

Design Patterns Use Cases (Java And Spring)

Creational

1. [Singleton](#)
 - a. Where centralised access, a single point of control, resource sharing, or global coordination is needed.
 - b. We can have a single class for inventory management, restaurant management
2. [Builder](#)
 - a. When an object has **many optional fields** or parameters.
 - b. To avoid a **telescoping constructor** (many overloaded constructors) and passing unwanted params during initialisation.
3. [Factory](#)
 - a. A Factory Pattern defines an interface for creating an object but lets subclasses decide which class to instantiate.
 - b. We can pick one among multiple strategies and instances
4. [Abstract Factory](#)
 - a. It's essentially a factory of factories — it creates related objects grouped under a common theme. Can be used in combination with template pattern to eliminate duplicate code.
5. [Object Pool](#)
 - a. Reusing objects that are expensive to create
6. [Prototype](#)

- a. create new objects by copying existing ones (**cloning**) instead of instantiating new ones from scratch

Behavioural

1. [Adapter](#)
 - a. The **Adapter Design Pattern** is a **structural pattern** used to allow the interface of an existing class to be used as another interface. It acts as a bridge between two incompatible interfaces.
 - b. Imagine you are integrating a third-party **Media Player** into your application, but your app expects a different interface. Instead of rewriting existing code, you write an adapter to connect both.
 - c. We can use a factory to pick the best version among the adapters
 - d. It will pick the best among multiple implementations
 - e. Autowired annotation and Map can be used in spring boot to get the corresponding instance with the Component qualifier name

2. [Bridge](#)
 - a. The Bridge Design Pattern is a structural pattern used to decouple abstraction from implementation, so that the two can vary independently.

Why People Say "Adapter or Adapter"

This informal phrase may come up because:

- The **Bridge pattern** introduces an **abstraction layer on top of another abstraction**, much like **multiple adapters chained together**.
- In some complex architectures, you may build **adapters that use other adapters internally** (e.g., in a plug-in system), and the result starts resembling a **Bridge** structurally.
- From a bird's eye view, both patterns are solving integration problems — **Bridge for decoupling, Adapter for compatibility**.

- You want to avoid a **combinatorial explosion** of classes.
- You want to **decouple abstraction from implementation** for better flexibility.
- You expect to change or extend **both abstraction and implementation** independently.
- **Adapter pattern:** You already have a StripeSDK and want to adapt it to your internal interface PaymentProcessor.
- **Bridge pattern:** You're designing the system from scratch to support multiple payment types (Stripe, Razorpay, PayPal) and multiple payment methods (card, wallet, UPI) with **separate hierarchies**.
- Autowired annotation and Map can be used in spring boot to get the corresponding instance with the Component qualifier name and type

3. Decorator (Wrapper)

- a. **The Decorator Design Pattern** (also known as the **Wrapper Pattern**) is a structural pattern used to dynamically add behavior or responsibilities to an object without modifying its code. This is especially useful when you want to extend functionality in a **flexible and reusable way, without subclassing**.
- b. We want to notify users via **email**, but also allow optional **SMS** and **Slack** notifications **without altering the original notifier**.
- c. Reusing an object without modifying it
- d. We can use Decorator via Configuration (Bean annotation) or Primary annotation

| Scenario | Use Decorator | Use Observer |
|--|----------------------|---------------------|
| Add retry, logging, or email formatting | ✓ Yes | ✗ No |
| Notify multiple systems (email, SMS, etc.) | ✗ No | ✓ Yes |
| Modify behavior of a single notifier | ✓ Yes | ✗ No |
| Broadcast events to multiple listeners | ✗ No | ✓ Yes |

Pattern Intent

| Pattern | Purpose |
|------------------|--|
| Decorator | Dynamically adds behavior to a single object by wrapping it. Example: Retry, logging, analytics |
| Observer | Publishes updates to multiple subscribers when an event occurs. Example: Send email, SMS, and webhook in response to a user signup |

4. [Composite](#)

- The **Composite Design Pattern** is a structural pattern used to treat **individual objects** and **compositions of objects** uniformly. It allows you to build tree-like structures where nodes can be either **leaf** (basic objects) or **composites** (containers of other objects).
- File And Directories

5. [Filter \(Criteria\)](#)

- The **Filter (Criteria) Design Pattern**, a structural pattern (often treated as behavioral as well), allows you to filter a set of objects using different criteria and chaining them using logical operations such as AND, OR, NOT.

6. [Proxy](#)

- The **Proxy Design Pattern** provides a surrogate or placeholder for another object to control access to it.
- Used for:
 - Access control (e.g. protecting real objects)
 - Lazy initialization (e.g. expensive object creation)
 - Remote proxies (e.g. RMI)
 - Logging / auditing / caching

7. [Flyweight](#)

- The **Flyweight Design Pattern** is a structural pattern used to minimize memory usage or computational expenses by sharing as much data as possible with similar objects.



When to Use

- You have many objects that consume a lot of memory.
- Most object data is intrinsic (shared) and only a small part is extrinsic (context-specific).
- You want to reuse objects instead of creating new ones each time.



Real-World Use Cases

| Use Case | Description |
|-----------------------------------|--|
| Text Editor (characters) | Characters share font info (intrinsic), but position is extrinsic. |
| Game Trees (chess, go) | Reuse similar game pieces instead of creating new ones. |
| Icons in UI | One icon object rendered in many places with different positions. |
| Particle systems | Reuse particle shape, size etc., change position/color only. |
| Map rendering (e.g., Google Maps) | Reuse marker shapes/icons across locations. |

8. [Facade](#)



What is the Facade Pattern?

The **Facade pattern** provides a simplified interface to a complex subsystem.
It's useful when:

- You want to **hide system complexity** from the client.
- You want to **provide a unified interface** to multiple components.

Structural

1. [Chain Of Responsibility](#)

- a. The **Chain of Responsibility (CoR)** design pattern allows you to **pass a request along a chain of handlers** until one of them handles it. It's useful when multiple objects can handle a request and you want to **decouple the sender and receiver**.
 - b. Validator Chain, Logger Chain, Middleware
2. [Command \(Action Or Transaction\)](#)

- a. The **Command Design Pattern** encapsulates a request as an object, thereby allowing you to **parameterize clients with queues, requests, and operations**, and support **undoable operations**. Can be used in a combination with queue to process commands together

Task Queue

```
@PostConstruct
public void startExecutionLoop() {
    // Start a background thread to execute commands
    new Thread(this::processCommands).start();
}
```

| Benefit | Description |
|----------------------------------|--|
| Decouples invocation & execution | Useful in background job systems |
| FIFO processing | Ensures order of command execution |
| Supports async workloads | Schedule, batch, or throttle command dispatching |

3. [Iterator](#)
- a. The Iterator pattern allows you to traverse a collection (like a list, set, or custom data structure) one element at a time, without knowing its internal structure.
 - b. Iterating over services or tasks.
 - c. Streamlining batch operations.

- d. Custom collection frameworks – when creating your own data structures.
- e. Traversal abstraction – when you want to hide the internal structure (tree, graph, etc.).
- f. Multiple traversals – forward, reverse, or filtered iteration.
- g. Undo functionality – iterate over a history stack.
- h. Promotes encapsulation by hiding collection internals.
- i. Allows multiple traversals.
- j. Clean separation between collection and traversal logic.

