Design Patterns Java Interview Questions

In software engineering, a Design Pattern describes an established solution to the most commonly encountered problems in software design.

Creation patterns

Singleton

Factory

Abstract factory

Builder

Structural Patterns

Proxy

Decorator

Adapter

Bridge

Facade

Flywegth

Behavioral Patterns

Memento

Mediator

State

Visitor

Command

Observer

Template method

Strategy

Chain of responsibility

Other Architectural Patterns

Service locator

Front controller

Intercepting filter

Null object

Gateway

**Creational Patterns**

Creational Design Patterns are concerned with the way in which objects are created. They reduce complexities and instability by creating objects in a controlled manner.

Singleton Design Pattern

The Singleton Design Pattern aims to keep a check on initialization of objects of a particular class by ensuring that only one instance of the object exists throughout the Java Virtual Machine.

lazy with static inner class

public class Singleton {

private Singleton() {}

private static class SingletonHolder {

public static final Singleton instance = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.instance;

}

}

static final field (not lazy)

public class Singleton {

public static final Singleton INSTANCE = new Singleton();

private Singleton() {}

}

Problems With Serialization and Deserialization In order to serialize the above singleton classes, we must implement those classes with a Serializable interface. But doing that is not enough. When deserializing a class, new instances are created. Now it doesn't matter the constructor is private or not. Now there can be more than one instance of the same singleton class inside the JVM, violating the singleton property.

Problems With Reflection An advanced user can change the private access modifier of the constructor to anything they want at runtime using reflection. If this happens, our only mechanism for non-instantiability breaks.

All of the above problems can be solved very easily by using the enum type to make singletons.

**Singleton With Enum**

The three lines above make a singleton without any of the problems discussed. Since enums are inherently serializable, we don't need to implement it with a serializable interface. The reflection problem is also not there. Therefore, it is 100% guaranteed that only one instance of the singleton is present within a JVM. Thus, this method is recommended as the best method of making singletons in Java.

How to Use

public enum SingletonEnum {

INSTANCE;

int value;

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

}

Main class implementation:

public class EnumDemo {

public static void main(String[] args) {

SingletonEnum singleton = SingletonEnum.INSTANCE;

System.out.println(singleton.getValue());

singleton.setValue(2);

System.out.println(singleton.getValue());

}

}

**When to Use Singleton Design Pattern**

For resources that are expensive to create (like database connection objects)

It's good practice to keep all loggers as Singletons which increases performance

Classes which provide access to configuration settings for the application

Classes that contain resources that are accessed in shared mode

**Factory Design Pattern**

The Factory Design Pattern or Factory Method Design Pattern is one of the most used design patterns in Java.

According to GoF, this pattern “defines an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses”.

In this example, we'll create a Polygon interface which will be implemented by several concrete classes. A PolygonFactory will be used to fetch objects from this family

Let's first create the Polygon interface

public interface Polygon {

String getType();

}

Next, we'll create a few implementations like Square, Triangle, etc. that implement this interface and return an object of Polygon type Now we can create a factory that takes the number of sides as an argument and returns the appropriate implementation of this interface

public class PolygonFactory {

public Polygon getPolygon(int numberOfSides) {

if(numberOfSides == 3) {

return new Triangle();

}

if(numberOfSides == 4) {

return new Square();

}

if(numberOfSides == 5) {

return new Pentagon();

}

if(numberOfSides == 7) {

return new Heptagon();

}

else if(numberOfSides == 8) {

return new Octagon();

}

return null;

}

}

**When to Use Factory Method Design Pattern**

When the implementation of an interface or an abstract class is expected to change frequently

When the current implementation cannot comfortably accommodate new change

When the initialization process is relatively simple, and the constructor only requires a handful of parameters

**Abstract Factory Design Pattern**

In the previous section, we saw how the Factory Method design pattern could be used to create objects related to a single family.

By contrast, the Abstract Factory Design Pattern is used to create families of related or dependent objects. It's also sometimes called a factory of factories.

**Builder Design Pattern**

When the complexity of creating object increases, the Builder pattern can separate out the instantiation process by using another object (a builder) to construct the object.

Note that all the access modifiers on the fields are declared private since we don't want outer objects to access them directly.

The constructor is also private so that only the Builder assigned to this class can access it. All of the properties set in the constructor are extracted from the builder object which we supply as an argument.

public class BankAccount {

private String name;

private String accountNumber;

private String email;

private boolean newsletter;

// constructors/getters

public static class BankAccountBuilder {

private String name;

private String accountNumber;

private String email;

private boolean newsletter;

public BankAccountBuilder(String name, String accountNumber) {

this.name = name;

this.accountNumber = accountNumber;

}

public BankAccountBuilder withEmail(String email) {

this.email = email;

return this;

}

public BankAccountBuilder wantNewsletter(boolean newsletter) {

this.newsletter = newsletter;

return this;

}

public BankAccount build() {

return new BankAccount(this);

}

}

}

Let's see a quick example of the builder pattern in action

BankAccount newAccount = new BankAccount

.BankAccountBuilder("Jon", "22738022275")

.withEmail("jon@example.com")

.wantNewsletter(true)

.build();

When to Use Builder Pattern

When the process involved in creating an object is extremely complex, with lots of mandatory and optional parameters

When an increase in the number of constructor parameters leads to a large list of constructors

When client expects different representations for the object that's constructed

The Lombok library provides a great way of simplifying data objects. One of the key features of Project Lombok is the @Builder annotation, which automatically creates Builder classes for creating immutable objects.

