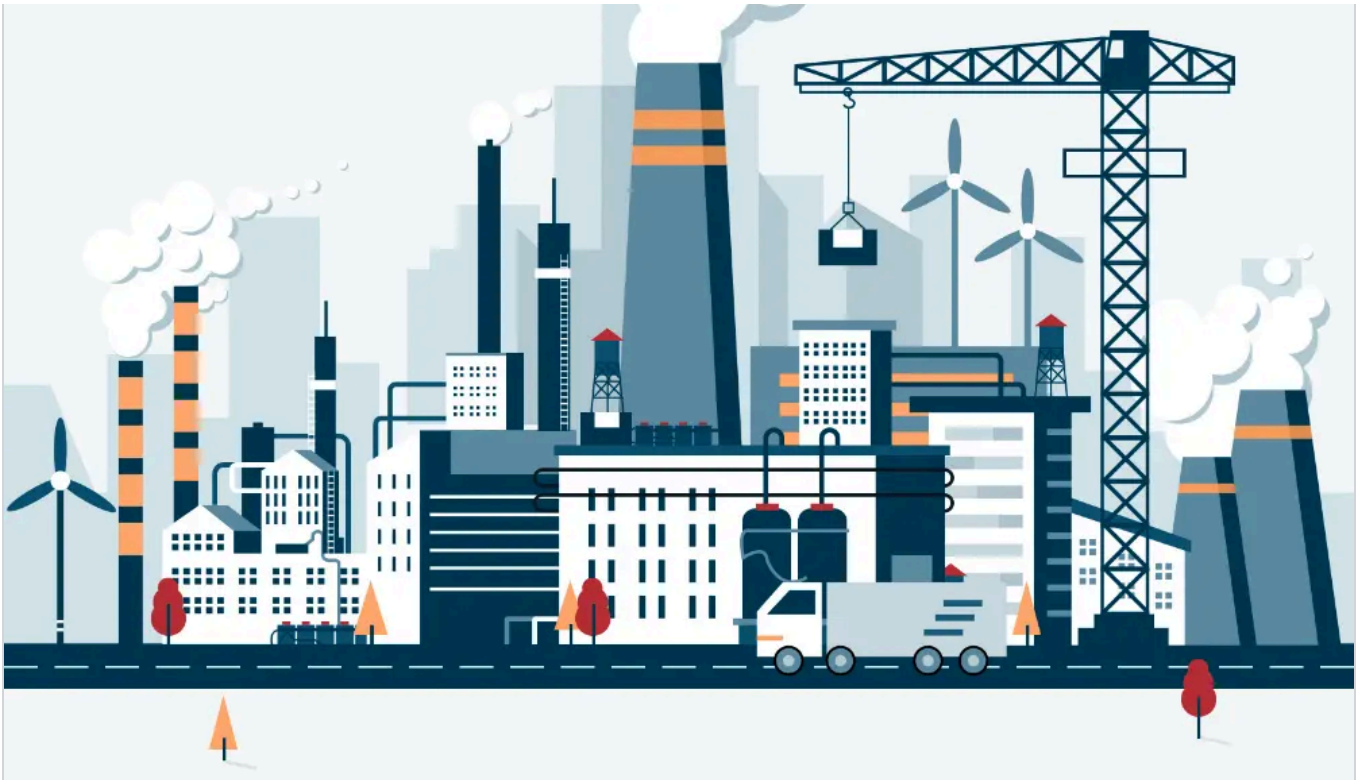
A Comprehensive Guide to Design Patterns in Java

Unlock the Power of Reusable Code with Design Patterns

Published April 14, 2023 • 20 min read



Contents

Amit kumar

Results-driven Software Engineer specializing in digital transformations, API development, and cloud-based solutions. Skilled in Java, microservices, and Agile methodologies. Adept at optimizing processes, enhancing security, and reducing deployment time. Passionate about collaborating with cross-functional teams to deliver innovative solutions that drive growth and create exceptional user experiences. Excited about the future of tech and actively pursuing knowledge in Artificial Intelligence (AI) and Machine Learning (ML) to stay at the forefront of industry innovation.

SERIES

Java Uncovered: Exploring the Infinite Possibilities

TAGS

#java #design-patterns #design-principles #design-and-architecture

Introduction to Design Patterns

Design patterns are reusable solutions to common problems that arise during software design. They provide a standardized way to approach these problems, making it easier for developers to create efficient and maintainable code. In this guide, we'll explore some of the most common design patterns in Java, complete with examples and code snippets.