4. [Interpreter](#)

- a. The Interpreter Pattern defines a representation for a grammar and provides an interpreter to deal with this grammar. It is used when we want to evaluate language expressions.
- b. Arithmetic expression evaluators ($2 + 3 * 4$)
- c. Rule engines (Json Rule Engine)
- d. SQL parsing
- e. Mini-language processing
- f. Regex parsing
- g. Search filters

5. [Mediator](#)

- a. It allows you to encapsulate the interaction between objects (colleagues) into a separate mediator object, improving code maintainability and scalability.
- b. The **Mediator Design Pattern** is a behavioral design pattern that **centralizes complex communications** and control logic between related objects in a system, promoting **loose coupling** by preventing objects from referring to each other explicitly.
- c. Chat System and Notification Channel
- d. Socket.io

6. [Memento](#)

- Without exposing the object's internal structure, you can save and restore its state.
- AI/ML rollbacks

Summary Table

| Domain | Use Case |
|------------------|---------------------------------------|
| Text Editors | Undo/Redo, Auto-save |
| Games | Save/Load checkpoints |
| Web Forms | Multi-step form navigation |
| Document Editors | Version control, draft recovery |
| Business Apps | Manual rollback of transactions |
| Config Tools | Reset to defaults, profile switching |
| Workflow Engines | Resume from saved workflow stage |
| FSMs | Rollback state transitions |
| Simulators | Snapshot and restore |
| AI/ML Training | Save best model state during training |

| Use Case | Example |
|---------------------------------------|---|
| Undo/Redo in editors | Text editors (Notepad, Word, etc.) |
| Game Save/Load | Saving a game state to resume later |
| Workflow snapshots | Save intermediate workflow state for rollback |
| Versioning | Object version history (drafts, edits) |
| Database transactions (manual) | Rollback logic where a memento stores object state before a transaction |

7. [Null State](#)

- The **Null Object Design Pattern** provides an object as a surrogate for the absence of a real object. Instead of returning null, return a special object that implements the expected interface but does nothing (a "do-nothing" implementation). This helps eliminate null checks and NullPointerExceptions.
- Used to eliminate null checks

✓ Real-world Use Cases

| Context | Use Case |
|------------------|---|
| Repositories | <code>findById()</code> returns NullObject if not found |
| Logging | NoOpLogger avoids logging if disabled |
| Strategy Pattern | Fallback behavior when no strategy matches |
| User Sessions | Return a GuestUser instead of null session |
| Spring Boot | Use Optional beans or empty object patterns |

8. `Observer`

- The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (the Subject) changes state, all its dependents (Observers) are notified and updated automatically.
- Email, SMS, or push notifications

| Role | Description |
|-------------------------|---|
| Subject | Maintains a list of observers and notifies them |
| Observer | Defines an updating interface |
| ConcreteSubject | Stores state and notifies observers |
| ConcreteObserver | Implements the observer interface |

9. `State`

- The State Design Pattern is a behavioral pattern that allows an object to change its behavior when its internal state changes. It appears as if the object changed its class.
- ATM, Circuit Breaker, TCP/IP Machine, Game Dev

✓ Use Cases of State Design Pattern

| Use Case | Description |
|---------------------------------|--|
| Workflow/Process engines | e.g., Order states: Placed → Shipped → Delivered |
| Game development | Player states: Idle, Running, Jumping, Attacking |

| | |
|------------------------------|--|
| UI components | Button states: Enabled, Disabled, Hovered, Clicked |
| TCP connection states | OPEN, LISTEN, CLOSED, SYN_SENT, etc. |
| ATM machine | NoCard, HasCard, Authorized, OutOfService |

10. [Strategy](#)

- a. The **Strategy Design Pattern** is a **behavioral pattern** used to define a **family of algorithms**, encapsulate each one, and make them interchangeable at runtime. It enables selecting an algorithm's behavior at runtime.
- b. Define a set of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- c. Includes functions like execute, pay and compute, etc.

✓ Use Cases of Strategy Pattern

| Use Case | Example |
|----------------------------------|---|
| Payment gateways | Switch between different payment processors dynamically |
| Compression algorithms | Support ZIP, RAR, TAR, etc. |
| Sorting strategies | Choose QuickSort, MergeSort, BubbleSort based on dataset size |
| Validation strategies | Apply different validation based on user type or context |
| Route selection | GPS apps: fastest vs shortest vs scenic |
| Tax calculations | Different tax rules for regions or countries |
| Authentication mechanisms | |

11. [Template](#)

- a. Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses override certain steps of the algorithm without changing its structure.
- b. Workflows, code resuability and similar algorithms

| Use Case | Explanation |
|--|--|
| Frameworks where basic workflow is fixed | E.g., JUnit testing framework uses template methods for test lifecycle (<code>setUp</code> , <code>tearDown</code>). |
| Algorithms with invariant structure but variable steps | Sorting algorithms that share common parts but vary the pivot choice. |
| Code reuse with common algorithm parts | Base classes define the skeleton, subclasses override details. |
| Workflow engines and batch jobs | Fixed steps with flexible sub-tasks inside a process. |
| UI frameworks | Rendering lifecycle with customizable rendering steps. |
| Parsing and compiling | Generic parse method with steps overridden by different language parsers. |

12. `Visitor`

- a. Separate an algorithm from the objects on which it operates. Visitor lets you add further operations to objects without modifying them.
- b. Shopping Cart and Document Processing
- c. Imagine a **document processing system** with different types of documents:
 - **Invoice**
 - **Report**
- d. We want to:
 - Generate a **summary** for each document type.
 - Generate a **detailed report** for each document type.

| Use Case | Explanation |
|---------------------------|---|
| Complex object structures | When you have complex object hierarchies and want to perform operations across them without cluttering the objects. |

| | |
|--|---|
| Adding operations without changing classes | When you need to add new functionality frequently to unrelated classes but want to avoid modifying their code. |
| Compilers and AST traversal | Visiting different types of nodes in Abstract Syntax Trees (AST) for operations like code generation, optimization, or type checking. |
| Serialization / Deserialization | Performing different serialization strategies on diverse object types. |
| UI rendering systems | Rendering or processing different UI elements without embedding logic in the elements themselves. |
| Multiple unrelated operations | If multiple unrelated operations must be performed on a fixed set of objects, visitor helps keep those operations clean and separate. |

Creational

1. Singleton

Where centralised access, a single point of control, resource sharing, or global coordination is needed.

Java

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

    private Singleton() {}
```

```

public static synchronized Singleton getInstance() { // Thread Safe
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
}

```



Summary: When to Use Singleton in System Design

| System Component | Role of Singleton |
|-----------------------|-------------------------------------|
| Config Loader | Shared config for services |
| Token/Auth Manager | Centralized authentication control |
| Cache Manager | Single control of in-memory cache |
| Logging Service | Centralized, thread-safe logging |
| Thread Pool Manager | Controlled thread execution |
| Load Balancer Manager | Central routing decision point |
| Metrics Exporter | Avoid metric flooding |
| Broker / Coordinator | Maintain consensus / topology state |

Spring Boot



LoggerService.java – Singleton by Spring

```

package com.example.singleton.service;

import org.springframework.stereotype.Service;

@Service // This makes it a Spring-managed Singleton bean by default
public class LoggerService {

```

```
public LoggerService() {  
    System.out.println("LoggerService instance created.");  
}  
  
public void log(String message) {  
    System.out.println("[LOG] " + message);  
}  
}
```



LogController.java

```
package com.example.singleton.controller;  
  
import com.example.singleton.service.LoggerService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/log")  
public class LogController {  
  
    private final LoggerService loggerService;  
  
    @Autowired  
    public LogController(LoggerService loggerService) {  
        this.loggerService = loggerService;  
    }  
  