**Structural Patterns**

Structural Patterns deal with the composition of classes and objects. They provide different ways of using object composition and inheritance to create some abstraction.

**Proxy Pattern**

With this pattern, we create an intermediary that acts as an interface to another resource, e.g., a file, a connection. This secondary access provides a surrogate for the real component and protects it from the underlying complexity.

Key Points of Differentiation:

The proxy provides the same interface as the object it's holding the reference to, and it doesn't modify the data in any manner; it's in contrast to Adapter and Decorator patterns which alter and decorate the functionalities of pre-existing instances respectively

The Proxy usually has the information about the real subject at the compile time itself whereas Decorator and Adapter get injected at runtime, knowing only the actual object's interface

Consider a heavy Java object (like a JDBC connection or a SessionFactory) that requires some initial configuration.

We only want such objects to be initialized on demand, and once they are, we'd want to reuse them for all calls. Let's now create a simple interface and the configuration for this object:

public interface ExpensiveObject {

void process();

}

And the implementation of this interface with a large initial configuration:

public class ExpensiveObjectImpl implements ExpensiveObject {

public ExpensiveObjectImpl() {

heavyInitialConfiguration();

}

@Override

public void process() {

LOG.info("processing complete.");

}

private void heavyInitialConfiguration() {

LOG.info("Loading initial configuration...");

}

}

We'll now utilize the Proxy pattern and initialize our object on demand:

public class ExpensiveObjectProxy implements ExpensiveObject {

private static ExpensiveObject object;

@Override

public void process() {

if (object == null) {

object = new ExpensiveObjectImpl();

}

object.process();

}

}

Note that we're calling the process() method twice. Behind the scenes, the settings part will occur only once – when the object is first initialized.

Let's talk about when to use the Proxy pattern:

When we want a simplified version of a complex or heavy object. In this case, we may represent it with a skeleton object which loads the original object on demand, also called as lazy initialization. This is known as the Virtual Proxy

When the original object is present in different address space, and we want to represent it locally. We can create a proxy which does all the necessary boilerplate stuff like creating and maintaining the connection, encoding, decoding, etc., while the client accesses it as it was present in their local address space. This is called the Remote Proxy

When we want to add a layer of security to the original underlying object to provide controlled access based on access rights of the client. This is called Protection Proxy

**Decorator Pattern**

This pattern is useful for enhancing the behavior of an object. Key points of differentiation:

Although Proxy and Decorator patterns have similar structures, they differ in intention; while Proxy's prime purpose is to facilitate ease of use or controlled access, a Decorator attaches additional responsibilities

Both Proxy and Adapter patterns hold a reference to the original object

All the decorators from this pattern can be used recursively, an infinite number of times, which is not possible with other models

Suppose we have a Christmas tree object and we want to decorate it. The decoration does not change the object itself; it’s just that in addition to the Christmas tree, we're adding some decoration items like garland, tinsel, tree-topper, bubble lights, etc.

First, we'll create a ChristmasTree interface and its implementation:

public interface ChristmasTree {

String decorate();

}

The implementation of this interface will look like:

public class ChristmasTreeImpl implements ChristmasTree {

@Override

public String decorate() {

return "Christmas tree";

}

}

We'll now create an abstract TreeDecorator class for this tree. This decorator will implement the ChristmasTree interface as well as hold the same object. The implemented method from the same interface will simply call the decorate() method from our interface:

public abstract class TreeDecorator implements ChristmasTree {

private ChristmasTree tree;

// standard constructors

@Override

public String decorate() {

return tree.decorate();

}

}

We'll now create some decorating element. These decorators will extend our abstract TreeDecorator class and will modify its decorate() method according to our requirement:

public class BubbleLights extends TreeDecorator {

public BubbleLights(ChristmasTree tree) {

super(tree);

}

public String decorate() {

return super.decorate() + decorateWithBubbleLights();

}

private String decorateWithBubbleLights() {

return " with Bubble Lights";

}

}

This is a good choice in the following cases:

When we wish to add, enhance or even remove the behavior or state of objects

When we just want to modify the functionality of a single object of class and leave others unchanged

Adapter Pattern

The Adapter pattern is used for connecting two incompatible interfaces that otherwise cannot be connected directly. An Adapter wraps an existing class with a new interface so that it becomes compatible with the interface needed.

The main differences between Adapter and Proxy patterns are:

While proxy provides the same interface, Adapter provides a different interface that’s compatible with its client

Adapter pattern is used after the application components are designed so that we can use them without modifying the source code. This is in contrast to the Bridge pattern, which is used before the components are designed.

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).