Benefits of Design Patterns and Real-World Use Cases

Design patterns offer several advantages to software developers, including:

- **Improved code readability and maintainability**: Design patterns provide a standard way of solving common problems, making the code easier to understand and maintain.

- **Enhanced modularity and reusability**: Design patterns encourage modular and reusable code, allowing you to use the same solution in different parts of your application.
- **Faster development**: By applying design patterns, developers can save time and effort when developing new features or fixing issues since they can leverage proven solutions instead of reinventing the wheel.
- **Easier communication among team members**: Design patterns provide a common vocabulary for developers, making it easier to discuss and understand the structure and behavior of the code.

Real-world use cases of design patterns:

- **Singleton**: Ensuring a single instance of a database connection or a logging class.
- **Factory Method**: Creating objects for different file formats like CSV, JSON, or XML parsers.
- **Observer**: Implementing a publish-subscribe model in a messaging system or updating multiple UI components when data changes.
- **Strategy**: Supporting different payment methods in an e-commerce application or providing different sorting algorithms in a text editor.

Creational Patterns

Creational design patterns deal with the process of object creation. They help to abstract the instantiation process and make the code more flexible, reusable, and maintainable.

Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This is useful when you need a single object to coordinate actions across the system.

Example: A Logger class that handles logging messages for an application.

```
public class Logger {
    private static Logger instance;
    private Logger() {}

    public static synchronized Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

Usage:

```
Logger logger = Logger.getInstance();
logger.log("This is a log message.");
```

Factory Pattern

The Factory pattern provides an interface for creating objects in a super class, allowing subclasses to decide which class to instantiate. It promotes loose coupling by eliminating the need for clients to know about the concrete classes.

Example: A ShapeFactory class that creates different shapes based on the input.

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
```

```
    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class ShapeFactory {
    public Shape createShape(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:

```
ShapeFactory shapeFactory = new ShapeFactory();
Shape circle = shapeFactory.createShape("CIRCLE");
Shape rectangle = shapeFactory.createShape("RECTANGLE");

circle.draw();
rectangle.draw();
```

Abstract Factory Pattern

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows you to switch between different product families at runtime.

Example: A GUI factory that creates different button and checkbox objects for different platforms.

```
public interface Button {
    void render();
}

public interface Checkbox {
    void check();
}

public class WindowsButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering a Windows button.");
    }
}

public class MacOSButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering a macOS button.");
    }
}

public class WindowsCheckbox implements Checkbox {
    @Override
    public void check() {
        System.out.println("Checking a Windows checkbox.");
    }
}

public class MacOSCheckbox implements Checkbox {
    @Override
    public void check() {
```

```
        System.out.println("Checking a macOS checkbox.");
    }
}

public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

public class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }

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

public class MacOSFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacOSButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new MacOSCheckbox();
    }
}
```

Usage:

```
GUIFactory factory;
String platform = "macos"; // Can be changed to "windows"
```

```
if (platform.equals("windows")) {
    factory = new WindowsFactory();
} else {
    factory = new MacOSFactory();
}

Button button = factory.createButton();
Checkbox checkbox = factory.createCheckbox();

button.render();
checkbox.check();
```

Builder Pattern

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It is useful when dealing with objects that have many optional or required parameters.

Example: A Computer class with a nested builder class to create a computer object.

```
public class Computer {
    private String processor;
    private int ram;
    private int storage;

    private Computer(Builder builder) {
        this.processor = builder.processor;
        this.ram = builder.ram;
        this.storage = builder.storage;
    }

    public static class Builder {
        private String processor;
        private int ram;
        private int storage;

        public Builder(String processor) {
```

```
            this.processor = processor;
        }

        public Builder setRam(int ram) {
            this.ram = ram;
            return this;
        }

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

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

Usage:

```
Computer computer = new Computer.Builder("Intel Core i9")
        .setRam(32)
        .setStorage(512)
        .build();
```

Prototype Pattern

The Prototype pattern involves creating new objects by cloning existing ones. It is useful when object creation is expensive or complex, and you want to avoid duplicating the effort.