    @PostMapping  
    public String logMessage(@RequestParam String message) {  
        loggerService.log(message);  
        return "Logged: " + message;  
    }  
}
```

```
    }  
}
```

✓ Output on Server Start

Only **one instance** of `LoggerService` is created:

```
LoggerService instance created.
```

Even if you hit the `/log?message=Hello` endpoint multiple times, the same instance will be reused.

2. `Builder`

- When an object has **many optional fields** or parameters.
- To avoid a **telescoping constructor** (many overloaded constructors) and passing unwanted params during initialisation.

`Java`

```
public class User {  
    // Required fields  
    private final String firstName;  
    private final String lastName;  
  
    // Optional fields  
    private final int age;  
    private final String phone;  
    private final String address;  
  
    // Private constructor to enforce use of Builder  
    private User(UserBuilder builder) {  
        this.firstName = builder.firstName;
```

```
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    public static class UserBuilder {
        private final String firstName;
        private final String lastName;

        private int age;
        private String phone;
        private String address;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }

        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }

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

```

        }
    }

@Override
public String toString() {
    return "User: " + firstName + " " + lastName + ", Age: " + age + ", Phone: " +
}
}

```

```

public class Main {
    public static void main(String[] args) {
        User user = new User.UserBuilder("Sai", "Ashish")
            .age(23)
            .phone("1234567890")
            .address("Hyderabad")
            .build();

        System.out.println(user);
    }
}

```

Spring Boot

```

package com.example.model;

import lombok.Builder;
import lombok.Getter;
import lombok.ToString;

@Getter
@ToString
@Builder
public class User {
    private String firstName;
    private String lastName;
}

```

```
    private Integer age;  
    private String phone;  
    private String address;  
}
```

```
package com.example.controller;  
  
import com.example.dto.UserRequest;  
import com.example.model.User;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/api/users")  
public class UserController {  
  
    @PostMapping("/build")  
    public User buildUser(@RequestBody UserRequest req) {  
        return User.builder()  
            .firstName(req.getFirstName())  
            .lastName(req.getLastName())  
            .age(req.getAge())  
            .phone(req.getPhone())  
            .address(req.getAddress())  
            .build();  
    }  
}
```

3. Factory

A Factory Pattern defines an interface for creating an object but lets subclasses decide which class to instantiate.

Java



Factory Pattern Using `enum` in Java

✓ 1. Define the Product Interface

```
public interface Notification {  
    void notifyUser();  
}
```

✓ 2. Implement Concrete Notification Types

```
public class EmailNotification implements Notification {  
    public void notifyUser() {  
        System.out.println("Sending an EMAIL notification");  
    }  
}  
  
public class SMSNotification implements Notification {  
    public void notifyUser() {  
        System.out.println("Sending an SMS notification");  
    }  
}  
  
public class PushNotification implements Notification {  
    public void notifyUser() {  
        System.out.println("Sending a PUSH notification");  
    }  
}
```

✓ 3. Create Enum for Notification Types

```
public enum NotificationType {  
    EMAIL,
```

```
SMS,  
PUSH  
}  
}
```

✓ 4. Factory Class with `enum`

```
public class NotificationFactory {  
  
    public static Notification createNotification(NotificationType type) {  
        return switch (type) {  
            case EMAIL → new EmailNotification();  
            case SMS → new SMSNotification();  
            case PUSH → new PushNotification();  
        };  
    }  
}
```

✓ 5. Client Usage

```
public class Main {  
    public static void main(String[] args) {  
        Notification notification = NotificationFactory.createNotification(NotificationType.SMS);  
        notification.notifyUser(); // Output: Sending an SMS notification  
    }  
}
```

Spring Boot

1. Interface and Implementations

`Notification.java`

```
package com.example.model;

public interface Notification {
    String notifyUser();
}
```

EmailNotification.java

```
package com.example.model;

public class EmailNotification implements Notification {
    @Override
    public String notifyUser() {
        return "Sending an EMAIL notification";
    }
}
```

SMSNotification.java

```
package com.example.model;

public class SMSNotification implements Notification {
    @Override
    public String notifyUser() {
        return "Sending an SMS notification";
    }
}
```

PushNotification.java

```
package com.example.model;

public class PushNotification implements Notification {
    @Override
```

```
public String notifyUser() {  
    return "Sending a PUSH notification";  
}  
}
```

2. Enum

NotificationType.java

```
package com.example.type;  
  
public enum NotificationType {  
    EMAIL,  
    SMS,  
    PUSH  
}
```



3. Factory Class

NotificationFactory.java

```
package com.example.factory;  
  
import com.example.model.*;  
import com.example.type.NotificationType;  
import org.springframework.stereotype.Component;  
  
@Component  
public class NotificationFactory {  
  
    public Notification createNotification(NotificationType type) {  
        return switch (type) {  
            case EMAIL → new EmailNotification();  
            case SMS → new SMSNotification();  
        };  
    }  
}
```

```
        case PUSH → new PushNotification();
    };
}
}
```

4. Controller

NotificationController.java

```
package com.example.controller;

import com.example.factory.NotificationFactory;
import com.example.model.Notification;
import com.example.type.NotificationType;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class NotificationController {

    private final NotificationFactory factory;

    public NotificationController(NotificationFactory factory) {
        this.factory = factory;
    }

    @PostMapping("/notify")
    public String notifyUser(@RequestParam NotificationType type) {
        Notification notification = factory.createNotification(type);
        return notification.notifyUser();
    }
}
```



5. Main Application

NotificationFactoryApplication.java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class NotificationFactoryApplication {
    public static void main(String[] args) {
        SpringApplication.run(NotificationFactoryApplication.class, args);
    }
}
```

```
package com.sai.designPatterns.factory;

import com.sai.designPatterns.factory.enums.VehicleType;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service
@Slf4j
public class VehicleProvider {
```

```

private Map<VehicleType, Vehicle> vehicleMap;

@Autowired
public VehicleProvider(List<Vehicle> vehicleList){
    vehicleMap = new HashMap<>();
    for(Vehicle vehicle : vehicleList){
        vehicleMap.put(vehicle.getType(), vehicle);
    }
}

public Vehicle getVehicle(VehicleType type){
    return vehicleMap.get(type);
}

}

```

```

package com.sai.designPatterns.factory;

public class VehicleFactory {

    public static Vehicle create(String type, int wheel) {

        switch (type) {
            case "bike":
                return new Bike(wheel);
            case "car":
                return new Car(wheel);
            default:
                System.out.println("invalid type");
        }

        return null;
    }
}

```

```
}
```

```
package com.sai.designPatterns.factory;

import com.sai.designPatterns.factory.builder.VehicleBuilder;
import com.sai.designPatterns.factory.enums.VehicleType;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class VehicleService {

    private final VehicleProvider vehicleProvider;
    private final VehicleBuilder vehicleBuilder;

    public void process(){
        Vehicle vehicle = vehicleProvider.getVehicle(VehicleType.BIKE);
        System.out.println("Vehicle's Wheel: " + vehicle.getWheel());
        vehicle.process(vehicleBuilder.buildBikeContext());
    }

}
```

```
public interface Vehicle<T extends VehicleContext> {
    int getWheel();
    VehicleType getType();
    void process(T context);
}
```

```
@Service
@Slf4j
@RequiredArgsConstructor
```

```

public class Car implements Vehicle<CarContext> {

    private static final VehicleType type = VehicleType.CAR;
    private int wheel;

    public Car(int wheel) {
        this.wheel = wheel;
    }

    @Override
    public int getWheel() {
        return wheel;
    }

    @Override
    public VehicleType getType() {
        return type;
    }

    @Override
    public void process(CarContext context) {
        log.info(type + " :process");
    }
}

```

