

Core Principle: Simplify relationships between entities

Primary Patterns: Adapter, Decorator, Facade, Proxy, Composite

2.3 Behavioral Patterns

Focus: Object interaction and responsibility distribution

Core Principle: Define communication patterns between objects

Primary Patterns: Observer, Strategy, Command, Iterator, Template Method

> **Note:** Patterns are not mutually exclusive. Many real-world solutions combine multiple patterns to solve complex problems.

3. CREATIONAL PATTERNS

3.1 Singleton Pattern

Intent: Ensure a class has only one instance and provide global access to it.

> **Use Cases:** Database connections, Configuration managers, Logger instances, Caching mechanisms

Java Implementation:

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

    // Private constructor to prevent instantiation
    private Singleton() {
        // Initialization code
    }

    // Public method to provide access to instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Business methods
    public void businessMethod() {
        System.out.println("Singleton business logic");
    }
}
```

Pros	Cons
• Controlled access to single instance	• Global state can lead to hidden dependencies
• Reduced memory usage	• Difficult to unit test
• Thread-safe (with proper implementation)	• Violates Single Responsibility Principle

3.2 Factory Method Pattern

Intent: Define an interface for creating objects, but let subclasses decide which class to instantiate.

```
// Product Interface
interface Document {
    void open();
    void save();
}

// Concrete Products
class PDFDocument implements Document {
    public void open() { System.out.println("Opening PDF"); }
    public void save() { System.out.println("Saving PDF"); }
}

class WordDocument implements Document {
    public void open() { System.out.println("Opening Word"); }
    public void save() { System.out.println("Saving Word"); }
}

// Creator
abstract class DocumentCreator {
    public abstract Document createDocument();

    public void newDocument() {
        Document doc = createDocument();
        doc.open();
    }
}

// Concrete Creators
class PDFCreator extends DocumentCreator {
    public Document createDocument() {
        return new PDFDocument();
    }
}

class WordCreator extends DocumentCreator {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

3.3 Abstract Factory Pattern

Intent: Provide an interface for creating families of related objects without specifying concrete classes.

GUI Factory Example:

```
// Abstract Products
interface Button {
    void render();
}

interface Checkbox {
    void check();
}

// Concrete Products for Windows
class WindowsButton implements Button {
    public void render() {
        System.out.println("Rendering Windows style button");
    }
}

class WindowsCheckbox implements Checkbox {
    public void check() {
        System.out.println("Windows checkbox checked");
    }
}

// Concrete Products for Mac
class MacButton implements Button {
    public void render() {
        System.out.println("Rendering Mac style button");
    }
}

class MacCheckbox implements Checkbox {
    public void check() {
        System.out.println("Mac checkbox checked");
    }
}

// Abstract Factory
interface GUIFactory {
```

```
    Button createButton();
    Checkbox createCheckbox();
}

// Concrete Factories
class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }

    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}
```

3.4 Builder Pattern

Intent: Separate construction of complex object from its representation.

```
class Computer {
    private String CPU;
    private String RAM;
    private String storage;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    // Builder class
    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
```

```
        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}

// Usage
Computer gamingPC = new Computer.Builder()
    .setCPU("Intel i9")
    .setRAM("32GB DDR5")
    .setStorage("2TB SSD")
    .build();
```

3.5 Prototype Pattern

Intent: Create new objects by copying existing objects (prototypes).

```
abstract class Shape implements Cloneable {
    private String id;
    protected String type;

    abstract void draw();

    public String getType() {
        return type;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}

class Rectangle extends Shape {
    public Rectangle() {
        type = "Rectangle";
    }

    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

class Circle extends Shape {
```

```
    public Circle() {
        type = "Circle";
    }

    public void draw() {
        System.out.println("Drawing Circle");
    }
}

// Prototype registry
class ShapeCache {
    private static Map shapeMap = new HashMap<>();

    static {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId(), circle);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("2");
        shapeMap.put(rectangle.getId(), rectangle);
    }

    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }
}
```