To deal with this problem, we'll create an adapter which will convert the values and give us the desired results

First, we'll create the original interface Movable which is supposed to return the speed of some luxury cars in miles per hour:

public interface Movable {

// returns speed in MPH

double getSpeed();

}

We'll now create one concrete implementation of this interface:

public class BugattiVeyron implements Movable {

@Override

public double getSpeed() {

return 268;

}

}

Now we'll create an adapter interface MovableAdapter that will be based on the same Movable class. It may be slightly modified to yield different results in different scenarios:

public interface MovableAdapter {

// returns speed in KM/H

double getSpeed();

}

The implementation of this interface will consist of private method convertMPHtoKMPH() that will be used for the conversion:

public class MovableAdapterImpl implements MovableAdapter {

private Movable luxuryCars;

**// standard constructors**

@Override

public double getSpeed() {

return convertMPHtoKMPH(luxuryCars.getSpeed());

}

private double convertMPHtoKMPH(double mph) {

return mph \* 1.60934;

}

}

When to Use Adapter Pattern

When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application. A suitable Adapter can be developed to make them compatible with each other

When our application is not compatible with the interface that our client is expecting

When we want to reuse legacy code in our application without making any modification in the original code

Bridge Pattern

The Bridge pattern is used to decouple an abstraction from its implementation so that the two can vary independently.

This means to create a bridge interface that uses OOP principles to separate out responsibilities into different abstract classes.

Key Points of Differentiation:

A Bridge pattern can only be implemented before the application is designed.

Allows an abstraction and implementation to change independently whereas an Adapter pattern makes it possible for incompatible classes to work together

For the Bridge pattern, we'll consider two layers of abstraction; one is the geometric shape (like triangle and square) which is filled with different colors (our second abstraction layer)

First, we'll define a color interface:

public interface Color {

String fill();

}

Now we'll create a concrete class for this interface:

public class Blue implements Color {

@Override

public String fill() {

return "Color is Blue";

}

}

Let's now create an abstract Shape class which consists a reference (bridge) to the Color object:

public abstract class Shape {

protected Color color;

//standard constructors

abstract public String draw();

}

We'll now create a concrete class of Shape interface which will utilize method from Color interface as well:

public class Square extends Shape {

public Square(Color color) {

super(color);

}

@Override

public String draw() {

return "Square drawn. " + color.fill();

}

}

This is a good choice in the following cases:

When we want a parent abstract class to define the set of basic rules, and the concrete classes to add additional rules

When we have an abstract class that has a reference to the objects, and it has abstract methods that will be defined in each of the concrete classes

**Facade Pattern**

Simply put, a facade encapsulates a complex subsystem behind a simple interface. It hides much of the complexity and makes the subsystem easy to use.

Besides a much simpler interface, there's one more benefit of using this design pattern. It decouples a client implementation from the complex subsystem. Thanks to this, we can make changes to the existing subsystem and don't affect a client.

**Flyweight Pattern**

This pattern is used to reduce the memory footprint. It can also improve performance in applications where object instantiation is expensive.

The flyweight object's state is made up of an invariant component shared with other similar objects (intrinsic) and a variant component which can be manipulated by the client code (extrinsic).

It's very important that the flyweight objects are immutable: any operation on the state must be performed by the factory.

The main elements of the pattern are:

an interface which defines the operations that the client code can perform on the flyweight object

one or more concrete implementations of our interface

a factory to handle objects instantiation and caching

To begin with, we'll create a Vehicle interface. Since this interface will be the return type of the factory method we need to make sure to expose all the relevant methods

public void start();

public void stop();

public Color getColor();

Next up, let's make a Car class as a concrete Vehicle. Our car will implement all the methods of the vehicle interface. As for its state, it'll have an engine and a color field:

private Engine engine;

private Color color;

Last but not least, we'll create the VehicleFactory. Building a new vehicle is a very expensive operation so the factory will only create one vehicle per color.

In order to do that, we keep track of the created vehicles using a map as a simple cache:

private static Map<Color, Vehicle> vehiclesCache

= new HashMap<>();

public static Vehicle createVehicle(Color color) {

Vehicle newVehicle = vehiclesCache.computeIfAbsent(color, newColor -> {

Engine newEngine = new Engine();

return new Car(newEngine, newColor);

});

return newVehicle;

}

Behavioral Patterns

**Memento Pattern**

The Memento Design Pattern offers a solution to implement undoable actions. We can do this by saving the state of an object at a given instant and restoring it if the actions performed since need to be undone.

Practically, the object whose state needs to be saved is called an Originator. The Caretaker is the object triggering the save and restore of the state, which is called the Memento.

The Memento object should expose as little information as possible to the Caretaker. This is to ensure that we don't expose the internal state of the Originator to the outside world, as it would break encapsulation principles. However, the Originator should access enough information in order to restore to the original state.

the Originator can produce and consume a Memento. Meanwhile, the Caretaker only keeps the state before restoring it. The internal representation of the Originator is kept hidden from the external world.

Typically, the Memento Design Pattern will be used in situations where some actions are undoable, therefore requiring to rollback to a previous state. However, if the state of the Originator is heavy, using the Memento Design Pattern can lead to an expensive creation process and increased use of memory.

Let's now see an example of the Memento Design Pattern. Let's imagine we have a text editor:

public class TextEditor {

private TextWindow textWindow;

public TextEditor(TextWindow textWindow) {

this.textWindow = textWindow;

}

}

It has a text window, which holds the currently entered text, and provides a way to add more text:

public class TextWindow {

private StringBuilder currentText;

public TextWindow() {

this.currentText = new StringBuilder();

}

public void addText(String text) {

currentText.append(text);

}

}

Now, let's imagine we want our text editor to implement some save and undo features. When saving, we want our current text to be saved. Thus, when undoing subsequent changes, we'll have our saved text restored.