Example: A Shape class that can be cloned to create new shape objects.

```
public abstract class Shape implements Cloneable {
    private String id;
```

```
    public String getId() {
        return id;
    }

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

    public abstract void draw();

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

public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
}

public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class ShapeCache {
    private static Map<String, Shape> shapeMap = new HashMap<>();
```

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

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

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

Usage:

```
ShapeCache.loadCache();

Shape clonedCircle = ShapeCache.getShape("1");
Shape clonedRectangle = ShapeCache.getShape("2");

clonedCircle.draw();
clonedRectangle.draw();
```

Structural Patterns

Structural patterns are concerned with the composition of classes and objects. They help you to create larger structures while maintaining the relationships between components.

Adapter Pattern

The Adapter pattern allows you to convert the interface of a class into another interface that clients expect. It's useful when you want to use an existing class with an incompatible interface.

Example: Adapting a LegacyRectangle class to the Shape interface.

```
public interface Shape {
    void draw(int x1, int y1, int x2, int y2);
}

public class LegacyRectangle {
    public void drawRectangle(int x1, int y1, int width, int heig
        System.out.println("Drawing rectangle: (" + x1 + ", " + y
    }
}

public class RectangleAdapter implements Shape {
    private LegacyRectangle legacyRectangle;

    public RectangleAdapter(LegacyRectangle legacyRectangle) {
        this.legacyRectangle = legacyRectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int width = x2 - x1;
        int height = y2 - y1;
        legacyRectangle.drawRectangle(x1, y1, width, height);
    }
}
```

Usage:

```
LegacyRectangle legacyRectangle = new LegacyRectangle();
Shape rectangle = new RectangleAdapter(legacyRectangle);
rectangle.draw(10, 20, 30, 40);
```

Bridge Pattern

The Bridge pattern decouples an abstraction from its implementation, allowing them to vary independently. It's useful when you want to separate the interface from its implementation and allow them to evolve independently.

Example: A Shape class with different rendering methods.

```
public interface Renderer {
    void renderShape(Shape shape);
}

public class VectorRenderer implements Renderer {
    @Override
    public void renderShape(Shape shape) {
        System.out.println("Rendering " + shape.getName() + " usi
    }
}

public class RasterRenderer implements Renderer {
    @Override
    public void renderShape(Shape shape) {
        System.out.println("Rendering " + shape.getName() + " usi
    }
}

public abstract class Shape {
    private Renderer renderer;

    public Shape(Renderer renderer) {
        this.renderer = renderer;
    }

    public abstract String getName();

    public void draw() {
        renderer.renderShape(this);
    }
}
```

```
public class Circle extends Shape {
    public Circle(Renderer renderer) {
        super(renderer);
    }

    @Override
    public String getName() {
        return "circle";
    }
}

public class Rectangle extends Shape {
    public Rectangle(Renderer renderer) {
        super(renderer);
    }
}
```

Usage :

```
Renderer vectorRenderer = new VectorRenderer();
Renderer rasterRenderer = new RasterRenderer();

Shape vectorCircle = new Circle(vectorRenderer);
Shape rasterRectangle = new Rectangle(rasterRenderer);

vectorCircle.draw();
rasterRectangle.draw();
```

Composite Pattern

The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects and compositions uniformly.

Example: A graphic class with composite and leaf components.y

```
public interface Graphic {
    void draw();
}

public class CompositeGraphic implements Graphic {
    private List<Graphic> graphics = new ArrayList<>();

    public void add(Graphic graphic) {
        graphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        graphics.remove(graphic);
    }

    @Override
    public void draw() {
        for (Graphic graphic : graphics) {
            graphic.draw();
        }
    }
}

public class Circle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
}

public class Rectangle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

Usage :

```
CompositeGraphic compositeGraphic = new CompositeGraphic();
Circle circle = new Circle();
Rectangle rectangle = new Rectangle();

compositeGraphic.add(circle);
compositeGraphic.add(rectangle);

compositeGraphic.draw();
```

Decorator Pattern

The Decorator pattern allows you to attach additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

Example: Adding borders and colors to a basic rectangle.

