

Be part of a better internet. [Get 20% off membership for a limited time](#)

# Understanding Behavioral Design Patterns in Java: Enhancing Object Interaction



Rizwanul Haque · [Follow](#)

Published in Dev Genius · 18 min read · May 2, 2024

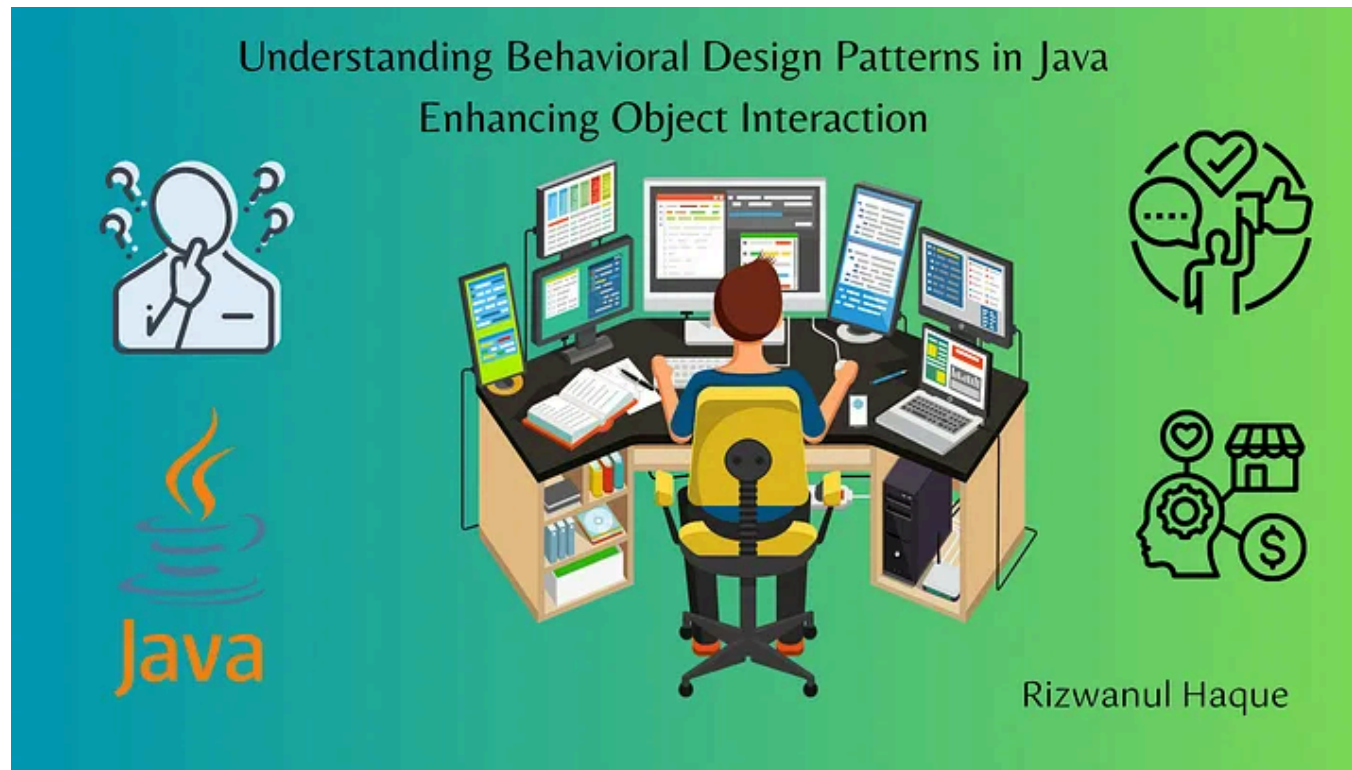


663



3





Behavioral design patterns in Java focus on defining the communication and interaction between objects and classes. These patterns provide solutions for managing algorithms, responsibilities, and collaborations among objects, ensuring flexible and efficient communication within a software system. In this article, we'll explore some common behavioral design patterns in Java along with their implementations and real-world use cases.

## Types of Behavioral Design Patterns

Behavioral design patterns in Java can be classified into several categories, each addressing specific aspects of object interaction:

1. **Chain of Responsibility Pattern:** Decouples the sender of a request from its receivers, allowing multiple objects to handle the request sequentially.
2. **Command Pattern:** Encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.
3. **Interpreter Pattern:** Defines a grammar for interpreting language expressions, providing a way to evaluate sentences or expressions.
4. **Iterator Pattern:** Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.
5. **Mediator Pattern:** Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
6. **Memento Pattern:** Captures and externalizes an object's internal state, allowing the object to be restored to this state later.
7. **Observer Pattern:** Defines a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically.