In order to do that, we'll make use of the Memento Design Pattern. First, we'll create an object holding the current text of the window:

public class TextWindowState {

private String text;

public TextWindowState(String text) {

this.text = text;

}

public String getText() {

return text;

}

}

This object is our Memento. As we can see, we choose to use String instead of StringBuilder to prevent any update of the current text by outsiders.

After that, we'll have to provide the TextWindow class with methods to create and consume the Memento object, making the TextWindow our Originator:

public TextWindowState save() {

return new TextWindowState(wholeText.toString());

}

public void restore(TextWindowState save) {

currentText = new StringBuilder(save.getText());

}

Finally, we have to update our TextEditor class. As the Caretaker, it will hold the state of the Originator and ask to restore it when needed:

private TextWindowState savedTextWindow;

public void hitSave() {

savedTextWindow = textWindow.save();

}

public void hitUndo() {

textWindow.restore(savedTextWindow);

}

Let's see if it works through a sample run. Imagine we add some text to our editor, save it, then add some more and, finally, undo. In order to achieve that, we'll add a print() method on our TextEditor that returns a String of the current text:

TextEditor textEditor = new TextEditor(new TextWindow());

textEditor.write("The Memento Design Pattern\n");

textEditor.write("How to implement it in Java?\n");

textEditor.hitSave();

textEditor.write("Buy milk and eggs before coming home\n");

textEditor.hitUndo();

assertThat(textEditor.print()).isEqualTo("The Memento Design Pattern\nHow to implement it in Java?\n");

Mediator Pattern

The intent of the Mediator Pattern is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another. This is achieved by creating a mediator object that takes care of the interaction between dependent objects. Consequently, all the communication goes through the mediator.

This promotes loose coupling, as a set of components working together no longer have to interact directly. Instead, they only refer to the single mediator object. This way, it is also easier to reuse these objects in other parts of the system.

Imagine we're building a simple cooling system that consists of a fan, a power supply, and a button. Pressing the button will either turn on or turn off the fan. Before we turn the fan on, we need to turn on the power. Similarly, we have to turn off the power right after the fan is turned off.

Now, let's implement the Mediator Pattern to reduce the dependencies between our classes and make the code more reusable.

First, let's introduce the Mediator class:

public class Mediator {

private Button button;

private Fan fan;

private PowerSupplier powerSupplier;

// constructor, getters and setters

public void press() {

if (fan.isOn()) {

fan.turnOff();

} else {

fan.turnOn();

}

}

public void start() {

powerSupplier.turnOn();

}

public void stop() {

powerSupplier.turnOff();

}

}

let's modify the remaining classes:

public class Button {

private Mediator mediator;

// constructor, getters and setters

public void press() {

mediator.press();

}

}

public class Fan {

private Mediator mediator;

private boolean isOn = false;

// constructor, getters and setters

public void turnOn() {

mediator.start();

isOn = true;

}

public void turnOff() {

isOn = false;

mediator.stop();

}

}

Now that we've implemented the Mediator Pattern, none of the Button, Fan, or PowerSupplier classes communicate directly. They only have a single reference to the Mediator. If we need to add a second power supply in the future, all we have to do is to update Mediator's logic; Button and Fan classes remain untouched.

This example shows how easily we can separate dependent objects and make our system easier to maintain.

**When to Use the Mediator Pattern**

The Mediator Pattern is a good choice if we have to deal with a set of objects that are tightly coupled and hard to maintain. This way we can reduce the dependencies between objects and decrease the overall complexity.

Additionally, by using the mediator object, we extract the communication logic to the single component, therefore we follow the Single Responsibility Principle. Furthermore, we can introduce new mediators with no need to change the remaining parts of the system. Hence, we follow the Open-Closed Principle.

Sometimes, however, we may have too many tightly coupled objects due to the faulty design of the system. If this is a case, we should not apply the Mediator Pattern. Instead, we should take one step back and rethink the way we've modeled our classes.

As with all other patterns, we need to consider our specific use case before blindly implementing the Mediator Pattern.

**State Pattern**

The main idea of State pattern is to allow the object for changing its behavior without changing its class. Also, by implementing it, the code should remain cleaner without many if/else statements.

Imagine we have a package which is sent to a post office, the package itself can be ordered, then delivered to a post office and finally received by a client. Now, depending on the actual state, we want to print its delivery status.

The simplest approach would be to add some boolean flags and apply simple if/else statements within each of our methods in the class. That won't complicate it much in a simple scenario. However, it might complicate and pollute our code when we'll get more states to process which will result in even more if/else statements.

Thanks to the State design pattern, we can encapsulate the logic in dedicated classes, apply the Single Responsibility Principle and Open/Closed Principle, have cleaner and more maintainable code.

First, let's define our context, that's going to be a Package class:

public class Package {

private PackageState state = new OrderedState();

// getter, setter

public void previousState() {

state.prev(this);

}

public void nextState() {

state.next(this);

}

public void printStatus() {

state.printStatus();

}

}

As we can see, it contains a reference for managing the state, notice previousState(), nextState() and printStatus() methods where we delegate the job to the state object. The states will be linked to each other and every state will set another one based on this reference passed to both methods. Next, we're going to have the PackageState which has three methods with the following signatures:

public interface PackageState {

void next(Package pkg);

void prev(Package pkg);

void printStatus();

}

This interface will be implemented by each concrete state class.