```
public interface Shape {
    void draw();
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    @Override
    public void draw() {
```

```
            decoratedShape.draw();
        }
    }

    public class BorderDecorator extends ShapeDecorator {
        public BorderDecorator(Shape decoratedShape) {
            super(decoratedShape);
        }

        @Override
        public void draw() {
            super.draw();
            addBorder();
        }

        private void addBorder() {
            System.out.println("Adding a border.");
        }
    }

    public class ColorDecorator extends ShapeDecorator {
        public ColorDecorator(Shape decoratedShape) {
            super(decoratedShape);
        }

        @Override
        public void draw() {
            super.draw();
            fillColor();
        }

        private void fillColor() {
            System.out.println("Filling with color.");
        }
    }
```

Usage :

```
Shape rectangle = new Rectangle();
Shape borderRectangle = new BorderDecorator(rectangle);
Shape coloredBorderRectangle = new ColorDecorator(borderRectangle

coloredBorderRectangle.draw();
```

Facade Pattern

The Facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

Example: A computer class with a simplified interface for starting and shutting down.

```
public class CPU {
    public void start() {
        System.out.println("CPU started.");
    }

    public void shutDown() {
        System.out.println("CPU shut down.");
    }
}

public class Memory {
    public void load() {
        System.out.println("Memory loaded.");
    }

    public void clear() {
        System.out.println("Memory cleared.");
    }
}

public class HardDrive {
    public void read() {
```

```
        System.out.println("Hard drive reading.");
    }

    public void stop() {
        System.out.println("Hard drive stopped.");
    }
}

public 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() {
        cpu.start();
        memory.load();
        hardDrive.read();
    }

    public void shutDown() {
        hardDrive.stop();
        memory.clear();
        cpu.shutDown();
    }
}
```

Usage :

```
ComputerFacade computer = new ComputerFacade();
computer.start();
computer.shutDown();
```

Flyweight Pattern

The Flyweight pattern uses sharing to support large numbers of fine-grained objects efficiently. It's useful when you have a large number of objects that share a common state and can be replaced by a single shared object.

Example: A shape factory that reuses existing circle objects with the same color.

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    private String color;

    public Circle(String color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a " + color + " circle.");
    }
}

public class ShapeFactory {
    private static final Map<String, Shape> circleMap = new HashM

    public static Shape getCircle(String color) {
        Circle circle = (Circle) circleMap.get(color);

        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating a " + color + " circle."
        }

        return circle;
```

```
    }
}
```

Usage :

```
Shape redCircle1 = ShapeFactory.getCircle("red");
Shape redCircle2 = ShapeFactory.getCircle("red");
Shape blueCircle = ShapeFactory.getCircle("blue");

redCircle1.draw();
redCircle2.draw();
blueCircle.draw();
```

Proxy Pattern

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. It's useful when you want to add a layer of indirection between the client and the real object.

Example: An image proxy that loads the real image on demand.

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

public class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromFile(fileName);
    }

    private void loadFromFile(String fileName) {
        System.out.println("Loading image from file: " + fileName
    }
```

```
    @Override
    public void display() {
        System.out.println("Displaying image: " + fileName);
    }
}

public class ImageProxy implements Image {
    private String fileName;
    private RealImage realImage;

    public ImageProxy(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```

Usage :

```
Image image = new ImageProxy("example.jpg");
image.display(); // Loads the image and displays it
image.display(); // Displays the image without loading it again
```

Behavioral Patterns

Behavioral patterns define the ways in which objects interact and communicate with one another. They help to streamline the communication between components and promote flexible and maintainable systems.

Chain of Responsibility Pattern

The Chain of Responsibility pattern allows you to pass requests along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain.

Example: A logger with different log levels.

```
public abstract class Logger {
    protected int logLevel;
    protected Logger nextLogger;

    public void setNextLogger(Logger nextLogger) {
        this.nextLogger = nextLogger;
    }

    public void logMessage(int level, String message) {
        if (this.logLevel ≤ level) {
            write(message);
        }
        if (nextLogger ≠ null) {
            nextLogger.logMessage(level, message);
        }
    }

    protected abstract void write(String message);
}

public class ConsoleLogger extends Logger {
    public ConsoleLogger(int logLevel) {
        this.logLevel = logLevel;
    }

    @Override
    protected void write(String message) {
        System.out.println("Console Logger: " + message);
    }
}

public class FileLogger extends Logger {
```

```
    public FileLogger(int logLevel) {
        this.logLevel = logLevel;
    }

    @Override
    protected void write(String message) {
        System.out.println("File Logger: " + message);
    }
}

public class ErrorLogger extends Logger {
    public ErrorLogger(int logLevel) {
        this.logLevel = logLevel;
    }

    @Override
    protected void write(String message) {
        System.out.println("Error Logger: " + message);
    }
}
```

Usage :

```
Logger consoleLogger = new ConsoleLogger(1);
Logger fileLogger = new FileLogger(2);
Logger errorLogger = new ErrorLogger(3);

consoleLogger.setNextLogger(fileLogger);
fileLogger.setNextLogger(errorLogger);

consoleLogger.logMessage(1, "This is an informational message.");
consoleLogger.logMessage(2, "This is a warning message.");
consoleLogger.logMessage(3, "This is an error message.");
```

Command Pattern

The Command pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.