```

@Service
@Slf4j
@RequiredArgsConstructor
public class Bike implements Vehicle<BikeContext> {

    private static final VehicleType type = VehicleType.BIKE;
    private int wheel;

    public Bike(int wheel) {
        this.wheel = wheel;
    }
}

```

```
}

@Override
public int getWheel() {
    return wheel;
}

@Override
public VehicleType getType() {
    return type;
}

@Override
public void process(BikeContext bikeContext) {
    log.info(type + " : process");
}

}
```

```
public enum VehicleType {
    BIKE,
    CAR
}
```

```
@Component
public class VehicleBuilder {

    private static final String bikeld = "1GNEK13ZX4R118208";

    public BikeContext buildBikeContext(){
        BikeContext bikeContext = new BikeContext();
        bikeContext.setId(bikeld);
        return bikeContext;
    }
}
```

```
}
```

```
@Data  
public class BikeContext extends VehicleContext {  
}
```

```
@Data  
public class CarContext extends VehicleContext {  
}
```

```
@Data  
public class VehicleContext {  
    private String id;  
}
```

J VehicleFactory.java 1 X

```
factory > J VehicleFactory.java
SaiAshish, 17 months ago | 1 author (SaiAshish)
1 package com.sai.designPatterns.factory;           SaiAshish, 17 mo
2
3 public class VehicleFactory {
4
5     public static Vehicle create(String type, int wheel) {
6
7         switch (type) {
8             case "bike":
9                 return new Bike(wheel);
10            case "car":
11                return new Car(wheel);
12            default:
13                System.out.println("invalid type");
14        }
15
16        return null;
17    }
18}
19
20}
21
```

J VehicleBuilder.java 1 X

```
factory > builder > J VehicleBuilder.java > {} com.sai.designPatterns.factory.builder
1 package com.sai.designPatterns.factory.builder;          You, 13 months ago • Uncomm
2
3 import com.sai.designPatterns.factory.context.BikeContext;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class VehicleBuilder {
8
9     private static final String bikeId = "1GNEK13ZX4R118208";
10
11    public BikeContext buildBikeContext(){
12        BikeContext bikeContext = new BikeContext();
13        bikeContext.setId(bikeId);
14        return bikeContext;
15    }
16
17 }
18
19
20
```

J VehicleProvider.java 1 X

```
factory > J VehicleProvider.java > {} com.sai.designPatterns.factory
1 package com.sai.designPatterns.factory;           SaiAshish, 17 months
2
3 import com.sai.designPatterns.factory.enums.VehicleType;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import java.util.HashMap;
9 import java.util.List;
10 import java.util.Map;
11
12 @Service
13 @Slf4j
14 public class VehicleProvider {
15
16     private Map<VehicleType, Vehicle> vehicleMap;
17
18     @Autowired
19     public VehicleProvider(List<Vehicle> vehicleList){
20         vehicleMap = new HashMap<>();
21         for(Vehicle vehicle : vehicleList){
22             vehicleMap.put(vehicle.getType(), vehicle);
23         }
24     }
25
26     public Vehicle getVehicle(VehicleType type){
27         return vehicleMap.get(type);
28     }
29
30 }
31
```

J VehicleService.java 1 ×

factory > J VehicleService.java > {} com.sai.designPatterns.factory

```
1 package com.sai.designPatterns.factory;           You, 13 months ago • Uncommitted
2
3 import com.sai.designPatterns.factory.builder.VehicleBuilder;
4 import com.sai.designPatterns.factory.enums.VehicleType;
5 import lombok.RequiredArgsConstructor;
6 import org.springframework.stereotype.Service;
7
8 @Service
9 @RequiredArgsConstructor
10 public class VehicleService {
11
12     private final VehicleProvider vehicleProvider;
13     private final VehicleBuilder vehicleBuilder;
14
15     public void process(){
16         Vehicle vehicle = vehicleProvider.getVehicle(VehicleType.BIKE);
17         System.out.println("Vehicle's Wheel: " + vehicle.getWheel());
18         vehicle.process(vehicleBuilder.buildBikeContext());
19     }
20
21 }
22
```

J Vehicle.java 1 X

```
factory > J Vehicle.java > {} com.sai.designPatterns.factory
You, 13 months ago | 2 authors (You and one other)
1 package com.sai.designPatterns.factory;           SaiAshish, 17 months ago .
2
3 import com.sai.designPatterns.factory.context.VehicleContext;
4 import com.sai.designPatterns.factory.enums.VehicleType;
5
6 You, 13 months ago | 2 authors (You and one other)
7 //public abstract class Vehicle<T extends VehicleContext> {
8 //    public abstract int getWheel();
9 //    public abstract VehicleType getType();
10 //    public abstract void process(T context);
11 //    public String toString() {
12 //        return "Wheel: " + this.getWheel();
13 //    }
14 //
15 //
16 //}
17
18
19 public interface Vehicle<T extends VehicleContext> {
20     int getWheel();
21     VehicleType getType();
22     void process(T context);
23 }
```

J Car.java 1 X

```
factory > J Car.java > {} com.sai.designPatterns.factory
You, 14 months ago | 2 authors (SaiAshish and one other)
1 package com.sai.designPatterns.factory;           SaiAshish, 17 months ago • A
2
3 import com.sai.designPatterns.factory.context.CarContext;
4 import com.sai.designPatterns.factory.enums.VehicleType;
5 import lombok.RequiredArgsConstructor;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.stereotype.Service;
8
9 You, 14 months ago | 2 authors (SaiAshish and one other)
10 @Service
11 @Slf4j
12 @RequiredArgsConstructor
13 public class Car implements Vehicle<CarContext> {
14
15     private static final VehicleType type = VehicleType.CAR;
16     private int wheel;
17
18     public Car(int wheel) {
19         this.wheel = wheel;
20     }
21
22     @Override
23     public int getWheel() {
24         return wheel;
25     }
26
27     @Override
28     public VehicleType getType() {
29         return type;
30     }
31
32     @Override
33     public void process(CarContext context) {
34         log.info(type + " :process");
35     }
36 }
```

J Bike.java 1 X

factory > J Bike.java > {} com.sai.designPatterns.factory
You, 13 months ago | 2 authors (SaiAshish and one other)

```
1 package com.sai.designPatterns.factory;          SaiAshish, 17 months ago • Add
2
3 import com.sai.designPatterns.factory.context.BikeContext;
4 import com.sai.designPatterns.factory.enums.VehicleType;
5 import lombok.RequiredArgsConstructor;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.stereotype.Service;
8
9 You, 13 months ago | 2 authors (SaiAshish and one other)
10 @Service
11 @Slf4j
12 @RequiredArgsConstructor
13 public class Bike implements Vehicle<BikeContext> {
14
15     private static final VehicleType type = VehicleType.BIKE;
16     private int wheel;
17
18     public Bike(int wheel) {
19         this.wheel = wheel;
20     }
21
22     @Override
23     public int getWheel() {
24         return wheel;
25     }
26
27     @Override
28     public VehicleType getType() {
29         return type;
30     }
31
32     @Override
33     public void process(BikeContext bikeContext) {
34         log.info(type + " : process");
35     }
36 }
37
```

J VehicleType.java 1 ×