Creational Patterns Comparison

Pattern	When to Use	Complexity
Singleton	Exactly one instance needed globally	Low
Factory Method	Subclasses decide object creation	Medium
Abstract Factory	Families of related objects needed	High
Builder	Complex object with many parameters	Medium
Prototype	Object creation is expensive	Low

4. STRUCTURAL PATTERNS

4.1 Adapter Pattern

Intent: Convert interface of a class into another interface clients expect.

```
// Legacy system interface
interface LegacyPrinter {
    void printDocument(String content);
}

// New system interface
interface ModernPrinter {
    void print(String text, String paperSize);
}

// Adapter
class PrinterAdapter implements ModernPrinter {
    private LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter legacyPrinter) {
        this.legacyPrinter = legacyPrinter;
    }

    public void print(String text, String paperSize) {
        // Adapt the interface
        String adaptedContent = "Paper: " + paperSize + "\n" + text;
        legacyPrinter.printDocument(adaptedContent);
    }
}

// Usage
LegacyPrinter oldPrinter = new LegacyPrinterImpl();
ModernPrinter adapter = new PrinterAdapter(oldPrinter);
adapter.print("Hello World", "A4");
```

4.2 Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically.

```
// Component
```

```
interface Coffee {
    double getCost();
    String getDescription();
}

// Concrete Component
class SimpleCoffee implements Coffee {
    public double getCost() {
        return 1.0;
    }

    public String getDescription() {
        return "Simple coffee";
    }
}

// Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    public double getCost() {
        return decoratedCoffee.getCost();
    }

    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
}

// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public double getCost() {
        return super.getCost() + 0.5;
    }

    public String getDescription() {
        return super.getDescription() + ", milk";
    }
}
```

```
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getDescription() {
        return super.getDescription() + ", sugar";
    }
}
```

4.3 Facade Pattern

Intent: Provide unified interface to a set of interfaces in a subsystem.

```
// Complex subsystem classes
class CPU {
    public void freeze() { System.out.println("CPU freezing"); }
    public void jump(long position) { System.out.println("Jumping to position " +
position); }
    public void execute() { System.out.println("CPU executing"); }
}

class Memory {
    public void load(long position, byte[] data) {
        System.out.println("Loading data at position " + position);
    }
}

class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("Reading from hard drive");
        return new byte[size];
    }
}

// Facade
class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void start() {
        System.out.println("Computer starting...");
        cpu.freeze();
        memory.load(0, hardDrive.read(0, 1024));
        cpu.jump(0);
        cpu.execute();
        System.out.println("Computer started successfully");
```

```
    }
}

// Client code
public class Client {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start(); // Simple interface to complex subsystem
    }
}
```

4.4 Proxy Pattern

Intent: Provide surrogate or placeholder for another object to control access.

```
interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename + " from disk...");
    }

    public void display() {
        System.out.println("Displaying " + filename);
    }
}

class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
```

```
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}

// Usage - image loads only when displayed
Image image1 = new ProxyImage("photo1.jpg");
Image image2 = new ProxyImage("photo2.jpg");

image1.display(); // Loading and displaying
image1.display(); // Already loaded, just displaying
image2.display(); // Loading and displaying
```

4.5 Composite Pattern

Intent: Compose objects into tree structures to represent part-whole hierarchies.

```
// Component
interface FileSystemComponent {
    void showDetails();
    int getSize();
}

// Leaf
class File implements FileSystemComponent {
    private String name;
    private int size;

    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }

    public void showDetails() {
        System.out.println("File: " + name + " (" + size + " bytes)");
    }

    public int getSize() {
        return size;
    }
}

// Composite
class Directory implements FileSystemComponent {
    private String name;
    private List components = new ArrayList<>();

    public Directory(String name) {
        this.name = name;
    }

    public void addComponent(FileSystemComponent component) {
        components.add(component);
    }

    public void removeComponent(FileSystemComponent component) {
        components.remove(component);
```

```java
    }

    public void showDetails() {
        System.out.println("Directory: " + name);
        for (FileSystemComponent component : components) {
            component.showDetails();
        }
    }

    public int getSize() {
        int totalSize = 0;
        for (FileSystemComponent component : components) {
            totalSize += component.getSize();
        }
        return totalSize;
    }
}

// Usage
Directory root = new Directory("Root");
File file1 = new File("file1.txt", 100);
File file2 = new File("file2.txt", 200);

Directory subDir = new Directory("SubDirectory");
File file3 = new File("file3.txt", 300);

subDir.addComponent(file3);
root.addComponent(file1);
root.addComponent(file2);
root.addComponent(subDir);

root.showDetails();
System.out.println("Total size: " + root.getSize() + " bytes");
```