Example: A remote control for electronic devices.

```
public interface Command {
    void execute();
}

public class TurnOnCommand implements Command {
    private ElectronicDevice device;

    public TurnOnCommand(ElectronicDevice device) {
        this.device = device;
    }

    @Override
    public void execute() {
        device.turnOn();
    }
}

public class TurnOffCommand implements Command {
    private ElectronicDevice device;

    public TurnOffCommand(ElectronicDevice device) {
        this.device = device;
    }

    @Override
    public void execute() {
        device.turnOff();
    }
}

public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
```

```
        this.command = command;
    }

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

public interface ElectronicDevice {
    void turnOn();
    void turnOff();
}

public class Television implements ElectronicDevice {
    @Override
    public void turnOn() {
        System.out.println("Television turned on.");
    }

    @Override
    public void turnOff() {
        System.out.println("Television turned off.");
    }
}
```

Usage :

```
ElectronicDevice television = new Television();
Command turnOnCommand = new TurnOnCommand(television);
Command turnOffCommand = new TurnOffCommand(television);

RemoteControl remoteControl = new RemoteControl();
remoteControl.setCommand(turnOnCommand);
remoteControl.pressButton(); //Turns on the television

remoteControl.setCommand(turnOffCommand);
remoteControl.pressButton(); // Turns off the television
```

Interpreter Pattern

The Interpreter pattern defines a representation for a language's grammar and provides an interpreter to evaluate expressions in the language. It's useful when you want to create a simple language or parse a complex expression.

Example: A calculator that evaluates arithmetic expressions.

```
public interface Expression {
    int interpret();
}

public class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}

public class SubtractExpression implements Expression {
    private Expression left;
    private Expression right;

    public SubtractExpression(Expression left, Expression right)
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() - right.interpret();
    }
```

```
}

public class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    @Override
    public int interpret() {
        return number;
    }
}
```

Usage :

```
Expression left = new NumberExpression(5);
Expression right = new NumberExpression(3);

Expression addExpression = new AddExpression(left, right);
System.out.println("5 + 3 = " + addExpression.interpret());

Expression subtractExpression = new SubtractExpression(left, righ
System.out.println("5 - 3 = " + subtractExpression.interpret());
```

Iterator Pattern

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It's useful when you want to traverse a data structure without knowing its implementation details.

Example: A custom list with an iterator.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}

public interface Iterable<E> {
    Iterator<E> iterator();
}

public class CustomList<E> implements Iterable<E> {
    private List<E> items;

    public CustomList() {
        items = new ArrayList<>();
    }

    public void add(E item) {
        items.add(item);
    }

    public E get(int index) {
        return items.get(index);
    }

    public int size() {
        return items.size();
    }

    @Override
    public Iterator<E> iterator() {
        return new CustomListIterator();
    }

    private class CustomListIterator implements Iterator<E> {
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentIndex < items.size();
```

```
        }

        @Override
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return items.get(currentIndex++);
        }
    }
}
```

Usage :

```
CustomList<String> names = new CustomList<>();
names.add("Alice");
names.add("Bob");
names.add("Carol");

Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Mediator Pattern

The Mediator pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and allows their interaction to be changed independently.