The first concrete state will be OrderedState:

public class OrderedState implements PackageState {

@Override

public void next(Package pkg) {

pkg.setState(new DeliveredState());

}

@Override

public void prev(Package pkg) {

System.out.println("The package is in its root state.");

}

@Override

public void printStatus() {

System.out.println("Package ordered, not delivered to the office yet.");

}

}

Let's have a look at the DeliveredState class:

public class DeliveredState implements PackageState {

@Override

public void next(Package pkg) {

pkg.setState(new ReceivedState());

}

@Override

public void prev(Package pkg) {

pkg.setState(new OrderedState());

}

@Override

public void printStatus() {

System.out.println("Package delivered to post office, not received yet.");

}

}

First, the strategy pattern defines a family of interchangeable algorithms. Generally, they achieve the same goal, but with a different implementation, for example, sorting or rendering algorithms.

In state pattern, the behavior might change completely, based on actual state.

Next, in strategy, the client has to be aware of the possible strategies to use and change them explicitly. Whereas in state pattern, each state is linked to another and create the flow as in Finite State Machine.

The state design pattern is great when we want to avoid primitive if/else statements. Instead, we extract the logic to separate classes and let our context object delegate the behavior to the methods implemented in the state class.

**Visitor Pattern**

The purpose of a Visitor pattern is to define a new operation without introducing the modifications to an existing object structure.

Imagine that we have a composite object which consists of components. The object's structure is fixed – we either can't change it, or we don't plan to add new types of elements to the structure.

Now, how could we add new functionality to our code without modification of existing classes?

The Visitor design pattern might be an answer. Simply put, we'll have to do is to add a function which accepts the visitor class to each element of the structure. That way our components will allow the visitor implementation to “visit” them and perform any required action on that element. In other words, we'll extract the algorithm which will be applied to the object structure from the classes. Consequently, we'll make good use of the Open/Closed principle as we won't modify the code, but we'll still be able to extend the functionality by providing a new Visitor implementation.

Our example will be custom Document object that consists of JSON and XML concrete elements; the elements have a common abstract superclass, the Element.

The Document class:

public class Document extends Element {

List<Element> elements = new ArrayList<>();

// ...

@Override

public void accept(Visitor v) {

for (Element e : this.elements) {

e.accept(v);

}

}

}

The Element class has an abstract method which accepts the Visitor interface:

public abstract void accept(Visitor v);

Therefore, when creating the new element, name it the JsonElement, we'll have to provide the implementation this method.

However, due to nature of the Visitor pattern, the implementation will be the same, so in most cases, it would require us to copy-paste the boilerplate code from other, already existing element:

public class JsonElement extends Element {

// ...

public void accept(Visitor v) {

v.visit(this);

}

}

Since our elements allow visiting them by any visitor, let's say that we want to process our Document elements, but each of them in a different way, depending on its class type.

Therefore, our visitor will have a separate method for the given type:

public class ElementVisitor implements Visitor {

@Override

public void visit(XmlElement xe) {

System.out.println(

"processing an XML element with uuid: " + xe.uuid);

}

@Override

public void visit(JsonElement je) {

System.out.println(

"processing a JSON element with uuid: " + je.uuid);

}

}

For testing purpose, let's have a look at VisitorDemoclass:

public class VisitorDemo {

public static void main(String[] args) {

Visitor v = new ElementVisitor();

Document d = new Document(generateUuid());

d.elements.add(new JsonElement(generateUuid()));

d.elements.add(new JsonElement(generateUuid()));

d.elements.add(new XmlElement(generateUuid()));

d.accept(v);

}

// ...

}

As each design pattern, even the Visitor has its downsides, particularly, its usage makes it more difficult to maintain the code if we need to add new elements to the object's structure.

For example, if we add new YamlElement, then we need to update all existing visitors with the new method desired for processing this element. Following this further, if we have ten or more concrete visitors, that might be cumbersome to update all of them.

The Visitor pattern is great to separate the algorithm from the classes on which it operates. Besides that, it makes adding new operation more easily, just by providing a new implementation of the Visitor.

Furthermore, we don't depend on components interfaces, and if they are different, that's fine, since we have a separate algorithm for processing per concrete element.

**Command Pattern**

Simply put, the pattern intends to encapsulate in an object all the data required for performing a given action (command), including what method to call, the method's arguments, and the object to which the method belongs.

This model allows us to decouple objects that produce the commands from their consumers, so that's why the pattern is commonly known as the producer-consumer pattern.

In a classic implementation, the command pattern requires implementing four components:

**the Command,**

**the Receiver,**

**the Invoker,**

and the Client.

To understand how the pattern works and the role that each component plays, let's create a basic example. Let's suppose that we want to develop a text file application. In such a case, we should implement all functionality required for performing some text-file related operations, such as opening, writing, saving a text file, and so forth.

A command is an object whose role is to store all the information required for executing an action, including the method to call, the method arguments, and the object (known as the receiver) that implements the method.

@FunctionalInterface

public interface TextFileOperation {

String execute();

}

public class OpenTextFileOperation implements TextFileOperation {

private TextFile textFile;

// constructors

@Override

public String execute() {

return textFile.open();

}

}

public class SaveTextFileOperation implements TextFileOperation {

// same field and constructor as above

@Override

public String execute() {

return textFile.save();

}

}

the TextFileOperation commands encapsulate all the information required for opening and saving a text file, including the receiver object, the methods to call, and the arguments (in this case, no arguments are required, but they could be).

