**DESIGN PATTERNS**

**Creational - Object creation**

**Singleton Pattern**

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.

**Key Features:**

1. **Single Instance**: Only one instance of the class is created.
2. **Global Access**: The instance is globally accessible.

**Steps to Implement:**

1. **Private Constructor**: Prevents instantiation from outside the class.
2. **Static Variable**: Holds the single instance of the class.
3. **Public Static Method**: Provides access to the instance.

**Example Code:**

java

Copy code

public class Singleton {

// Step 2: Static variable to hold the single instance

private static Singleton instance;

// Step 1: Private constructor

private Singleton() { }

// Step 3: Public method to provide access

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

**Variants:**

* **Eager Initialization**: The instance is created at class loading.
* **Lazy Initialization**: The instance is created when first requested (as in the example above).
* **Thread-Safe Singleton**: Use synchronization to handle multi-threaded environments.

**Thread-Safe Singleton Example:**

java

Copy code

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton() { }

public static synchronized ThreadSafeSingleton getInstance() {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

return instance;

}

}

**Advantages:**

* Saves memory by preventing multiple instances.
* Centralized control of the instance.

**Disadvantages:**

* Can make unit testing difficult due to global state.
* Potential performance issues in a multi-threaded environment without optimizations.

**Factory Method Pattern**

The Factory Method Pattern is a **creational design pattern** that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

**Key Features:**

1. **Encapsulation of Object Creation**: The logic for object creation is encapsulated within a method.
2. **Decoupling**: The client code relies on the interface, not on the concrete implementation of the object.
3. **Polymorphism**: Enables the use of polymorphism by allowing subclasses to specify the object to create.

**Real-time Example in Java**

Let's consider a real-world example of a **Document Creation System** that generates different types of documents such as Word and PDF. A factory method is used to create different types of documents, depending on the user's request.

Here’s how we can implement the Factory Method Pattern in Java:

**1. Abstract Class/Interface**

This defines the Factory Method.

java

Copy code

// Document.java - Abstract product

public abstract class Document {

public abstract void print();

}

**2. Concrete Products**

These are the specific types of documents that will be created.

java

Copy code

// WordDocument.java - Concrete product 1

public class WordDocument extends Document {

@Override

public void print() {

System.out.println("Printing a Word Document.");

}

}

// PdfDocument.java - Concrete product 2

public class PdfDocument extends Document {

@Override

public void print() {

System.out.println("Printing a PDF Document.");

}

}

**3. Creator Class**

This is the abstract class that declares the factory method. It can also provide a default implementation.

java

Copy code

// DocumentCreator.java - Creator (abstract class)

public abstract class DocumentCreator {

// The factory method

public abstract Document createDocument();

public void printDocument() {

Document document = createDocument();

document.print();

}

}

**4. Concrete Creators**

These are the classes that implement the factory method to instantiate the specific product.

java

Copy code

// WordDocumentCreator.java - Concrete creator 1

public class WordDocumentCreator extends DocumentCreator {

@Override

public Document createDocument() {

return new WordDocument(); // Factory method creates a WordDocument

}

}

// PdfDocumentCreator.java - Concrete creator 2

public class PdfDocumentCreator extends DocumentCreator {

@Override

public Document createDocument() {

return new PdfDocument(); // Factory method creates a PdfDocument

}

}

**5. Client Code**

The client code can use the creator class to get an object of the desired type without worrying about its instantiation.

java

Copy code

public class FactoryMethodExample {

public static void main(String[] args) {

DocumentCreator wordCreator = new WordDocumentCreator();

wordCreator.printDocument(); // Prints a Word Document

DocumentCreator pdfCreator = new PdfDocumentCreator();

pdfCreator.printDocument(); // Prints a PDF Document

}

}

**Output:**

css

Copy code

Printing a Word Document.

Printing a PDF Document.

**Explanation:**

1. **Document** is the abstract product class that declares the print method.
2. **WordDocument** and **PdfDocument** are concrete products implementing the print method.
3. **DocumentCreator** is the abstract creator class that declares the factory method createDocument. It also provides a printDocument method that delegates the creation and printing of the document.
4. **WordDocumentCreator** and **PdfDocumentCreator** are the concrete creators that override the factory method to create instances of their respective document types.
5. The client code (FactoryMethodExample) uses the creator classes to get instances of documents and prints them.

**Benefits of Factory Method Pattern:**

* **Encapsulation of Object Creation**: The client code doesn't need to know the specific class that needs to be instantiated.
* **Extensibility**: New types of documents (or products) can be added without changing the existing code.
* **Separation of Concerns**: The responsibility of object creation is separated from the client code.