8. **State Pattern:** Allows an object to alter its behavior when its internal state changes, encapsulating state-specific logic into separate objects.
9. **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to choose algorithms at runtime.
10. **Template Method Pattern:** Defines the skeleton of an algorithm in a method, deferring some steps to subclasses, allowing subclasses to redefine certain steps of the algorithm without changing its structure.
11. **Visitor Pattern:** Represents an operation to be performed on elements of an object structure, separating the operation from the object structure.

## **1. Chain of Responsibility Pattern:**

The Chain of Responsibility pattern in Java is a behavioral design pattern that allows multiple objects to handle a request without the sender needing to know which object will handle it. Each handler in the chain has the ability to either handle the request or pass it on to the next handler in the chain. This pattern promotes loose coupling and flexibility in the system, as the sender and receivers are decoupled from each other.

Let's dive into the implementation of the Chain of Responsibility pattern in Java with a simple example:

## 1. Define the Handler Interface

```
public interface RequestHandler {  
    void handleRequest(Request request);  
    void setNextHandler(RequestHandler nextHandler);  
}
```

## 2. Implement Concrete Handlers

```
public class Type1Handler implements RequestHandler {  
    private RequestHandler nextHandler;  
  
    public void handleRequest(Request request) {  
        if (request.getType().equals("Type1")) {  
            System.out.println("Request handled by Type1Handler");  
        } else if (nextHandler != null) {  
            nextHandler.handleRequest(request);  
        } else {  
            System.out.println("Request cannot be handled");  
        }  
    }  
  
    public void setNextHandler(RequestHandler nextHandler) {
```

```
        this.nextHandler = nextHandler;
    }
}

public class Type2Handler implements RequestHandler {
    private RequestHandler nextHandler;

    public void handleRequest(Request request) {
        if (request.getType().equals("Type2")) {
            System.out.println("Request handled by Type2Handler");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("Request cannot be handled");
        }
    }

    public void setNextHandler(RequestHandler nextHandler) {
        this.nextHandler = nextHandler;
    }
}
```

### 3. Define the Request Class

```
public class Request {
    private String type;

    public Request(String type) {
        this.type = type;
    }
}
```

```
public String getType() {  
    return type;  
}  
}
```

## 4. Create and Configure the Chain of Handlers

```
public class ChainOfResponsibilityDemo {  
    public static void main(String[] args) {  
        RequestHandler handler1 = new Type1Handler();  
        RequestHandler handler2 = new Type2Handler();  
  
        handler1.setNextHandler(handler2);  
  
        // Send requests to the chain  
        handler1.handleRequest(new Request("Type1")); // Output: Request handled  
        handler1.handleRequest(new Request("Type2")); // Output: Request handled  
        handler1.handleRequest(new Request("Type3")); // Output: Request cannot  
    }  
}
```

### Real-world Use Case: Logging System

A common real-world use case for the Chain of Responsibility pattern is a logging system, where different loggers handle log messages based on their severity levels. For example, a logger may handle INFO-level messages, while

another logger may handle WARNING-level messages. If a logger cannot handle a particular message, it passes it on to the next logger in the chain until the message is handled or reaches the end of the chain.

The Chain of Responsibility pattern in Java provides a flexible and scalable way to handle requests in a system by decoupling senders from receivers. By allowing multiple handlers to process requests, this pattern promotes code reusability and flexibility, making it an invaluable tool for building robust and maintainable software systems. Whether you're building logging systems, event handling mechanisms, or authentication systems, the Chain of Responsibility pattern offers a clean and elegant solution to manage request processing in Java applications.

## **2. Command Pattern:**

The Command Pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations. This pattern decouples the sender of a request from its receiver, providing flexibility and extensibility in managing commands and actions within a software system. In this article, we'll explore the Command Pattern in Java, including its implementation with a code example and a real-world use case.



## Overview of the Command Pattern

The Command Pattern consists of four main components:

1. **Command:** Defines an interface for executing an operation.
2. **ConcreteCommand:** Implements the Command interface and encapsulates the receiver object and the action to be performed.
3. **Invoker:** Asks the command to carry out the request.
4. **Receiver:** Performs the actual action associated with the command.