A receiver is an object that performs a set of cohesive actions. It's the component that performs the actual action when the command's execute() method is called.

public class TextFile {

private String name;

// constructor

public String open() {

return "Opening file " + name;

}

public String save() {

return "Saving file " + name;

}

// additional text file methods (editing, writing, copying, pasting)

}

An invoker is an object that knows how to execute a given command but doesn't know how the command has been implemented. It only knows the command's interface.

public class TextFileOperationExecutor {

private final List<TextFileOperation> textFileOperations

= new ArrayList<>();

public String executeOperation(TextFileOperation textFileOperation) {

textFileOperations.add(textFileOperation);

return textFileOperation.execute();

}

}

A client is an object that controls the command execution process by specifying what commands to execute and at what stages of the process to execute them.

public static void main(String[] args) {

TextFileOperationExecutor textFileOperationExecutor

= new TextFileOperationExecutor();

textFileOperationExecutor.executeOperation(

new OpenTextFileOperation(new TextFile("file1.txt"))));

textFileOperationExecutor.executeOperation(

new SaveTextFileOperation(new TextFile("file2.txt"))));

}

As the TextFileOperation interface is a functional interface, we can pass command objects in the form of lambda expressions to the invoker, without having to create the TextFileOperation instances explicitly:

TextFileOperationExecutor textFileOperationExecutor

= new TextFileOperationExecutor();

textFileOperationExecutor.executeOperation(() -> "Opening file file1.txt");

textFileOperationExecutor.executeOperation(() -> "Saving file file1.txt");

**Observer Pattern**

It specifies communication between objects: observable and observers. An observable is an object which notifies observers about the changes in its state.

For example, a news agency can notify channels when it receives news. Receiving news is what changes the state of the news agency, and it causes the channels to be notified.

Let's see how we can implement it ourselves.

First, let's define the NewsAgency class:

public class NewsAgency {

private String news;

private List<Channel> channels = new ArrayList<>();

public void addObserver(Channel channel) {

this.channels.add(channel);

}

public void removeObserver(Channel channel) {

this.channels.remove(channel);

}

public void setNews(String news) {

this.news = news;

for (Channel channel : this.channels) {

channel.update(this.news);

}

}

}

To be able to do that, the observable object needs to keep references to the observers, and in our case, it's the channels variable.

**Template Method Pattern**

It makes it easier to implement complex algorithms by encapsulating logic in a single method.

The ComputerBuilder class is responsible for outlining the steps required to build a computer by declaring methods for adding and setting up different components, such as a motherboard and a processor.

public abstract class ComputerBuilder {

// ...

public final Computer buildComputer() {

addMotherboard();

setupMotherboard();

addProcessor();

return new Computer(computerParts);

}

public abstract void addMotherboard();

public abstract void setupMotherboard();

public abstract void addProcessor();

// ...

}

Here, the build() method is the template method, which defines steps of the algorithm for assembling the computer parts and returns fully-initialized Computer instances.

Notice that it's declared as final to prevent it from being overridden. With the base class already set, let's try to use it by creating two subclasses. One which builds a “standard” computer, and the other that builds a “high-end” computer:

public class StandardComputerBuilder extends ComputerBuilder {

@Override

public void addMotherboard() {

computerParts.put("Motherboard", "Standard Motherboard");

}

@Override

public void setupMotherboard() {

motherboardSetupStatus.add(

"Screwing the standard motherboard to the case.");

motherboardSetupStatus.add(

"Pluging in the power supply connectors.");

motherboardSetupStatus.forEach(

step -> System.out.println(step));

}

@Override

public void addProcessor() {

computerParts.put("Processor", "Standard Processor");

}

}

**Strategy Pattern**

Essentially, the strategy pattern allows us to change the behavior of an algorithm at runtime.

Typically, we would start with an interface which is used to apply an algorithm, and then implement it multiple times for each possible algorithm.

Let's say we have a requirement to apply different types of discounts to a purchase, based on whether it's a Christmas, Easter or New Year. First, let's create a Discounter interface which will be implemented by each of our strategies:

public interface Discounter {

BigDecimal applyDiscount(BigDecimal amount);

}

Then let's say we want to apply a 50% discount at Easter and a 10% discount at Christmas. Let's implement our interface for each of these strategies:

public static class EasterDiscounter implements Discounter {

@Override

public BigDecimal applyDiscount(final BigDecimal amount) {

return amount.multiply(BigDecimal.valueOf(0.5));

}

}

public static class ChristmasDiscounter implements Discounter {

@Override

public BigDecimal applyDiscount(final BigDecimal amount) {

return amount.multiply(BigDecimal.valueOf(0.9));

}

}

usage

Discounter easterDiscounter = new EasterDiscounter();

BigDecimal discountedValue = easterDiscounter

.applyDiscount(BigDecimal.valueOf(100));

assertThat(discountedValue)

.isEqualByComparingTo(BigDecimal.valueOf(50));

Chain of Responsibility Pattern

The Chain of Responsibility pattern is handy for:

Decoupling a sender and receiver of a command

Picking a processing strategy at processing-time

Let's first create an abstract base class for our processors:

public abstract class AuthenticationProcessor {

public AuthenticationProcessor nextProcessor;

// standard constructors

public abstract boolean isAuthorized(AuthenticationProvider authProvider);