```
factory > enums > J VehicleType.java > {} com.sai.designPatterns.factory.enums
SaiAshish, 17 months ago | 1 author (SaiAshish)
1 package com.sai.designPatterns.factory.enums;           SaiAshish, 17 months ago
2
3 public enum VehicleType {
4     BIKE,
5     CAR
6 }
7
```

J BikeContext.java 3 ×

```
factory > context > J BikeContext.java > {} com.sai.designPatterns.factory.context
SaiAshish, 17 months ago | 1 author (SaiAshish)
1 package com.sai.designPatterns.factory.context;           SaiAshish, 17 months ago
2
3 import lombok.Data;
4
5 @Data
6 public class BikeContext extends VehicleContext {
7 }
8
```

J CarContext.java 3 X

```
factory > context > J CarContext.java > {} com.sai.designPatterns.factory.context
SaiAshish, 17 months ago | 1 author (SaiAshish)
1 package com.sai.designPatterns.factory.context;           SaiAshish, 17 months a
2
3 import lombok.Data;
4
5 SaiAshish, 17 months ago | 1 author (SaiAshish)
6 @Data
7 public class CarContext extends VehicleContext {
8 }
```

J VehicleContext.java 1 X

```
factory > context > J VehicleContext.java > {} com.sai.designPatterns.factory.context
You, 13 months ago | 2 authors (SaiAshish and one other)
1 package com.sai.designPatterns.factory.context;           SaiAshish, 17 mon
2
3 import lombok.Data;
4
5 You, 13 months ago | 2 authors (SaiAshish and one other)
6 @Data
7 public class VehicleContext {
8     private String id;
9 }
10
11
12
```

4. Abstract Factory

It's essentially a factory of factories — it creates related objects grouped under a common theme. Can be used in combination with template pattern to eliminate duplicate code.

✓ 1. Abstract Products

```
// Abstract Product A
public interface Button {
    void render();
}

// Abstract Product B
public interface Checkbox {
    void render();
}
```

✓ 2. Concrete Products

```
// Windows
public class WindowsButton implements Button {
    public void render() {
        System.out.println("Rendering Windows Button");
    }
}

public class WindowsCheckbox implements Checkbox {
    public void render() {
        System.out.println("Rendering Windows Checkbox");
    }
}

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

public class MacCheckbox implements Checkbox {
```

```
public void render() {  
    System.out.println("Rendering Mac Checkbox");  
}  
}
```

✓ 3. Abstract Factory

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

✓ 4. Concrete Factories

```
public class WindowsFactory implements GUIFactory {  
    public Button createButton() {  
        return new WindowsButton();  
    }  
  
    public Checkbox createCheckbox() {  
        return new WindowsCheckbox();  
    }  
}  
  
public class MacFactory implements GUIFactory {  
    public Button createButton() {  
        return new MacButton();  
    }  
  
    public Checkbox createCheckbox() {  
        return new MacCheckbox();  
    }  
}
```

✓ 5. Client

```
public class Application {  
    private final Button button;  
    private final Checkbox checkbox;  
  
    public Application(GUIFactory factory) {  
        this.button = factory.createButton();  
        this.checkbox = factory.createCheckbox();  
    }  
  
    public void renderUI() {  
        button.render();  
        checkbox.render();  
    }  
}
```

✓ 6. Demo Main Class

```
public class Main {  
    public static void main(String[] args) {  
        GUIFactory factory = new WindowsFactory(); // or new MacFactory()  
        Application app = new Application(factory);  
        app.renderUI();  
    }  
}
```

Abstract Factory + Template Pattern

J AbstractFactory.java 1 X

abstractFactory > J AbstractFactory.java > {} com.sai.designPatterns.abstractFactory

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish, 17 months ago • Add file
2
3     SaiAshish, 17 months ago | 1 author (SaiAshish)
4     public abstract class AbstractFactory implements Shape {
5
6         @Override
7         public void process(){
8             preprocess();
9             handleProcess();
10            postProcess();
11
12         abstract protected void preprocess();
13         abstract protected void handleProcess();
14         abstract protected void postProcess();
15
16     }
17
```

J FactoryProvider.java 1 X

```
abstractFactory > J FactoryProvider.java > {} com.sai.designPatterns.abstractFactory
    You, 13 months ago | 2 authors (SaiAshish and one other)
1 package com.sai.designPatterns.abstractFactory;           SaiAshish, 17 months
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 import java.util.HashMap;
7 import java.util.List;
8 import java.util.Map;
9
You, 13 months ago | 2 authors (SaiAshish and one other)
10 @Service
11 public class FactoryProvider {
12
13     private Map<ShapeType, Shape> shapeProviders;
14
15     @Autowired
16     public FactoryProvider(List<Shape> shapeList) {
17         shapeProviders = new HashMap<>();
18         for (Shape shape : shapeList) {
19             shapeProviders.put(shape.getType(), shape);
20         }
21     }
22
23     public Shape getShape(ShapeType shapeType){
24         return shapeProviders.get(shapeType);
25     }
26
27 }
28
29
30 //     public static AbstractFactory create(boolean rounded) {
31 //         if (rounded) {
32 //             return new RoundedShapeFactory();
33 //         } else {
34 //             return new ShapeFactory();
35 //         }
36 //     }
37
```

J Rectangle.java 1 ×

abstractFactory > J Rectangle.java > {} com.sai.designPatterns.abstractFactory

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish, 17
2
3 import org.springframework.stereotype.Service;
4
5 @Service
6 public class Rectangle extends AbstractFactory {
7
8     @Override
9     protected void preprocess() {
10         System.out.println(getType() + " :preProcess");
11     }
12
13     @Override
14     protected void handleProcess() {
15         System.out.println(getType() + " :handleProcess");
16     }
17
18     @Override
19     protected void postProcess() {
20         System.out.println(getType() + " :postProcess");
21     }
22
23     @Override
24     public ShapeType getType() {
25         return ShapeType.RECTANGLE;
26     }
27 }
28
```

J RoundedRectangle.java 1 ×

abstractFactory > J RoundedRectangle.java > {} com.sai.designPatterns.abstractF

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish,  
2  
3 import org.springframework.stereotype.Service;  
4  
5 @Service  
6 public class RoundedRectangle extends AbstractFactory {  
7  
8     @Override  
9     public ShapeType getType() {  
10         return ShapeType.ROUNDED_RECTANGLE;  
11     }  
12  
13     @Override  
14     protected void preProcess() {  
15  
16     }  
17  
18     @Override  
19     protected void handleProcess() {  
20  
21     }  
22  
23     @Override  
24     protected void postProcess() {  
25  
26     }  
27 }  
28
```

J RoundedSquare.java 1 X

```
al ~/Desktop/sai/folders/projects/DesignPaternsImplementations/src/main/java/com/sai/des  
ignPatterns/abstractFactory/RoundedSquare.java (preview ⌂) SaiASHISH, 17  
1 package com.sai.designPatterns.abstractfactory; SaiASHISH, 17  
2  
3 import org.springframework.stereotype.Service;  
4  
SaiASHISH, 17 months ago | 1 author (SaiASHISH)  
5 @Service  
6 public class RoundedSquare extends AbstractFactory {  
7  
8     @Override  
9     public ShapeType getType() {  
10         return ShapeType.ROUNDED_SQUARE;  
11     }  
12  
13     @Override  
14     protected void preProcess() {  
15  
16     }  
17  
18     @Override  
19     protected void handleProcess() {  
20  
21     }  
22  
23     @Override  
24     protected void postProcess() {  
25  
26     }  
27 }  
28
```