Structural Patterns Summary:

- **Adapter:** Makes incompatible interfaces work together
- **Decorator:** Adds functionality without subclassing
- **Facade:** Simplifies complex subsystem interfaces
- **Proxy:** Controls access to expensive objects
- **Composite:** Treats individual and composite objects uniformly

5. BEHAVIORAL PATTERNS

5.1 Observer Pattern

Intent: Define one-to-many dependency between objects so when one changes state, all dependents are notified.

```
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(String message);
}

// Subject interface
interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

// Concrete Subject
class NewsAgency implements Subject {
    private List observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
```

```
        }
    }
}

// Concrete Observers
class NewsChannel implements Observer {
    private String news;
    private String name;

    public NewsChannel(String name) {
        this.name = name;
    }

    public void update(String news) {
        this.news = news;
        display();
    }

    public void display() {
        System.out.println(name + " received news: " + news);
    }
}

// Usage
NewsAgency agency = new NewsAgency();
NewsChannel channel1 = new NewsChannel("CNN");
NewsChannel channel2 = new NewsChannel("BBC");

agency.registerObserver(channel1);
agency.registerObserver(channel2);

agency.setNews("Breaking News: Design Patterns are Awesome!");
```

5.2 Strategy Pattern

Intent: Define family of algorithms, encapsulate each one, and make them interchangeable.

```
// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
```

```
    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}

class BitcoinPayment implements PaymentStrategy {
    private String walletAddress;

    public BitcoinPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Bitcoin");
    }
}

// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

5.3 Command Pattern

Intent: Encapsulate a request as an object, allowing parameterization and queuing of requests.

```
// Command interface
interface Command {
    void execute();
    void undo();
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}

// Concrete Commands
class TurnOnLightCommand implements Command {
    private Light light;

    public TurnOnLightCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }

    public void undo() {
        light.turnOff();
    }
}

class TurnOffLightCommand implements Command {
    private Light light;

    public TurnOffLightCommand(Light light) {
        this.light = light;
    }
```

```
    public void execute() {
        light.turnOff();
    }

    public void undo() {
        light.turnOn();
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }

    public void pressUndo() {
        command.undo();
    }
}

// Usage
Light livingRoomLight = new Light();
Command turnOn = new TurnOnLightCommand(livingRoomLight);
Command turnOff = new TurnOffLightCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(turnOn);
remote.pressButton(); // Turns light on

remote.setCommand(turnOff);
remote.pressButton(); // Turns light off
remote.pressUndo();   // Turns light back on
```

5.4 Iterator Pattern

Intent: Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation.

```java
// Iterator interface
interface Iterator {
    boolean hasNext();
    T next();
}

// Aggregate interface
interface Container {
    Iterator getIterator();
}

// Concrete Aggregate
class NameRepository implements Container {
    private String[] names = {"Robert", "John", "Julie", "Lora"};

    public Iterator getIterator() {
        return new NameIterator();
    }

    // Concrete Iterator
    private class NameIterator implements Iterator {
        int index;

        public boolean hasNext() {
            return index < names.length;
        }

        public String next() {
            if (this.hasNext()) {
                return names[index++];
            }
            return null;
        }
    }
}

// Usage
NameRepository namesRepository = new NameRepository();
Iterator iterator = namesRepository.getIterator();

while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println("Name: " + name);
}
```

5.5 Template Method Pattern

Intent: Define skeleton of an algorithm in operation, deferring some steps to subclasses.