## Implementation of Command Pattern in Java

Let's implement the Command Pattern with a simple example:

### 1. Define the Command Interface

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

## 2. Implement Concrete Commands

```
// Concrete command to turn on the light
public class TurnOnLightCommand implements Command {
    private Light light;

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

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

// Concrete command to turn off the light
public class TurnOffLightCommand implements Command {
    private Light light;

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

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

### 3. Implement the Receiver

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

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

### 4. Implement the Invoker

```
// Invoker class
public class RemoteControl {
    private Command command;

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

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

```
    }  
}
```

## 5. Client Code

```
// Client code  
public class CommandPatternDemo {  
    public static void main(String[] args) {  
        // Create the receiver  
        Light light = new Light();  
  
        // Create the commands  
        Command turnOnCommand = new TurnOnLightCommand(light);  
        Command turnOffCommand = new TurnOffLightCommand(light);  
  
        // Create the invoker  
        RemoteControl remoteControl = new RemoteControl();  
  
        // Set and execute the commands  
        remoteControl.setCommand(turnOnCommand);  
        remoteControl.pressButton(); // Output: Light is on  
  
        remoteControl.setCommand(turnOffCommand);  
        remoteControl.pressButton(); // Output: Light is off  
    }  
}
```

## **Real-world Use Case: Remote Control Device**

A real-world use case for the Command Pattern is a remote control device that interacts with various electronic appliances such as lights, fans, or televisions. Each button on the remote control represents a command, and pressing a button executes the corresponding command, such as turning a light on or off, adjusting the fan speed, or changing the channel on a television. By encapsulating each command as an object, the Command Pattern provides a flexible and extensible solution for managing device interactions.

## **3. Interpreter Pattern:**

The Interpreter Pattern is a behavioral design pattern that defines a grammar for interpreting language expressions. It provides a way to evaluate sentences or expressions represented in a language, facilitating the interpretation and execution of complex rules or commands. In this article, we'll explore the Interpreter Pattern in Java, its implementation, and a real-world use case.

### **Understanding the Interpreter Pattern**

The Interpreter Pattern involves defining a grammar for a language and providing an interpreter to interpret and execute expressions in that

language. It consists of several key components:

1. **Abstract Expression:** Defines an abstract interface for interpreting expressions.
2. **Terminal Expression:** Implements the abstract expression interface for terminal symbols in the grammar.
3. **Non-Terminal Expression:** Implements the abstract expression interface for non-terminal symbols in the grammar.
4. **Context:** Contains information that is global to the interpreter and is shared across expressions.
5. **Client:** Builds and evaluates expressions using the interpreter.

## Implementation of the Interpreter Pattern in Java

Let's implement a simple example of the Interpreter Pattern to interpret and evaluate arithmetic expressions:

### 1. Define the Abstract Expression Interface

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

}

## 2. Implement Terminal and Non-Terminal Expressions

```
public class NumberExpression implements Expression {
    private int number;

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

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

public class AdditionExpression implements Expression {
    private Expression leftOperand;
    private Expression rightOperand;

    public AdditionExpression(Expression leftOperand, Expression rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    @Override
    public int interpret(Context context) {
        return leftOperand.interpret(context) + rightOperand.interpret(context);
    }
}
```

```
}  
}
```

### 3. Define the Context Class

```
public class Context {  
    // Optional: Include context information relevant to the interpreter.  
}
```

### 4. Build and Evaluate Expressions using the Interpreter

```
public class InterpreterDemo {  
    public static void main(String[] args) {  
        // Define context (if needed)  
        Context context = new Context();  
  
        // Build expression tree  
        Expression expression = new AdditionExpression(  
            new NumberExpression(10),  
            new AdditionExpression(  
                new NumberExpression(5),  
                new NumberExpression(3)  
            )  
        );  
    }  
}
```



```
);  
  
// Evaluate expression  
int result = expression.interpret(context);  
System.out.println("Result: " + result); // Output: Result: 18  
}  
}
```

## Real-world Use Case: SQL Query Parser

One real-world use case of the Interpreter Pattern is in the implementation of a SQL query parser. The parser interprets SQL queries and translates them into executable code or database operations. Each part of the query, such as SELECT, FROM, WHERE, etc., is represented as a terminal or non-terminal expression in the grammar. By using the Interpreter Pattern, complex SQL queries can be parsed and executed efficiently, enabling database interactions in Java applications.

The Interpreter Pattern in Java provides a powerful mechanism for interpreting and evaluating language expressions. By defining a grammar and providing an interpreter, developers can handle complex rules and commands in a flexible and extensible manner. Whether it's parsing arithmetic expressions or interpreting SQL queries, the Interpreter Pattern

finds application in various domains, making it a valuable tool in software development.