This is a simple and effective use of the Factory Method Pattern in Java.

**Builder Pattern:**

**Definition:** The Builder pattern is a creational design pattern used to construct complex objects step by step. It separates the construction of an object from its representation, allowing the same construction process to create different representations.

**Key Components:**

1. **Product**: The complex object being built.
2. **Builder**: Defines the steps for constructing the object.
3. **ConcreteBuilder**: Implements the Builder interface and assembles the parts of the Product.
4. **Director**: Responsible for managing the construction process using the Builder object.
5. **Client**: Invokes the construction process to build the object.

**Purpose:**

* To separate the construction logic of an object from its representation.
* To construct different types of objects using the same construction process.

**Benefits:**

* **Separation of Concerns**: The construction process is separate from the object's structure.
* **Flexibility**: You can create different representations of the same type of object using the same builder.
* **Readability**: Helps in building complex objects in a step-by-step manner, improving code readability.

**When to Use:**

* When an object needs to be created with many possible configurations.
* When an object has several parameters, some of which are optional.
* To avoid telescoping constructors (multiple constructors with different parameters).

Let's consider a real-world example of creating a Computer object with multiple optional components like CPU, RAM, storage, etc. Using the Builder pattern, we can create different configurations of a computer without changing the main code.

**Step 1: Define the Computer Class (Product)**

java

Copy code

public class Computer {

private String CPU;

private String RAM;

private String storage;

private boolean isGraphicsCard;

private boolean isBluetooth;

public Computer(String CPU, String RAM, String storage, boolean isGraphicsCard, boolean isBluetooth) {

this.CPU = CPU;

this.RAM = RAM;

this.storage = storage;

this.isGraphicsCard = isGraphicsCard;

this.isBluetooth = isBluetooth;

}

public void displaySpecifications() {

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + storage);

System.out.println("Graphics Card: " + (isGraphicsCard ? "Yes" : "No"));

System.out.println("Bluetooth: " + (isBluetooth ? "Yes" : "No"));

}

}

**Step 2: Define the ComputerBuilder Interface**

java

Copy code

public interface ComputerBuilder {

void buildCPU();

void buildRAM();

void buildStorage();

void buildGraphicsCard();

void buildBluetooth();

Computer getComputer();

}

**Step 3: Concrete ComputerBuilder Implementation**

java

Copy code

public class GamingComputerBuilder implements ComputerBuilder {

private Computer computer;

public GamingComputerBuilder() {

computer = new Computer("", "", "", false, false);

}

@Override

public void buildCPU() {

computer = new Computer("Intel i9", "32GB", "1TB SSD", true, true);

}

@Override

public void buildRAM() {

// Already set in buildCPU method

}

@Override

public void buildStorage() {

// Already set in buildCPU method

}

@Override

public void buildGraphicsCard() {

// Already set in buildCPU method

}

@Override

public void buildBluetooth() {

// Already set in buildCPU method

}

@Override

public Computer getComputer() {

return computer;

}

}

**Step 4: Define the Director Class**

java

Copy code

public class Director {

private ComputerBuilder builder;

public Director(ComputerBuilder builder) {

this.builder = builder;

}

public Computer construct() {

builder.buildCPU();

builder.buildRAM();

builder.buildStorage();

builder.buildGraphicsCard();

builder.buildBluetooth();

return builder.getComputer();

}

}

**Step 5: Client Code**

java

Copy code

public class Client {

public static void main(String[] args) {

ComputerBuilder builder = new GamingComputerBuilder();

Director director = new Director(builder);

Computer computer = director.construct();

computer.displaySpecifications();

}

}

**Output:**

yaml

Copy code

CPU: Intel i9

RAM: 32GB

Storage: 1TB SSD

Graphics Card: Yes

Bluetooth: Yes

In this example:

* The GamingComputerBuilder builds a high-performance gaming computer by setting various components like CPU, RAM, storage, and more.
* The Director is responsible for constructing the Computer object using the builder.
* The client code uses the builder to create a complex Computer object without directly interacting with its inner parts.

**Conclusion:**

This example demonstrates how the Builder pattern simplifies the construction of complex objects with various configurations, ensuring that you can reuse the construction process while maintaining flexibility in the object’s final state.

**STRUCTURAL DESIGN PATTERN**

### ****Adapter Design Pattern****

The **Adapter Design Pattern** is a **structural design pattern** used to enable incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.