```
abstract class DataProcessor {
    // Template method - defines algorithm structure
    public final void process() {
        readData();
        processData();
        saveData();
        if (hook()) {
            additionalOperation();
        }
    }

    // Concrete steps
    private void readData() {
        System.out.println("Reading data from source...");
    }

    private void saveData() {
        System.out.println("Saving processed data...");
    }

    // Abstract steps - to be implemented by subclasses
    protected abstract void processData();

    // Hook - optional step
    protected boolean hook() {
        return false;
    }

    protected void additionalOperation() {
        // Default implementation does nothing
    }
}

// Concrete implementations
class CSVProcessor extends DataProcessor {
    protected void processData() {
        System.out.println("Processing CSV data...");
    }
}
```

```
class XMLProcessor extends DataProcessor {
    protected void processData() {
        System.out.println("Processing XML data...");
    }

    // Override hook
    protected boolean hook() {
        return true;
    }

    protected void additionalOperation() {
        System.out.println("Performing XML validation...");
    }
}

// Usage
DataProcessor csvProcessor = new CSVProcessor();
csvProcessor.process();

System.out.println("\n---\n");

DataProcessor xmlProcessor = new XMLProcessor();
xmlProcessor.process();
```

Behavioral Patterns Benefits:

- **Observer:** Loose coupling between subject and observers
- **Strategy:** Easy algorithm swapping at runtime
- **Command:** Parameterize objects with operations
- **Iterator:** Uniform traversal of different collections
- **Template Method:** Code reuse through inheritance

6. PATTERN SELECTION GUIDELINES

Decision Matrix for Pattern Selection

Problem Type	Recommended Pattern	When to Use
Need single instance	Singleton	Global access to shared resource
Complex object creation	Builder	Many construction parameters
Family of related objects	Abstract Factory	Cross-platform compatibility
Incompatible interfaces	Adapter	Integrating legacy code
Add functionality dynamically	Decorator	Runtime behavior extension
Simplify complex system	Facade	Hide subsystem complexity
One-to-many dependency	Observer	Event-driven systems
Multiple algorithms	Strategy	Switch behaviors at runtime

Anti-Patterns to Avoid

Common Mistakes:

- **Golden Hammer:** Using same pattern for every problem
- **Over-engineering:** Adding patterns where not needed
- **Singleton Abuse:** Using singleton for everything
- **Pattern Mania:** Forcing patterns into code

Best Practices

1. **Understand the Problem First:** Don't start with patterns
2. **Keep It Simple:** Use patterns only when they add value
3. **Refactor Gradually:** Introduce patterns during refactoring
4. **Team Consensus:** Ensure team understands chosen patterns
5. **Document Usage:** Comment why pattern was chosen

Performance Considerations

Performance Tips:

- Some patterns add overhead (Proxy, Decorator)
- Consider memory usage with object proliferation
- Balance flexibility with performance needs
- Profile before optimizing pattern usage

7. CONCLUSION & REFERENCES

Summary

Design patterns provide proven solutions to common software design problems. They:

- Improve code maintainability and scalability
- Enhance team communication with common vocabulary
- Reduce development time through reusable solutions
- Promote best practices and clean architecture

Key Takeaways

> **Remember:**
>
> 1. Patterns are tools, not goals
> 2. Start simple and refactor into patterns when needed
> 3. Consider the trade-offs of each pattern
> 4. Patterns work best when the team understands them
> 5. Combine patterns to solve complex problems

References

- **Primary Reference:** Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- **Practical Guide:** Freeman, E., & Freeman, E. (2004). *Head First Design Patterns*. O'Reilly Media.
- **Modern Approaches:** Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.
- **Online Resources:**
  - Refactoring.guru - Design Patterns
  - SourceMaking - Design Patterns
  - GeeksforGeeks - Design Patterns

Further Learning

Architectural Patterns

MVC, MVP, MVVM, Microservices

Concurrency Patterns

Thread Pool, Producer-Consumer, Future

Cloud Patterns

Circuit Breaker, Retry, Bulkhead

Document Version: 3.0 | **Last Updated:** January 2025