## **4. Memento Pattern:**

The Memento Pattern is a behavioral design pattern that allows the capture and externalization of an object's internal state without violating encapsulation, enabling the object to be restored to this state later. In Java, the Memento Pattern is particularly useful in scenarios where you need to implement undo mechanisms, maintain checkpoints, or manage the history of an object's states. Let's delve into the details of the Memento Pattern with a code example and a real-world use case.

### **Understanding the Memento Pattern**

The Memento Pattern consists of three main components:

1. **Originator:** The object whose state needs to be saved and restored.
2. **Memento:** Represents the saved state of the originator object.
3. **CareTaker:** Manages the mementos, but doesn't modify them.

### **Implementation of the Memento Pattern in Java**

Let's implement the Memento Pattern with a simple example of a text editor that allows users to undo text changes:

## 1. Define the Memento Class

```
// Memento class represents the saved state of the text editor.  
class TextMemento {  
    private String text;  
  
    public TextMemento(String text) {  
        this.text = text;  
    }  
  
    public String getText() {  
        return text;  
    }  
}
```

## 2. Implement the Originator Class

```
// Originator class represents the text editor.  
class TextEditor {  
    private String text;
```

```
public void setText(String text) {
    this.text = text;
}

public String getText() {
    return text;
}

public TextMemento save() {
    return new TextMemento(text);
}

public void restore(TextMemento memento) {
    this.text = memento.getText();
}
}
```

### 3. Create the CareTaker Class

```
// CareTaker class manages the mementos.
import java.util.Stack;

class History {
    private Stack<TextMemento> mementos = new Stack<>();

    public void push(TextMemento memento) {
        mementos.push(memento);
    }

    public TextMemento pop() {
```

```
        return mementos.pop();  
    }  
}
```

## 4. Using the Memento Pattern

```
public class MementoPatternDemo {  
    public static void main(String[] args) {  
        TextEditor editor = new TextEditor();  
        History history = new History();  
  
        // Editing text  
        editor.setText("First draft");  
        history.push(editor.save());  
  
        // Editing text again  
        editor.setText("Second draft");  
        history.push(editor.save());  
  
        // Undoing changes  
        editor.restore(history.pop());  
        System.out.println("Text after undo: " + editor.getText());  
    }  
}
```

## Real-world Use Case: Text Editor Undo Mechanism

In a text editor application, the Memento Pattern can be used to implement the undo mechanism. Each time the user makes a change to the text, a memento representing the current state of the text is created and stored in a history. When the user requests to undo a change, the text editor restores the text to its previous state using the memento from the history.

The Memento Pattern provides an elegant solution for capturing and restoring object states in Java applications. By encapsulating the object's state in a memento and delegating the responsibility of managing mementos to a caretaker, this pattern promotes encapsulation and separation of concerns. Whether you're implementing undo mechanisms, managing checkpoints, or maintaining the history of an object's states, the Memento Pattern offers a flexible and efficient solution for managing object states in Java applications.

## **5. Observer Pattern:**

The Observer pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes, usually by calling one of their methods. This pattern facilitates a one-to-many relationship between objects, ensuring that when the state of the subject changes, all its observers are notified and

updated automatically. Let's delve into the implementation of the Observer pattern in Java along with a real-world use case.

## Implementation of Observer Pattern in Java

### 1. Define the Subject Interface

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

// Subject interface
interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}
```

### 2. Define the Subject Interface

```
// Concrete subject
class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;

    public int getState() {
```

```
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyObservers();
    }

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

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

### 3. Define the Observer Interface

```
// Observer interface
interface Observer {
```



```
void update();  
}
```

## 4. Implement the Concrete Observer

```
// Concrete observer  
class ConcreteObserver implements Observer {  
    private ConcreteSubject subject;  
  
    public ConcreteObserver(ConcreteSubject subject) {  
        this.subject = subject;  
        this.subject.attach(this);  
    }  
  
    public void update() {  
        System.out.println("Observer updated with state: " + subject.getState())  
    }  
}
```

## 5. Client Code

```
// Client code  
public class ObserverPatternDemo {
```

```
public static void main(String[] args) {  
    ConcreteSubject subject = new ConcreteSubject();  
    ConcreteObserver observer1 = new ConcreteObserver(subject);  
    ConcreteObserver observer2 = new ConcreteObserver(subject);  
  
    subject.setState(5); // Output: Observer updated with state: 5  
}  
}
```

## Real-world Use Case: Weather Monitoring System

A common real-world use case for the Observer pattern is in a weather monitoring system. In this scenario, multiple weather stations across different locations act as subjects, constantly measuring weather conditions such as temperature, humidity, and pressure. Weather displays, weather apps, and other devices act as observers, subscribing to updates from these weather stations. Whenever the weather conditions change at a particular station, all its observers are notified, and they update their displays accordingly.