### ****Key Points****

1. **Purpose**: Converts the interface of a class into another interface that a client expects.
2. **Use Case**: When you want to use an existing class but its interface doesn't match the requirement.
3. **Types**:
   1. **Class Adapter**: Uses inheritance to adapt one interface to another.
   2. **Object Adapter**: Uses composition to delegate requests to an adaptee object.

### ****Components****

1. **Target**: The interface expected by the client. PaymentGatway(Interface)
2. **Adaptee**: The existing class with an incompatible interface.(PayPal, Stripe
3. **Adapter**: A class that implements the target interface and translates requests to the adaptee.(PaypalAdapter, StripeAdapter)
4. PaymentProcessor - main

### ****Implementation Steps****

1. Define a target interface.
2. Create an adaptee class with an incompatible interface.
3. Implement an adapter class to bridge the target interface and adaptee.

The **Adapter Design Pattern** is a structural pattern used to bridge the gap between two incompatible interfaces by providing a "wrapper" or "adapter" class. It allows an existing class to be used with a different interface.

### Real-Time Example: Payment Gateway Integration

In a real-world scenario, imagine you are building an e-commerce application. The application needs to support multiple payment gateways (e.g., PayPal, Stripe, and Razorpay). Each payment gateway has its own API and interface. To standardize interaction with these gateways, you can use the Adapter pattern.

### Code Example in Java:

#### Step 1: Define a Common Interface

Define a common interface (PaymentGateway) that all adapters will implement.

java

Copy code

public interface PaymentGateway {

void processPayment(String amount);

}

#### Step 2: Existing Classes (Incompatible Interfaces)

Suppose you have different payment gateway classes with incompatible methods:

**PayPal API:**

java

Copy code

public class PayPal {

public void makePayment(String amount) {

System.out.println("Payment of " + amount + " processed through PayPal.");

}

}

**Stripe API:**

java

Copy code

public class Stripe {

public void initiatePayment(String amount) {

System.out.println("Payment of " + amount + " processed through Stripe.");

}

}

#### Step 3: Create Adapters

Create adapter classes that implement the PaymentGateway interface and adapt the existing payment classes.

**PayPal Adapter:**

java

Copy code

public class PayPalAdapter implements PaymentGateway {

private PayPal payPal;

public PayPalAdapter() {

this.payPal = new PayPal();

}

@Override

public void processPayment(String amount) {

payPal.makePayment(amount);

}

}

**Stripe Adapter:**

java

Copy code

public class StripeAdapter implements PaymentGateway {

private Stripe stripe;

public StripeAdapter() {

this.stripe = new Stripe();

}

@Override

public void processPayment(String amount) {

stripe.initiatePayment(amount);

}

}

#### Step 4: Client Code

Use the adapters to interact with the different payment gateways through the common interface.

java

Copy code

public class PaymentProcessor {

public static void main(String[] args) {

PaymentGateway payPalAdapter = new PayPalAdapter();

payPalAdapter.processPayment("100");

PaymentGateway stripeAdapter = new StripeAdapter();

stripeAdapter.processPayment("200");

}

}

### Output:

Copy code

Payment of 100 processed through PayPal.

Payment of 200 processed through Stripe.

### Advantages:

1. **Reusability:** You can reuse existing classes without modifying their code.
2. **Interoperability:** Makes it easy to work with APIs or classes with incompatible interfaces.
3. **Open/Closed Principle:** New adapters can be added without changing the existing code.

This example demonstrates how the Adapter pattern helps in integrating multiple payment gateways seamlessly into a single unified interface.

### Proxy Design Pattern in Java

The Proxy design pattern provides a surrogate or placeholder for another object to control access to it. It is one of the structural design patterns in Java.

#### Types of Proxies:

1. **Virtual Proxy**: Used to delay the creation and initialization of an expensive object until it is actually needed.
2. **Protection Proxy**: Controls access to an object by providing different levels of access rights.
3. **Remote Proxy**: Represents an object located in a different address space (e.g., in distributed systems).
4. **Smart Proxy**: Provides additional functionality, such as reference counting or logging, whenever an object is accessed.

#### Components of Proxy Pattern:

1. **Subject Interface**: Common interface for RealSubject and Proxy.
2. **RealSubject**: The actual object that the Proxy represents.
3. **Proxy**: The surrogate object that controls access to the RealSubject.