Example: A chat room with users.

```
public interface Mediator {
    void sendMessage(String message, User user);
}
```

```
public class ChatRoom implements Mediator {
    private List<User> users = new ArrayList<>();

    public void addUser(User user) {
        users.add(user);
    }

    @Override
    public void sendMessage(String message, User sender) {
        for (User user : users) {
            if (user != sender) {
                user.receiveMessage(message);
            }
        }
    }
}

public abstract class User {
    protected Mediator mediator;
    protected String name;

    public User(Mediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }

    public void sendMessage(String message) {
        mediator.sendMessage(message, this);
    }

    public abstract void receiveMessage(String message);
}

public class ConcreteUser extends User {
    public ConcreteUser(Mediator mediator, String name) {
        super(mediator, name);
    }

    @Override
    public void receiveMessage(String message) {
```

```
            System.out.println(name + " received: " + message);
        }
    }
```

Usage :

```
ChatRoom chatRoom = new ChatRoom();
User alice = new ConcreteUser(chatRoom, "Alice");
User bob = new ConcreteUser(chatRoom, "Bob");

chatRoom.addUser(alice);
chatRoom.addUser(bob);

alice.sendMessage("Hi, Bob!");
bob.sendMessage("Hello, Alice!");
```

Memento Pattern

The Memento pattern captures and externalizes an object's internal state so that the object can be restored to this state later. It's useful when you need to implement undo or rollback functionality.

Example: A text editor with undo functionality.

```
public class TextEditor {
    private String text = "";
    private String lastSavedText = "";

    public void write(String newText) {
        text += newText;
    }

    public void save() {
        lastSavedText = text;
    }
```

```
    public void undo() {
        text = lastSavedText;
    }

    public String read() {
        return text;
    }
}
```

Usage :

```
TextEditor textEditor = new TextEditor();
textEditor.write("Hello, world!");
textEditor.save();

textEditor.write(" And this is a new line.");
System.out.println(textEditor.read());

textEditor.undo();
System.out.println(textEditor.read());
```

Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It's useful when you want to maintain consistency between related objects.

Example: A weather station with multiple display elements.

```
public interface Observer {
    void update(float temperature, float humidity, float pressure
}

public interface Subject {
    void registerObserver(Observer observer);
```

```
    void removeObserver(Observer observer);
    void notifyObservers();
}

public interface DisplayElement {
    void display();
}

public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void setMeasurements(float temperature, float humidity
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
```

```
    }
}

public class CurrentConditionsDisplay implements Observer, Displa
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float p
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    @Override
    public void display() {
        System.out.println("Current conditions: " + temperature +
    }
}

public class ForecastDisplay implements Observer, DisplayElement
    private float currentPressure = 29.92f;
    private float lastPressure;
    private Subject weatherData;

    public ForecastDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float p
        lastPressure = currentPressure;
        currentPressure = pressure;
```

```
        display();
    }

    @Override
    public void display() {
        System.out.print("Forecast: ");
        if (currentPressure > lastPressure) {
            System.out.println("Improving weather on the way!");
        } else if (currentPressure == lastPressure) {
            System.out.println("More of the same");
        } else if (currentPressure < lastPressure) {
            System.out.println("Watch out for cooler, rainy weath
        }
    }
}
```

Usage :

```
WeatherData weatherData = new WeatherData();
CurrentConditionsDisplay currentConditionsDisplay = new CurrentCo
ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData

weatherData.setMeasurements(25.5f, 65f, 30.4f);
weatherData.setMeasurements(22.2f, 70f, 29.2f);
```

State Pattern

The State pattern allows an object to alter its behavior when its internal state changes. The object appears to change its class. It's useful when you have an object with a large number of conditional statements based on its state.

Example: A gumball machine with different states.

```
public interface State {
    void insertCoin();
```

```
    void ejectCoin();
    void turnCrank();
    void dispense();
}

public class GumballMachine {
    State soldOutState;
    State noCoinState;
    State hasCoinState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int count) {
        soldOutState = new SoldOutState(this);
        noCoinState = new NoCoinState(this);
        hasCoinState = new HasCoinState(this);
        soldState = new SoldState(this);

        this.count = count;
        if (count > 0) {
            state = noCoinState;
        } else {
            state = soldOutState;
        }
    }

    public void insertCoin() {
        state.insertCoin();
    }

    public void ejectCoin() {
        state.ejectCoin();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }
```

```
    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out...");
        if (count != 0) {
            count--;
        }
    }

    int getCount() {
        return count;
    }

    State getSoldOutState() {
        return soldOutState;
    }

    State getNoCoinState() {
        return noCoinState;
    }

    State getHasCoinState() {
        return hasCoinState;
    }

    State getSoldState() {
        return soldState;
    }
}
```

The `State` implementations are not shown here due to length constraints. You can find the full example in the GitHub repository

Usage :

```
GumballMachine gumballMachine = new GumballMachine(5);

gumballMachine.insertCoin();
gumballMachine.turnCrank();

gumballMachine.insertCoin();
gumballMachine.ejectCoin();
gumballMachine.turnCrank();

gumballMachine.insertCoin();
gumballMachine.turnCrank();
gumballMachine.insertCoin();
gumballMachine.turnCrank();
gumballMachine.ejectCoin();

gumballMachine.insertCoin();
gumballMachine.insertCoin();
gumballMachine.turnCrank();
gumballMachine.insertCoin();
gumballMachine.turnCrank();
gumballMachine.insertCoin();
gumballMachine.turnCrank();
```

Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it. It's useful when you have multiple ways to solve a problem and want to choose an algorithm at runtime.