The Observer pattern in Java provides an elegant solution for implementing communication and interaction between objects in a loosely coupled manner. By allowing multiple observers to subscribe to changes in the state of a subject, this pattern facilitates efficient event handling and synchronization within a software system. Whether you're building weather

monitoring systems, stock market trackers, or user interface components, the Observer pattern proves to be a valuable tool for maintaining consistency and responsiveness in Java applications.

## 6. State Pattern:

The State design pattern is a behavioral pattern that allows an object to alter its behavior when its internal state changes. This pattern encapsulates state-specific behavior into separate state objects and allows the object to change its state dynamically. In this article, we'll delve into the State pattern in Java, providing a code example and discussing a real-world use case.

### Understanding the State Pattern

The State pattern is based on the principle of encapsulation, where each state of an object is represented by a separate class. The context object maintains a reference to the current state object and delegates state-specific behavior to it. When the context's state changes, it switches to a different state object, thus altering its behavior dynamically.

### State Pattern Components

The State pattern consists of the following components:

1. **Context:** Represents the object whose behavior varies based on its internal state. It maintains a reference to the current state object.
2. **State:** Defines an interface for encapsulating the behavior associated with a particular state of the context object.
3. **Concrete State:** Implements the behavior associated with a specific state of the context object.

## State Pattern Implementation in Java

Let's implement the State pattern with a simple example of a traffic light system:

```
// State interface
interface TrafficLightState {
    void handleState(TrafficLight context);
}

// Concrete states
class RedLightState implements TrafficLightState {
    public void handleState(TrafficLight context) {
        System.out.println("Red Light: Stop");
        // Transition to the next state
        context.setState(new GreenLightState());
    }
}

class GreenLightState implements TrafficLightState {
```

```
        public void handleState(TrafficLight context) {
            System.out.println("Green Light: Go");
            // Transition to the next state
            context.setState(new YellowLightState());
        }
    }

    class YellowLightState implements TrafficLightState {
        public void handleState(TrafficLight context) {
            System.out.println("Yellow Light: Prepare to Stop");
            // Transition to the next state
            context.setState(new RedLightState());
        }
    }

    // Context
    class TrafficLight {
        private TrafficLightState currentState;

        public TrafficLight() {
            // Initial state
            currentState = new RedLightState();
        }

        public void setState(TrafficLightState state) {
            currentState = state;
        }

        public void changeState() {
            currentState.handleState(this);
        }
    }

    // Client code
    public class StatePatternDemo {
        public static void main(String[] args) {
```

```
TrafficLight trafficLight = new TrafficLight();

// Simulate traffic light changes
trafficLight.changeState(); // Output: Red Light: Stop
trafficLight.changeState(); // Output: Green Light: Go
trafficLight.changeState(); // Output: Yellow Light: Prepare to Stop
trafficLight.changeState(); // Output: Red Light: Stop
    }
}
```

## Real-world Use Case: Traffic Light System

A common real-world use case for the State pattern is a traffic light system. The behavior of a traffic light varies based on its current state (red, green, or yellow). By implementing the State pattern, we can encapsulate the behavior associated with each state into separate state objects. This allows the traffic light to change its behavior dynamically based on its internal state, ensuring safe and efficient traffic flow on the roads.

The State pattern in Java provides a flexible and maintainable solution for managing object behavior that varies based on its internal state. By encapsulating state-specific behavior into separate state objects, the State pattern promotes code reuse, modularity, and scalability. Whether you're simulating traffic light systems, implementing vending machines, or

modeling game characters, the State pattern offers a powerful way to manage object behavior in Java applications.

## 7. Strategy Pattern:

The Strategy Pattern is a behavioral design pattern that enables the definition of a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from clients that use it, providing flexibility and promoting code reuse. In this article, we'll explore the Strategy Pattern in Java, including its implementation with a code example and a real-world use case.

### Understanding the Strategy Pattern

The Strategy Pattern involves three key components:

1. **Context:** Represents the client that uses the strategy. It maintains a reference to a Strategy object and delegates the algorithm's execution to it.
2. **Strategy:** Defines an interface or an abstract class for the algorithm family. Concrete implementations represent specific algorithms.
3. **Concrete Strategy:** Implements the algorithm defined by the Strategy interface. Multiple concrete strategies can exist within the same family of

algorithms.