#### Implementation Example:

java

Copy code

// Subject interface

public interface Image {

void display();

}

// RealSubjectpublic class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk(); // Expensive operation

}

private void loadFromDisk() {

System.out.println("Loading " + fileName);

}

@Override

public void display() {

System.out.println("Displaying " + fileName);

}

}

// Proxypublic class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

// Clientpublic class ProxyPatternDemo {

public static void main(String[] args) {

Image image = new ProxyImage("test\_image.jpg");

// Image is loaded and displayed

image.display();

// Image is displayed without loading again

image.display();

}

}

#### Advantages:

1. Adds a level of indirection to manage complex or resource-intensive objects.
2. Improves performance through lazy initialization.
3. Enhances security and access control.

#### Disadvantages:

1. Increases code complexity.
2. May introduce latency due to the additional layer of abstraction.

The **Observer Pattern** is a behavioral design pattern in Java where an object (called the subject) maintains a list of its dependents (called observers) and notifies them of any state changes, typically by calling their update method. This pattern is useful for implementing distributed event-handling systems.

### Key Features of Observer Pattern:

1. **Decoupling**: Observers and subjects are loosely coupled.
2. **Dynamic Relationships**: Observers can be added or removed at runtime.
3. **Push or Pull Model**: The subject can push updates to the observers or allow them to pull updates as needed.

### Components of the Observer Pattern:

**Subject (Observable)**:

* 1. Maintains a list of observers.
  2. Provides methods to add, remove, and notify observers.

**Observer**:

* 1. An interface or abstract class defining the update method.
  2. Concrete classes implement this interface to react to state changes.

### Steps to Implement in Java:

**Define the Subject interface**:

java

Copy code

interface Subject {

void addObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**Define the Observer interface**:

java

Copy code

interface Observer {

void update(String message);

}

**Create a Concrete Subject**:

java

Copy code

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

class ConcreteSubject implements Subject {

private List<Observer> observers = new ArrayList<>();

private String state;

@Override

public void addObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(state);

}

}

public void setState(String state) {

this.state = state;

notifyObservers();

}

}

**Create a Concrete Observer**:

java

Copy code

class ConcreteObserver implements Observer {

private String name;

public ConcreteObserver(String name) {

this.name = name;

}

@Override

public void update(String message) {

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

}

}

**Usage Example**:

java

Copy code

public class ObserverPatternExample {

public static void main(String[] args) {

ConcreteSubject subject = new ConcreteSubject();

Observer observer1 = new ConcreteObserver("Observer 1");

Observer observer2 = new ConcreteObserver("Observer 2");

subject.addObserver(observer1);

subject.addObserver(observer2);

subject.setState("New State!");

subject.removeObserver(observer1);

subject.setState("Another State!");

}

}

### Java Built-in Support:

Java provides the java.util.Observer interface and java.util.Observable class, but they are deprecated in newer versions (starting Java 9) due to design limitations. The recommended approach is to implement the Observer Pattern manually as demonstrated above.

### ****Strategy Pattern in Java: Short Notes****

The **Strategy Pattern** is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable, promoting flexibility and reusability.

### ****Key Concepts****

1. **Context**: Maintains a reference to a Strategy object and uses it to execute the algorithm.
2. **Strategy Interface**: Defines a common interface for all supported algorithms.
3. **Concrete Strategies**: Implement different variations of the algorithm.

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from clients that use it.

### Real-Time Example: Payment System

Imagine you are developing an e-commerce application where customers can choose from multiple payment methods like Credit Card, PayPal, or Google Pay.

### 1. ****Strategy Interface****

The interface defines a common contract for all payment methods.

java

Copy code

public interface PaymentStrategy {

void pay(double amount);

}

### 2. ****Concrete Strategies****

Each payment method implements the PaymentStrategy interface.

#### PayPal Payment Strategy

java

Copy code

public class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

#### Credit Card Payment Strategy

java

Copy code

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

#### Google Pay Payment Strategy

java

Copy code

public class GooglePayPayment implements PaymentStrategy {

private String phoneNumber;

public GooglePayPayment(String phoneNumber) {

this.phoneNumber = phoneNumber;

}

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using Google Pay.");

}

}

### 3. ****Context Class****

The context class interacts with the PaymentStrategy.

java

Copy code

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public PaymentContext(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

paymentStrategy.pay(amount);

}

}

### 4. ****Usage Example****

The client code dynamically selects the desired payment method.

java

Copy code

public class StrategyPatternExample {

public static void main(String[] args) {

// Pay using PayPal

PaymentContext context = new PaymentContext(new PayPalPayment("user@example.com", "password123"));

context.executePayment(150.75);

// Pay using Credit Card

context = new PaymentContext(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

context.executePayment(200.50);

// Pay using Google Pay

context = new PaymentContext(new GooglePayPayment("9876543210"));

context.executePayment(100.00);

}

}

### Output

arduino

Copy code

Paid 150.75 using PayPal.