J Square.java 1 X

abstractFactory > J Square.java > {} com.sai.designPatterns.abstractFactory

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish
2
3 import org.springframework.stereotype.Service;
4
5 @Service
6 public class Square extends AbstractFactory {
7
8     @Override
9     public void process() {
10         System.out.println(x:"Square");
11     }
12
13     @Override
14     protected void preProcess() {
15
16     }
17
18     @Override
19     protected void handleProcess() {
20
21     }
22
23     @Override
24     protected void postProcess() {
25
26     }
27
28     @Override
29     public ShapeType getType() {
30         return ShapeType.SQUARE;
31     }
32
33 }
```

J RoundedShapeFactory.java 1 X

abstractFactory > J RoundedShapeFactory.java > {} com.sai.designPatterns.abstractFactory

You, 9 months ago | 2 authors (SaiAshish and one other)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish, 1
2
3 You, 9 months ago | 2 authors (SaiAshish and one other)
4 public class RoundedShapeFactory {
5     //extends AbstractFactory {
6     //    @Override
7     //    public Shape getShape(ShapeType shapeType) {
8     //        switch (shapeType) {
9     //            case ROUNDED_SQUARE: {
10 //                return new RoundedSquare();
11 //            }
12 //            case ROUNDED_RECTANGLE:
13 //                return new RoundedRectangle();
14 //            }
15 //        }
16    }
17
```

J Shape.java 1 ×

abstractFactory > J Shape.java > {} com.sai.designPatterns.abstractFactory

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish,  
2  
3     SaiAshish, 17 months ago | 1 author (SaiAshish)  
4     public interface Shape {  
5         void process();  
6         ShapeType getType();  
7     }
```

J ShapeFactory.java 1 ×

abstractFactory > J ShapeFactory.java > {} com.sai.designPatterns.abstractFactory

SaiAshish, 17 months ago | 1 author (SaiAshish)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish, 17 months  
2  
3     SaiAshish, 17 months ago | 1 author (SaiAshish)  
4     public class ShapeFactory {  
5         //extends AbstractFactory  
6         //    @Override  
7         //    public Shape getShape(ShapeType shapeType) {  
8         //        switch (shapeType){  
9         //            case SQUARE: {  
10            //                return new Square();  
11            //            }  
12            //            case RECTANGLE:  
13            //                return new Rectangle();  
14            //            }  
15            //        return null;  
16        }  
17    }
```

J ShapeType.java 1 X

abstractFactory > J ShapeType.java > {} com.sai.designPatterns.abstractFactory

You, 13 months ago | 2 authors (SaiAshish and one other)

```
1 package com.sai.designPatterns.abstractFactory;      SaiAshish
2
3 public enum ShapeType {
4     ROUNDED_RECTANGLE,
5     RECTANGLE,
6     ROUNDED_SQUARE,
7     SQUARE
8 }
9
10
11
```

5. Object Pool Pattern

Key Concepts

- **Expensive objects:** These are objects that are costly to instantiate (e.g., database connections, thread pools, socket connections).
- **Pooling:** Keeping a fixed number of objects and reusing them rather than creating/destroying repeatedly.
- **Borrow & Return:** Clients borrow objects from the pool and return them when done.

When to Use

- Object creation is expensive (time/memory/resources).
- You need a large number of short-lived objects.
- You want to control the number of instances due to system resource limits (e.g., threads, DB connections).

Structure

```
Client → ObjectPool → ReusableObject
```

Example in Java

```
class Reusable {  
    public void doWork() {  
        System.out.println("Using reusable object: " + this);  
    }  
}  
  
class ObjectPool {  
    private List<Reusable> available = new ArrayList<>();  
    private List<Reusable> inUse = new ArrayList<>();  
  
    public synchronized Reusable acquire() {  
        if (available.isEmpty()) {  
            available.add(new Reusable());  
        }  
        Reusable obj = available.remove(available.size() - 1);  
        inUse.add(obj);  
        return obj;  
    }  
  
    public synchronized void release(Reusable obj) {  
        inUse.remove(obj);  
    }  
}
```

```
    available.add(obj);
}
}
```

Usage:

```
ObjectPool pool = new ObjectPool();

Reusable r1 = pool.acquire();
r1.doWork();

pool.release(r1);
```



Real-World Examples

- **Database Connection Pooling** (e.g., HikariCP, Apache DBCP)
- **Thread Pool Executors** in Java
- **TCP Socket Pools**
- **Web Browser Tabs Pooling** (in performance browsers)

Pros and Cons

Pros:

- Reduces memory and CPU cost.
- Controls number of instances (helps with scalability).
- Performance boost with pre-initialized objects.

Cons:

- Increased complexity in managing the pool.
- Potential for memory leaks if objects are not returned.
- Not suitable for objects that are cheap to create or heavily stateful.

Example: Custom Object Pool in Spring Boot

Step 1: Create a Reusable Object

```
package com.example.pool;

public class Reusable {
    private final long id;

    public Reusable(long id) {
        this.id = id;
    }

    public void use() {
        System.out.println("Using object with ID: " + id);
    }

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

Step 2: Implement the Object Pool

```
package com.example.pool;

import org.springframework.stereotype.Component;

import java.util.*;
import java.util.concurrent.atomic.AtomicLong;

@Component
public class ReusableObjectPool {
    private final List<Reusable> available = new LinkedList<>();
    private final Set<Reusable> inUse = new HashSet<>();
```

```

private final int MAX_POOL_SIZE = 5;
private final AtomicLong idGenerator = new AtomicLong(1);

public synchronized Reusable acquire() {
    if (!available.isEmpty()) {
        Reusable obj = available.remove(0);
        inUse.add(obj);
        return obj;
    }

    if (inUse.size() < MAX_POOL_SIZE) {
        Reusable obj = new Reusable(idGenerator.getAndIncrement());
        inUse.add(obj);
        return obj;
    }

    throw new RuntimeException("All objects are in use.");
}

public synchronized void release(Reusable obj) {
    if (inUse.remove(obj)) {
        available.add(obj);
    }
}

public int availableCount() {
    return available.size();
}

public int inUseCount() {
    return inUse.size();
}

```

Step 3: Use It in a REST Controller

```

package com.example.pool;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/pool")
public class PoolController {

    private final ReusableObjectPool objectPool;

    public PoolController(ReusableObjectPool objectPool) {
        this.objectPool = objectPool;
    }

    @GetMapping("/acquire")
    public String acquire() {
        try {
            Reusable reusable = objectPool.acquire();
            reusable.use();
            return "Acquired object with ID: " + reusable.getId();
        } catch (RuntimeException e) {
            return "Pool is full. Try again later.";
        }
    }

    @PostMapping("/release/{id}")
    public String release(@PathVariable long id) {
        Reusable objToRelease = new Reusable(id);
        objectPool.release(objToRelease);
        return "Released object with ID: " + id;
    }

    @GetMapping("/status")
    public String status() {
        return "Available: " + objectPool.availableCount()
    }
}

```

```
        + ", In Use: " + objectPool.inUseCount();
    }
}
```

```
package com.sai.designPatterns.objectPool;

public class Connection {
    private String id;

    public Connection(String id) {
        this.id = id;
    }

    public void executeQuery(String query) {
        System.out.println("Executing query '" + query + "' on connection " + id);
    }
}
```

```
package com.sai.designPatterns.objectPool;

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

@Component
public class ConnectionPoolManager {
    private static final int MAX_POOL_SIZE = 5;

    private Queue<Connection> pool;

    public ConnectionPoolManager() {
        pool = new ConcurrentLinkedQueue<>();
```