## Implementation of Strategy Pattern in Java

Let's implement the Strategy Pattern with a code example:

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

// Concrete strategies
class CreditCardPaymentStrategy implements PaymentStrategy {
    private String cardNumber;
    private String expiryDate;
    private String cvv;

    public CreditCardPaymentStrategy(String cardNumber, String expiryDate, String cvv) {
        this.cardNumber = cardNumber;
        this.expiryDate = expiryDate;
        this.cvv = cvv;
    }

    public void pay(double amount) {
        // Implement credit card payment logic
        System.out.println("Paid " + amount + " via credit card.");
    }
}

class PayPalPaymentStrategy implements PaymentStrategy {
    private String email;
```



```
private String password;

public PayPalPaymentStrategy(String email, String password) {
    this.email = email;
    this.password = password;
}

public void pay(double amount) {
    // Implement PayPal payment logic
    System.out.println("Paid " + amount + " via PayPal.");
}
}

// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

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

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

// Client code
public class StrategyPatternDemo {
    public static void main(String[] args) {
        // Create a shopping cart
        ShoppingCart cart = new ShoppingCart();

        // Set payment strategy to credit card
        cart.setPaymentStrategy(new CreditCardPaymentStrategy("1234567890123456")

        // Checkout using the selected payment strategy
    }
}
```

```
cart.checkout(100.00);

// Change payment strategy to PayPal
cart.setPaymentStrategy(new PayPalPaymentStrategy("example@example.com",

// Checkout using the new payment strategy
cart.checkout(50.00);
    }
}
```

## Real-world Use Case: Payment Processing System

A common real-world use case for the Strategy Pattern is in payment processing systems. Different payment methods, such as credit cards, PayPal, and cryptocurrency, require varying algorithms for payment processing. By applying the Strategy Pattern, the payment processing system can support multiple payment methods with interchangeable algorithms. For instance:

- **Credit Card Payment Strategy:** Implements the algorithm for credit card payments, including validation and transaction processing.
- **PayPal Payment Strategy:** Implements the algorithm for PayPal payments, including authentication and fund transfer.

The Strategy Pattern in Java provides a flexible and elegant solution for managing a family of algorithms and making them interchangeable. By separating the algorithm from the client and encapsulating each algorithm in its own class, the Strategy Pattern promotes code reuse, maintainability, and extensibility. Whether you're building payment processing systems, sorting algorithms, or any other application that requires interchangeable algorithms, the Strategy Pattern offers a powerful and versatile approach in Java development.

## 8. Visitor Pattern:

The Visitor Pattern is a behavioral design pattern that allows adding new behaviors to existing object structures without modifying those structures. It achieves this by separating the algorithm from the object structure it operates on. In this article, we'll explore the Visitor Pattern in Java, including its implementation and real-world use cases.

### Visitor Pattern Overview

The Visitor Pattern consists of the following key components:

- **Visitor:** Defines a new operation to be performed on elements of an object structure. It declares a `visit` method for each class of element in

the object structure.

- **ConcreteVisitor:** Implements the `visit` methods defined in the Visitor interface. Each `visit` method implements a specific operation on an element of the object structure.
- **Element:** Defines an interface for accepting visitor objects and declares an `accept` method that takes a Visitor as an argument.
- **ConcreteElement:** Implements the `accept` method defined in the Element interface. It allows the Visitor to visit and operate on the element.
- **Object Structure:** Represents a collection or structure of elements that accept visitor objects. It provides a way to iterate over its elements and accept visitor objects.

## Implementation of Visitor Pattern in Java

Let's implement the Visitor Pattern in Java with a simple example:

### 1. Define the Visitor and Element Interfaces

```
// Visitor interface
interface Visitor {
    void visit(ConcreteElementA element);
    void visit(ConcreteElementB element);
}

// Element interface
interface Element {
    void accept(Visitor visitor);
}
```

## 2. Implement Concrete Visitor and Concrete Element Classes

```
// ConcreteVisitor class
class ConcreteVisitor implements Visitor {
    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("Visitor is visiting ConcreteElementA");
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("Visitor is visiting ConcreteElementB");
    }
}

// ConcreteElement classes
class ConcreteElementA implements Element {
```

```
@Override
public void accept(Visitor visitor) {
    visitor.visit(this);
}

}

class ConcreteElementB implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

### 3. Implement the Object Structure

```
// Object Structure class
class ObjectStructure {
    private List<Element> elements = new ArrayList<>();

    public void addElement(Element element) {
        elements.add(element);
    }

    public void accept(Visitor visitor) {
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