}

Next, let's create concrete processors which extend AuthenticationProcessor:

public class OAuthProcessor extends AuthenticationProcessor {

public OAuthProcessor(AuthenticationProcessor nextProcessor) {

super(nextProcessor);

}

@Override

public boolean isAuthorized(AuthenticationProvider authProvider) {

if (authProvider instanceof OAuthTokenProvider) {

return true;

} else if (nextProcessor != null) {

return nextProcessor.isAuthorized(authProvider);

}

return false;

}

}

public class UsernamePasswordProcessor extends AuthenticationProcessor {

public UsernamePasswordProcessor(AuthenticationProcessor nextProcessor) {

super(nextProcessor);

}

@Override

public boolean isAuthorized(AuthenticationProvider authProvider) {

if (authProvider instanceof UsernamePasswordProvider) {

return true;

} else if (nextProcessor != null) {

return nextProcessor.isAuthorized(authProvider);

}

return false;

}

}

Here, we created two concrete processors for our incoming authorization requests: UsernamePasswordProcessor and OAuthProcessor.

For each one, we overrode the isAuthorized method.

Now let's create a couple of tests:

public class ChainOfResponsibilityTest {

private static AuthenticationProcessor getChainOfAuthProcessor() {

AuthenticationProcessor oAuthProcessor = new OAuthProcessor(null);

return new UsernamePasswordProcessor(oAuthProcessor);

}

@Test

public void givenOAuthProvider\_whenCheckingAuthorized\_thenSuccess() {

AuthenticationProcessor authProcessorChain = getChainOfAuthProcessor();

assertTrue(authProcessorChain.isAuthorized(new OAuthTokenProvider()));

}

@Test

public void givenSamlProvider\_whenCheckingAuthorized\_thenSuccess() {

AuthenticationProcessor authProcessorChain = getChainOfAuthProcessor();

assertFalse(authProcessorChain.isAuthorized(new SamlTokenProvider()));

}

}

The example above creates a chain of authentication processors: UsernamePasswordProcessor -> OAuthProcessor. In the first test, the authorization succeeds, and in the other, it fails.

First, UsernamePasswordProcessor checks to see if the authentication provider is an instance of UsernamePasswordProvider.

Not being the expected input, UsernamePasswordProcessor delegates to OAuthProcessor.

Last, the OAuthProcessor processes the command. In the first test, there is a match and the test passes. In the second, there are no more processors in the chain, and, as a result, the test fails.

We need to keep few important principles in mind while implementing Chain of Responsibility:

Each processor in the chain will have its implementation for processing a command In our example above, all processors have their implementation of isAuthorized

Every processor in the chain should have reference to the next processor Above, UsernamePasswordProcessor delegates to OAuthProcessor

Each processor is responsible for delegating to the next processor so beware of dropped commands Again in our example, if the command is an instance of SamlProvider then the request may not get processed and will be unauthorized

Processors should not form a recursive cycle In our example, we don't have a cycle in our chain: UsernamePasswordProcessor -> OAuthProcessor. But, if we explicitly set UsernamePasswordProcessor as next processor of OAuthProcessor, then we end up with a cycle in our chain: UsernamePasswordProcessor -> OAuthProcessor -> UsernamePasswordProcessor. Taking the next processor in the constructor can help with this

Only one processor in the chain handles a given command In our example, if an incoming command contains an instance of OAuthTokenProvider, then only OAuthProcessor will handle the command

Disadvantages And now that we've seen how interesting Chain of Responsibility is, let's keep in mind some drawbacks:

Mostly, it can get broken easily:

if a processor fails to call the next processor, the command gets dropped

if a processor calls the wrong processor, it can lead to a cycle

It can create deep stack traces, which can affect performance

It can lead to duplicate code across processors, increasing maintenance

Other Architectural Patterns

Service Locator Pattern

The purpose of the Service Locator pattern is to return the service instances on demand. This is useful for decoupling service consumers from concrete classes.

An implementation will consist of the following components:

Client – the client object is a service consumer. It's responsible for invoking the request from the service locator

Service Locator – is a communication entry point for returning the services from the cache

Cache – an object for storing service references to reuse them later

Initializer – creates and registers references to services in the cache

Service – the Service component represents the original services or their implementation

The original service object is looked up by the locator and returned on demand.

First, we'll create a MessagingService interface for sending messages in different ways:

public interface MessagingService {

String getMessageBody();

String getServiceName();

}

Next, we'll define two implementations of the interface above, that send messages through email and SMS:

public class EmailService implements MessagingService {

public String getMessageBody() {

return "email message";

}

public String getServiceName() {

return "EmailService";

}

}

The SMSService class definition is similar to the EmailService class.

After defining the two services, we have to define the logic to initialize them:

public class InitialContext {

public Object lookup(String serviceName) {

if (serviceName.equalsIgnoreCase("EmailService")) {

return new EmailService();

} else if (serviceName.equalsIgnoreCase("SMSService")) {

return new SMSService();

}

return null;

}

}

The last component we need before putting the service locator object together is the cache.

In our example, this is a simple class with a List property:

public class Cache {

private List<MessagingService> services = new ArrayList<>();

public MessagingService getService(String serviceName) {

// retrieve from the list

}

public void addService(MessagingService newService) {

// add to the list

}

}

Finally, we can implement our service locator class:

public class ServiceLocator {

private static Cache cache = new Cache();

public static MessagingService getService(String serviceName) {

MessagingService service = cache.getService(serviceName);

if (service != null) {

return service;

}

InitialContext context = new InitialContext();

MessagingService service1 = (MessagingService) context

.lookup(serviceName);

cache.addService(service1);

return service1;

}

}

The logic here is fairly simple.