```

        for (int i = 0; i < MAX_POOL_SIZE; i++) {
            pool.add(createConnection("Connection-" + (i + 1)));
        }
    }

public Connection borrowConnection() {
    if (pool.isEmpty()) {
        throw new RuntimeException("No available connections in the pool");
    }
    return pool.poll();
}

public void returnConnection(Connection connection) {
    if (pool.size() >= MAX_POOL_SIZE) {
        throw new RuntimeException("Pool is full, cannot return connection");
    }
    pool.add(connection);
}

private Connection createConnection(String id) {
    return new Connection(id);
}

```

```

package com.sai.designPatterns.objectPool;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class DatabaseService {

    private final ConnectionPoolManager connectionPoolManager;

```

```

public void executeQuery(String query) {
    Connection connection = null;
    try {
        connection = connectionPoolManager.borrowConnection();
        connection.executeQuery(query);
    } finally {
        if (connection != null) {
            connectionPoolManager.returnConnection(connection);
        }
    }
}

```

2. Create the Reusable Object (**ExpensiveWorker**)

```

package com.example.objectpool.pool;

public class ExpensiveWorker {
    private final long id;

    public ExpensiveWorker(long id) {
        this.id = id;
        simulateHeavyInitialization();
    }

    private void simulateHeavyInitialization() {
        try {
            Thread.sleep(500); // simulate expensive creation
        } catch (InterruptedException ignored) {}
    }

    public void doWork() {
        System.out.println("Worker " + id + " is working...");
    }
}

```

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



3. Create the Object Pool (WorkerPool)

```
package com.example.objectpool.pool;  
  
import org.springframework.stereotype.Component;  
  
import java.util.*;  
import java.util.concurrent.atomic.AtomicLong;  
  
@Component  
public class WorkerPool {  
  
    private final List<ExpensiveWorker> available = new LinkedList<>();  
    private final Set<ExpensiveWorker> inUse = new HashSet<>();  
    private final int MAX_POOL_SIZE = 5;  
    private final AtomicLong idGenerator = new AtomicLong(1);  
  
    public synchronized ExpensiveWorker acquire() {  
        if (!available.isEmpty()) {  
            ExpensiveWorker worker = available.remove(0);  
            inUse.add(worker);  
            return worker;  
        }  
  
        if (inUse.size() < MAX_POOL_SIZE) {  
            ExpensiveWorker worker = new ExpensiveWorker(idGenerator.getAndIncrement());  
            inUse.add(worker);  
            return worker;  
        }  
    }  
}
```

```

    }

    throw new RuntimeException("All workers are currently in use.");
}

public synchronized void release(long id) {
    Optional<ExpensiveWorker> workerOpt = inUse.stream()
        .filter(w → w.getId() == id)
        .findFirst();

    workerOpt.ifPresent(worker → {
        inUse.remove(worker);
        available.add(worker);
    });
}

public synchronized int getAvailableCount() {
    return available.size();
}

public synchronized int getInUseCount() {
    return inUse.size();
}

public synchronized List<Long> getInUseIds() {
    List<Long> ids = new ArrayList<>();
    for (ExpensiveWorker w : inUse) {
        ids.add(w.getId());
    }
    return ids;
}

public synchronized List<Long> getAvailableIds() {
    List<Long> ids = new ArrayList<>();
    for (ExpensiveWorker w : available) {
        ids.add(w.getId());
    }
}

```

```
    }
    return ids;
}
}
```

4. Expose REST API via **PoolController**

```
package com.example.objectpool.controller;

import com.example.objectpool.pool.ExpensiveWorker;
import com.example.objectpool.pool.WorkerPool;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/api/pool")
public class PoolController {

    private final WorkerPool pool;

    public PoolController(WorkerPool pool) {
        this.pool = pool;
    }

    @GetMapping("/acquire")
    public Map<String, Object> acquire() {
        Map<String, Object> response = new HashMap<>();
        try {
            ExpensiveWorker worker = pool.acquire();
            worker.doWork();
            response.put("message", "Worker acquired successfully");
        } catch (Exception e) {
            response.put("error", e.getMessage());
        }
        return response;
    }
}
```

```

        response.put("workerId", worker.getId());
    } catch (RuntimeException e) {
        response.put("error", e.getMessage());
    }
    return response;
}

@PostMapping("/release/{id}")
public Map<String, Object> release(@PathVariable long id) {
    pool.release(id);
    return Map.of("message", "Worker " + id + " released.");
}

@GetMapping("/status")
public Map<String, Object> status() {
    return Map.of(
        "availableCount", pool.getAvailableCount(),
        "inUseCount", pool.getInUseCount(),
        "availableIds", pool.getAvailableIds(),
        "inUseIds", pool.getInUseIds()
    );
}
}

```

5. Application Entry Point

```

package com.example.objectpool;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ObjectPoolApplication {

```

```
public static void main(String[] args) {  
    SpringApplication.run(ObjectPoolApplication.class, args);  
}  
}
```



Test with cURL or Postman

- **Acquire a worker:**

```
GET http://localhost:8080/api/pool/acquire
```

- **Release a worker:**

```
POST http://localhost:8080/api/pool/release/{id}
```

- **View pool status:**

```
GET http://localhost:8080/api/pool/status
```

6. Prototype Pattern



Real-World Analogy

Imagine you're designing a document editor where users can duplicate shapes (like a circle or rectangle) — instead of re-creating the shape every time, you can **clone** an existing shape.



Java Implementation of Prototype Pattern

1. Prototype Interface

```
public interface Prototype<T> {  
    T clone();  
}
```

2. Concrete Class (e.g., Shape)

```
public class Shape implements Prototype<Shape> {  
    private String type;  
    private int x;  
    private int y;  
  
    public Shape(String type, int x, int y) {  
        this.type = type;  
        this.x = x;  
        this.y = y;  
    }  
  
    @Override  
    public Shape clone() {  
        return new Shape(type, x, y); // deep copy  
    }  
  
    public void draw() {  
        System.out.println("Drawing " + type + " at (" + x + ", " + y + ")");  
    }  
  
    public void setPosition(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

3. Client Usage

```
public class PrototypeDemo {  
    public static void main(String[] args) {  
        Shape original = new Shape("Circle", 10, 20);  
        original.draw();  
  
        Shape clone = original.clone();  
        clone.setPosition(30, 40);  
        clone.draw();  
    }  
}
```

Output

```
Drawing Circle at (10, 20)  
Drawing Circle at (30, 40)
```

Optional: Clone Using **Cloneable** Interface

Java also has the **Cloneable** interface and **Object.clone()** method:

```
public class Shape implements Cloneable {  
    private String type;  
    private int x;  
  
    @Override  
    public Shape clone() {  
        try {  
            return (Shape) super.clone();  
        } catch (CloneNotSupportedException e) {  
            throw new AssertionError();  
        }  
    }  
}
```

```
    }  
}
```

⚠ Note: The `Cloneable` interface and `Object.clone()` have limitations (e.g., shallow copy), so many developers prefer implementing custom `clone()` logic.