```
}  
}
```

## 4. Client Code

```
public class VisitorPatternDemo {  
    public static void main(String[] args) {  
        ObjectStructure objectStructure = new ObjectStructure();  
        objectStructure.addElement(new ConcreteElementA());  
        objectStructure.addElement(new ConcreteElementB());  
  
        Visitor visitor = new ConcreteVisitor();  
        objectStructure.accept(visitor);  
    }  
}
```

### Real-world Use Case: Document Processing

A real-world use case of the Visitor Pattern is document processing.

Consider a document processing system where different types of documents (e.g., text documents, spreadsheets, presentations) need to be processed.

Each document type may have its own structure and elements.

By implementing the Visitor Pattern, you can define a visitor interface with methods to process different types of document elements. Concrete visitor classes can then be created to perform specific operations on each type of document element. This approach allows for adding new document processing operations without modifying the existing document classes, promoting maintainability and extensibility.

The Visitor Pattern in Java provides a flexible and elegant solution for adding new behaviors to existing object structures. By separating the algorithm from the object structure, it enables easy extension and modification of behavior without modifying the structure itself. Whether you're processing documents, traversing complex data structures, or implementing other domain-specific operations, the Visitor Pattern offers a powerful and versatile tool for enhancing object behavior in Java applications.

## **9. Template Method Pattern:**

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring certain steps to subclasses. This pattern allows subclasses to redefine certain steps of the algorithm without changing its structure. In this article, we'll delve into the Template



Method Pattern in Java, providing a code example and exploring a real-world use case.

## Understanding the Template Method Pattern

The Template Method Pattern consists of two main components:

1. **Abstract Class:** Defines the skeleton of the algorithm as a series of steps in a method, with some steps implemented and others left abstract.
2. **Concrete Subclasses:** Implement the abstract methods to provide concrete implementations for specific steps of the algorithm.

The Template Method Pattern promotes code reuse and ensures consistency in algorithm execution while allowing flexibility for subclasses to customize certain steps.

## Implementing the Template Method Pattern

Let's implement a simple example of a template method pattern in Java for preparing beverages:

```
// Abstract class defining the template method
abstract class Beverage {
```

```
// Template method defining the algorithm
public final void prepareBeverage() {
    boilWater();
    brew();
    pourInCup();
    if (addCondiments()) {
        addCondiments();
    }
}

// Abstract methods to be implemented by subclasses
abstract void brew();
abstract void addCondiments();

// Concrete methods
void boilWater() {
    System.out.println("Boiling water");
}

void pourInCup() {
    System.out.println("Pouring into cup");
}
}

// Concrete subclass implementing specific beverage - Coffee
class Coffee extends Beverage {
    @Override
    void brew() {
        System.out.println("Brewing coffee");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}
```

```
// Concrete subclass implementing specific beverage - Tea
class Tea extends Beverage {
    @Override
    void brew() {
        System.out.println("Steeping tea");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}

// Client code
public class TemplateMethodPatternDemo {
    public static void main(String[] args) {
        Beverage coffee = new Coffee();
        coffee.prepareBeverage();