Paid 200.5 using Credit Card.

Paid 100.0 using Google Pay.

### Advantages of the Strategy Pattern in This Example:

1. **Open/Closed Principle**: Adding a new payment method doesn’t require changes to existing code.
2. **Code Reusability**: Payment strategies are encapsulated and can be reused independently.
3. **Flexibility**: Payment methods can be swapped dynamically at runtime.

### ****Command Pattern Overview****

The Command Pattern is a behavioral design pattern used to encapsulate a request as an object, allowing parameterization of objects with different requests, delayed execution, or logging of requests.

### ****Key Concepts****

1. **Command**: An interface that declares an execution method (e.g., execute()).
2. **ConcreteCommand**: Implements the Command interface and defines the binding between a Receiver and an action.
3. **Receiver**: The actual object that performs the action when the command is executed.
4. **Invoker**: Holds a command and invokes its execute() method.
5. **Client**: Configures the objects and relationships, linking commands to receivers.

In Java, the **Command Pattern** is a behavioral design pattern used to encapsulate a request as an object, allowing you to parameterize objects with different requests, delay execution of requests, or support undoable operations.

Here’s a **real-time example** using the Command Pattern for a **Personal Finance Tracker** application:

### Scenario

The application supports financial transactions such as adding income, recording expenses, and transferring funds. These actions can be undone or redone.

### Key Components

1. **Command Interface**: Defines an executable operation.
2. **Concrete Command**: Implements the operation (e.g., AddIncome, RecordExpense).
3. **Invoker**: Executes commands (e.g., TransactionManager).
4. **Receiver**: The object that performs the actual work (e.g., Account).

### Implementation

java

Copy code

// Command Interfacepublic interface Command {

void execute();

void undo();

}

// Receiverclass Account {

private double balance;

public Account(double balance) {

this.balance = balance;

}

public void add(double amount) {

balance += amount;

System.out.println("Added: " + amount + ", New Balance: " + balance);

}

public void deduct(double amount) {

balance -= amount;

System.out.println("Deducted: " + amount + ", New Balance: " + balance);

}

public double getBalance() {

return balance;

}

}

// Concrete Commandsclass AddIncomeCommand implements Command {

private Account account;

private double amount;

public AddIncomeCommand(Account account, double amount) {

this.account = account;

this.amount = amount;

}

@Override

public void execute() {

account.add(amount);

}

@Override

public void undo() {

account.deduct(amount);

}

}

class RecordExpenseCommand implements Command {

private Account account;

private double amount;

public RecordExpenseCommand(Account account, double amount) {

this.account = account;

this.amount = amount;

}

@Override

public void execute() {

account.deduct(amount);

}

@Override

public void undo() {

account.add(amount);

}

}

// Invokerclass TransactionManager {

private final Stack<Command> commandHistory = new Stack<>();

public void executeCommand(Command command) {

command.execute();

commandHistory.push(command);

}

public void undoLastCommand() {

if (!commandHistory.isEmpty()) {

Command command = commandHistory.pop();

command.undo();

} else {

System.out.println("No commands to undo.");

}

}

}

// Main Classpublic class PersonalFinanceTracker {

public static void main(String[] args) {

Account account = new Account(1000.0);

TransactionManager manager = new TransactionManager();

Command addIncome = new AddIncomeCommand(account, 500.0);

Command recordExpense = new RecordExpenseCommand(account, 200.0);

// Execute Commands

manager.executeCommand(addIncome); // Added: 500.0, New Balance: 1500.0

manager.executeCommand(recordExpense); // Deducted: 200.0, New Balance: 1300.0

// Undo Commands

manager.undoLastCommand(); // Added: 200.0, New Balance: 1500.0

manager.undoLastCommand(); // Deducted: 500.0, New Balance: 1000.0

}

}

### Explanation

1. **Account**: The Receiver that performs the actual operations like adding or deducting amounts.
2. **AddIncomeCommand** & **RecordExpenseCommand**: Concrete implementations of the Command interface, encapsulating the actions.
3. **TransactionManager**: The Invoker that executes and manages commands, including their undo operations.

### Benefits in Real-Time Use

1. **Undo/Redo Functionality**: Essential for a finance tracker where users may make mistakes.
2. **Scalability**: New operations like "TransferFunds" can easily be added without modifying existing code.
3. **Encapsulation**: Each transaction is encapsulated as an independent object, making the system modular and extensible.

This example demonstrates the flexibility and power of the Command Pattern in creating a robust and maintainable system.