The class holds an instance of the Cache. Then, in the getService() method, it will first check the cache for an instance of the service. Then, if that's null, it will call the initializing logic and add the new object to the cache. Finally, let's consider a few reasons to avoid using the Service Locator pattern.

One argument against it is that it makes unit testing difficult. With dependency injection, we can pass mock objects of the dependent class to the tested instance. On the other hand, this is a bottleneck with the Service Locator pattern.

Another issue is that it's trickier to use APIs based on this pattern. The reason for this is that the dependencies are hidden inside the class and they're only verified at runtime.

Front Controller Pattern

n this tutorial we'll be digging deeper into the Front Controller Pattern, part of the Enterprise Patterns as defined in Martin Fowler‘s book “Patterns of Enterprise Application Architecture”.

Front Controller is defined as “a controller that handles all requests for a Web site”. It stands in front of a web-application and delegates requests to subsequent resources. It also provides an interface to common behavior such as security, internationalization and presenting particular views to certain users.

This enables an application to change its behavior at runtime. Furthermore it helps to read and maintain an application by preventing code duplication.

The Front Controller consolidates all request handling by channeling requests through a single handler object.

The Front Controller Pattern is mainly divided into two parts. A single dispatching controller and a hierarchy of commands. his single controller dispatches requests to commands in order to trigger behavior associated with a request.

Intercepting Filter Pattern

Intercepting Filters are filters that trigger actions before or after an incoming request is processed by a handler.

Intercepting filters represents centralized components in a web application, common to all requests and extensible without affecting existing handlers.

Let's extend the example from the previous guide and implement an authentication mechanism, request logging, and a visitor counter. In addition, we want the ability to deliver our pages in various different encoding.

All these are use cases for intercepting filters because they are common to all requests and should be independent of the handlers.

Null Object Pattern

In most object-oriented programming languages, we're not allowed to use a null reference. That's why we're often forced to write null checks:

Command cmd = getCommand();

if (cmd != null) {

cmd.execute();

}

Sometimes, if the number of such if statements get high, the code may become ugly, hard to read and error-prone. This is when the Null Object Pattern may come in handy.

The intent of the Null Object Pattern is to minimize that kind of null check. Instead, we can identify the null behavior and encapsulate it in the type expected by the client code. More often then not, such neutral logic is very simple – do nothing. This way we no longer need to deal with special handling of null references.

We simply may treat null objects the same way we treat any other instance of a given type that actually contains some more sophisticated business logic. Consequently, the client code stays cleaner.

As null objects should not have any state, there's no need to create identical instances multiple times. Thus, we'll often implement null objects as singletons.

As we can see, we can identify the following participants:

Client requires an instance of AbstractObject

AbstractObject defines the contract Client expects – it may also contain shared logic for the implementing classes

RealObject implements AbstractObject and provides real behavior

NullObject implements AbstractObject and provides neutral behavior

Imagine we have a message router application. Each message should have a valid priority assigned. Our system is supposed to route high priority messages to an SMS gateway whereas messages with medium priority should be routed to a JMS queue.

From time to time, however, messages with “undefined” or empty priority might come to our application. Such messages should be discarded from further processing.

First, we'll create the Router interface:

public interface Router {

void route(Message msg);

}

Next, let's create two implementations of the above interface – the one responsible for routing to an SMS gateway and the one that will route the messages to JMS queue:

public class SmsRouter implements Router {

@Override

public void route(Message msg) {

// implementation details

}

}

public class JmsRouter implements Router {

@Override

public void route(Message msg) {

// implementation details

}

}

Finally, let's implement our null object:

public class NullRouter implements Router {

@Override

public void route(Message msg) {

// do nothing

}

}

We're now ready to put all the pieces together. Let's see how the example client code may look like:

public class RoutingHandler {

public void handle(Iterable<Message> messages) {

for (Message msg : messages) {

Router router = RouterFactory.getRouterForMessage(msg);

router.route(msg);

}

}

}

As we can see, we treat all Router objects the same way, no matter what implementation is returned by the RouterFactory. This allows us to keep our code clean and readable. We should use the Null Object Pattern when a Client would otherwise check for null just to skip execution or perform a default action. In such cases, we may encapsulate the neutral logic within a null object and return that to the client instead of the null value. This way client's code no longer needs to be aware if a given instance is null or not.

Most of the developers would return Collections.emptyList() from findByNameAndLastname() in case none of the customers matches the provided search criteria. This is a very good example of following the Null Object Pattern.

In contrast, the getById() should return the customer with the given id. Someone calling this method expects to get the specific customer entity. In case no such customer exists we should explicitly return null to signal there is something wrong with the provided id.

As with all other patterns, we need to consider our specific use case before blindly implementing the Null Object Pattern. Otherwise, we may unintentionally introduce some bugs in our code that will be hard to find.

The Gateway Pattern

How to use the Gateway pattern to retrieve data from multiple services with a single request. Spring Cloud now also provides the Spring Cloud Gateway project which implements this pattern. One common use case for the Gateway pattern is to have endpoints that encapsulate commonly called services. This can increase performance by reducing the number of client requests.