Example: A text editor with different sorting algorithms.

```
public interface SortingStrategy {
    void sort(List<String> list);
}

public class QuickSortStrategy implements SortingStrategy {
```

```
    @Override
    public void sort(List<String> list) {
        // Implement quick sort algorithm here
        System.out.println("List sorted using Quick Sort");
    }
}

public class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(List<String> list) {
        // Implement merge sort algorithm here
        System.out.println("List sorted using Merge Sort");
    }
}

public class TextEditor {
    private SortingStrategy strategy;

    public TextEditor(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(List<String> list) {
        strategy.sort(list);
    }
}
```

Usage:

```
List<String> text = Arrays.asList("apple", "orange", "banana", "g

TextEditor textEditor = new TextEditor(new QuickSortStrategy());
textEditor.sort(text);
```

```
textEditor.setStrategy(new MergeSortStrategy());
textEditor.sort(text);
```

Template Method Pattern

The Template Method pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. It's useful when you want to share common behavior between classes with different implementations.

Example: A data exporter with different export formats.

```
public abstract class DataExporter {
    public final void exportData() {
        fetchData();
        formatData();
        writeDataToFile();
    }

    protected abstract void fetchData();
    protected abstract void formatData();

    private void writeDataToFile() {
        // Write data to a file
        System.out.println("Data written to file");
    }
}

public class CSVDataExporter extends DataExporter {
    @Override
    protected void fetchData() {
        System.out.println("Fetching data for CSV export");
    }

    @Override
    protected void formatData() {
        System.out.println("Formatting data as CSV");
```

```
    }
}

public class XMLDataExporter extends DataExporter {
    @Override
    protected void fetchData() {
        System.out.println("Fetching data for XML export");
    }

    @Override
    protected void formatData() {
        System.out.println("Formatting data as XML");
    }
}
```

Usage :

```
DataExporter csvExporter = new CSVDataExporter();
csvExporter.exportData();

DataExporter xmlExporter = new XMLDataExporter();
xmlExporter.exportData();
```

Conclusion

Design patterns are an essential tool for every Java programmer. They provide reusable solutions to common problems, allowing you to write cleaner, more maintainable code. By studying and implementing these patterns in your projects, you can improve your skills as a developer and make your software more adaptable to change. Don't forget to practice implementing these patterns in your projects to get a better grasp of their usage.

Remember that design patterns are not a one-size-fits-all solution, and it's essential to understand the problem you're trying to solve and choose the appropriate design pattern for the situation. The examples provided in this blog post are just the starting point for understanding the patterns. To gain a deeper

knowledge of design patterns, consider studying additional resources and examples to expand your understanding and mastery of these powerful tools.

More from this blog

Spring Boot (Imperative) vs Spring WebFlux (Reactive)

1. Introduction Spring Boot and Spring WebFlux are two approaches to building applications using the Spring framework. Spring Boot follows a traditional imperative (blocking) programming model, while Spring WebFlux is reactive (non-blocking) and desi...

Apr 4, 2025 4 min read

A Comprehensive Guide to Reflection in Java for Developers

Exploring Java Reflection for Dynamic Code Execution

Apr 22, 2023 10 min read

Unleashing the Power of Annotations in Java: A Comprehensive Guide

Discover the power of Java annotations, learn how to create and process custom annotations, and enhance your projects with this essential feature.

Apr 20, 2023 6 min read

Mastering Generics in Java: A Comprehensive Guide for Java Developers

Delve into the world of Java generics to write robust, type-safe, and reusable code.

Apr 17, 2023 7 min read

Mastering Multithreading in Java for Enhanced Performance

A comprehensive guide to advanced multithreading in Java, with practical code examples for optimal performance

Apr 16, 2023 11 min read

Embracing Tech Evolution

12 posts published