        Beverage tea = new Tea();
        tea.prepareBeverage();
    }
}
```

## Real-world Use Case: Web Application Frameworks

Web application frameworks often use the Template Method Pattern to define the structure of request processing. For example, in Java Servlets, the `doGet()` and `doPost()` methods are template methods that define the sequence of steps for handling HTTP GET and POST requests. Subclasses,

such as servlets, can then override these methods to implement specific request handling logic while leveraging the common structure provided by the framework.

The Template Method Pattern in Java provides a flexible and reusable way to define the structure of algorithms while allowing subclasses to customize certain steps. By encapsulating the common algorithm structure in a template method, this pattern promotes code reuse, consistency, and maintainability. Whether you're preparing beverages or processing HTTP requests in web applications, the Template Method Pattern offers a powerful tool for streamlining algorithm structure in Java projects.

[Design Patterns](#)[Behavioural Design](#)[Java](#)[Software Development](#)



## Written by Rizwanul Haque

1.1K Followers · Writer for Dev Genius

Follow

Lead Android Developer | Passionate about Coding | Sharing Insights 🚀 | Let's explore together! 📱💻 #AndroidDev #Kotlin #Tech

### More from Rizwanul Haque and Dev Genius



 Rizwanul Haque in Stackademic

### Important Coroutine Interview Questions for Experienced Andro...



Most Asked  
Java 8  
Interview  
Coding Questions  
(Part-1)

 Anusha SP in Dev Genius

### Java 8 Coding and Programming Interview Questions and Answers

Coroutines have revolutionized asynchronous programming in Kotlin, offering a powerful...

Feb 13



1K



3



It has been 8 years since Java 8 was released. I have already shared the Java 8 Interview...

Jan 31, 2023




1K



15



 Amit Mishra in Dev Genius

## 45 JavaScript Super Hacks Every Developer Should Know

JavaScript is a versatile and powerful language that is essential for modern web...



 Rizwanul Haque

## SSL Pinning in Android Apps for Enhanced Security

Introduction:

# Medium



Search



Write



See all from Rizwanul Haque

See all from Dev Genius

## Recommended from Medium



Renan Schmitt in JavaJams

### Spring Boot: Handling a REST Endpoint That Queries More Data...

If you are a developer and have not faced this issue yet, it is still worth reading, as at some...



Jun 24



176



3



Oliver Foster

### JAVA: What Happens When a Thread Calls Start Twice?

My article is open to everyone; non-member readers can click this link to read the full text.



Jun 26



127



1



## Lists



General Coding Knowledge

20 stories · 1360 saves



Stories to Help You Grow as a Software Developer

19 stories · 1184 saves



Coding & Development

11 stories · 690 saves



Good Product Thinking

11 stories · 622 saves



Ashay & Shreyansh

3 Reasons Why You Fail in Interviews? (Coding and System...

Failing Despite Preparation



5d ago



102



Priya Upmaka

Beginners guide to System Design

BASICS



May 22

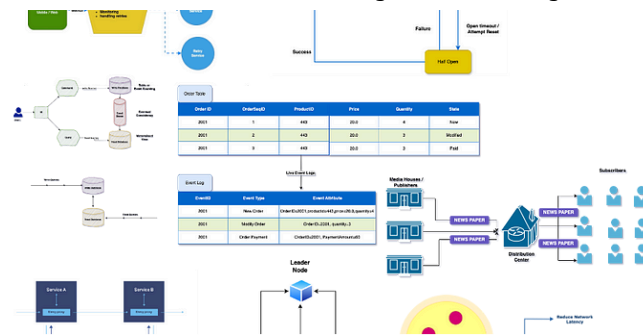



60



Network & Communication Protocols (IP, DNS, HTTP, TCP, UDP)	RDBMS and NoSQL DBs (MySQL, PostgreSQL, MongoDB, Cassandra)	Servers (EC2, Lambda)	Messaging System (Point to point, Pub-Sub)	Handshaking, Security protocols (TLS, SSL, HTTPS), Encryption	Creational, Structural & Behavioral	Solid Principle
Storage, Latency & Throughput	ACID properties	Availability & Scaling (horizontal, vertical)	Message serialization	JWT ( token handling)	Singleton	12 Factor App
Polling, Streaming, Sockets	Indexes	Load Balancing	Messaging Queues (RabbitMQ, Service Bus, Kafka)	OAuth	Factory	Microservices & Autopattern
	Joins	API Gateway	API architecture	Saving username & pwd	Event Bus Pattern	Various system design approach
	Data Partitioning/ Sharding	Proxies	REST, SOAP	IDP, SP and SSO	Microservices Patterns	
	Data Replication Strategies	Server Selection Strategies	Session between Microservices	Logging, monitoring, and ELK	Event Driven CQRS Pattern	
	CAP Theorem	Consistent Hashing	Communication between Microservices (3 ways to do it)	OWASP framework	Observer Pattern	
	Eventual Consistency	VMs and Docker/Containers	Restrict microservices calling each other?			
	GraphQL	Kubernetes	Circuit breaker, failure recovery			
	Caching (Redis, Map-reduce, LRU Cache)	Leader Election	gRPC			
	Distributed Transactions	Consensus				





 Anil Gudigar in Javarevisited

## Most-Used Distributed System Design Patterns

Distributed system design patterns provide architects and developers with proven...

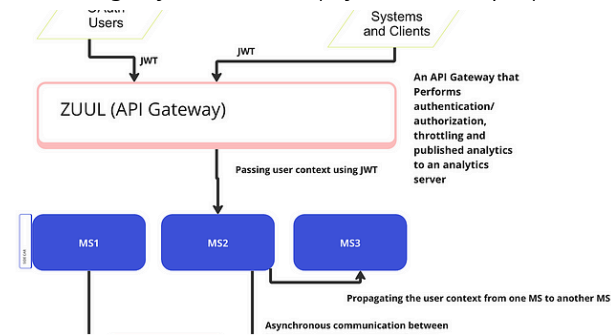
Jun 20  370  3



Jun 20  13



See more recommendations



 Vikas Taank

## Essential Interview Questions for RESTful Micro services

What are the differnt HTTP methods?