✓ When to Use Prototype Pattern

- When object creation is expensive (e.g., from DB or over the network)
- When you want to avoid subclassing (unlike Factory pattern)
- When you need many copies of similar objects

```
package com.sai.designPatterns.prototype;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Vehicle implements Cloneable {  
  
    private List<String> vehicleList;  
  
    public Vehicle() {  
        this.vehicleList = new ArrayList<>();  
    }  
  
    public Vehicle(List<String> list) {  
        this.vehicleList = list;  
    }  
  
    public void insert() {  
        vehicleList.add("BMW");  
        vehicleList.add("Audi");  
    }  
}
```

```

public List<String> getVehicleList() {
    return this.vehicleList;
}

@Override
public Object clone() throws CloneNotSupportedException {
    List<String> temp = new ArrayList<>();
    for (String s : this.vehicleList) {
        temp.add(s);
    }
    return new Vehicle(temp);
}

}

```

```

package com.sai.designPatterns.prototype;

import com.sai.designPatterns.prototype.spring.AbstractVehicle;

public class PrototypePattern {

    public static void main(String[] args) throws CloneNotSupportedException {
        Vehicle v = new Vehicle();
        v.insert();
        Vehicle v1 = (Vehicle) v.clone();
        System.out.println(v1.getVehicleList());
    }

}

```

```

package com.sai.designPatterns.prototype.spring;

import lombok.AllArgsConstructor;

```

```

import lombok.Data;

import java.util.List;

@Data
@AllArgsConstructor
public class Car extends AbstractVehicle {

    private List<String> vehicleList;

    @Override
    public AbstractVehicle clone() {
        return new Car(vehicleList);
    }
}

```

```

package com.sai.designPatterns.prototype.spring;

import lombok.Data;

import java.util.List;

@Data
public abstract class AbstractVehicle {
    private List<String> vehicleList;
    protected abstract AbstractVehicle clone();
}

```

Shallow Copy vs Deep Copy

| Feature | Shallow Copy | Deep Copy |
|-------------------|--------------------------|---|
| Definition | Copies object references | Copies entire object + objects inside (recursively) |

| | | |
|-----------------|---|--|
| Memory | Original and copy share same nested object references | Original and copy are completely independent |
| Speed | Faster (less memory work) | Slower (more memory and recursion) |
| Use Case | When nested objects don't need to be duplicated | When full separation is needed |

Java Example

Let's say we have:

```
class Address {
    String city;

    Address(String city) {
        this.city = city;
    }
}

class Person {
    String name;
    Address address;

    Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
}
```

1. Shallow Copy

```
Person original = new Person("Sai", new Address("Bangalore"));
Person shallowCopy = new Person(original.name, original.address);
```

```
// Change in nested object affects both
shallowCopy.address.city = "Hyderabad";

System.out.println(original.address.city); // 👉 Hyderabad (changed!)
```

👉 `original` and `shallowCopy` share the same `Address` object.

2. Deep Copy

```
Person original = new Person("Sai", new Address("Bangalore"));

// Deep copy: manually clone nested object
Person deepCopy = new Person(original.name, new Address(original.address.
city));

deepCopy.address.city = "Hyderabad";

System.out.println(original.address.city); // 👉 Bangalore (unchanged)
```

👉 Now `deepCopy.address` is a **new object**, so changes don't affect the original.

Summary

| Scenario | Use Deep Copy? |
|--|----------------|
| Clone configs or DTOs safely | ✓ |
| Avoid memory overhead | ✗ |
| Copy primitives and shallow objects only | ✗ |
| Clone nested or mutable objects | ✓ |

Behavioural

1. Adapter

Use Case

Imagine you are integrating a third-party **Media Player** into your application, but your app expects a different interface. Instead of rewriting existing code, you write an adapter to connect both.

Structure

- **Target**: The interface your client expects.
- **Adaptee**: The existing interface you need to adapt.
- **Adapter**: The class that implements **Target** and wraps an instance of **Adaptee**.

Example in Java

1. Target Interface

```
public interface MediaPlayer {  
    void play(String audioType, String fileName);  
}
```

2. Adaptee Class

```
public class AdvancedMediaPlayer {  
    public void playVlc(String fileName) {  
        System.out.println("Playing VLC file: " + fileName);  
    }  
  
    public void playMp4(String fileName) {  
        System.out.println("Playing MP4 file: " + fileName);  
    }  
}
```

3. Adapter Class

```
public class MediaAdapter implements MediaPlayer {  
    private AdvancedMediaPlayer advancedMediaPlayer;  
  
    public MediaAdapter(String audioType) {  
        advancedMediaPlayer = new AdvancedMediaPlayer();  
    }  
  
    @Override  
    public void play(String audioType, String fileName) {  
        if ("vlc".equalsIgnoreCase(audioType)) {  
            advancedMediaPlayer.playVlc(fileName);  
        } else if ("mp4".equalsIgnoreCase(audioType)) {  
            advancedMediaPlayer.playMp4(fileName);  
        } else {  
            System.out.println("Invalid format: " + audioType);  
        }  
    }  
}
```

4. Client Class

```
public class AudioPlayer implements MediaPlayer {  
    private MediaAdapter mediaAdapter;  
  
    @Override  
    public void play(String audioType, String fileName) {  
        if ("mp3".equalsIgnoreCase(audioType)) {  
            System.out.println("Playing MP3 file: " + fileName);  
        } else if ("vlc".equalsIgnoreCase(audioType) || "mp4".equalsIgnoreCase(  
                audioType)) {  
            mediaAdapter = new MediaAdapter(audioType);  
            mediaAdapter.play(audioType, fileName);  
        } else {  
        }  
    }  
}
```

```

        System.out.println("Unsupported format: " + audioType);
    }
}
}

```

5. Main Method

```

public class AdapterPatternDemo {
    public static void main(String[] args) {
        MediaPlayer player = new AudioPlayer();
        player.play("mp3", "track.mp3");
        player.play("mp4", "movie.mp4");
        player.play("vlc", "video.vlc");
        player.play("avi", "clip.avi");
    }
}

```



Output

Playing MP3 file: track.mp3
 Playing MP4 file: movie.mp4
 Playing VLC file: video.vlc
 Unsupported format: avi

🎯 Use Cases

| Scenario | Description |
|----------------------|--|
| ✓ Legacy integration | When working with legacy or third-party code that has a different interface. |
| ✓ Interface mismatch | When two classes have incompatible interfaces, and you can't change their source code. |

| | |
|--|---|
| <input checked="" type="checkbox"/> UI libraries | Adapting custom UI components to fit third-party interfaces (e.g., adapting Swing components for a JavaFX app). |
| <input checked="" type="checkbox"/> API bridging | Integrating APIs with differing method signatures or formats. |

The **Adapter Design Pattern** is commonly used in **Spring Boot** projects when integrating external systems, legacy APIs, or adapting interfaces to fit domain models.

Real-world Example in Spring Boot

Let's say you have a Spring Boot app that consumes **payment services** from multiple vendors (e.g., **Stripe**, **Razorpay**), but you want to expose a **common internal interface** for your application to interact with any of them uniformly.

Step-by-Step Adapter Implementation in Spring Boot

1. Target Interface (Expected by your application)

```
public interface PaymentGateway {
    void pay(double amount);
}
```

2. Adaptee Classes (Vendor SDKs or APIs)

Stripe SDK Simulation

```
public class StripePaymentSDK {
    public void makeStripePayment(double amount) {
        System.out.println("Paid via Stripe: ₹" + amount);
    }
}
```

Razorpay SDK Simulation

```
public class RazorpaySDK {  
    public void doRazorTransaction(double amountInPaise) {  
        System.out.println("Paid via Razorpay: ₹" + amountInPaise / 100);  
    }  
}
```

3. Adapters for Each SDK

Stripe Adapter

```
@Component("stripeAdapter")  
public class StripeAdapter implements PaymentGateway {  
    private final StripePaymentSDK stripe = new StripePaymentSDK();  
  
    @Override  
    public void pay(double amount) {  
        stripe.makeStripePayment(amount);  
    }  
}
```

Razorpay Adapter

```
@Component("razorpayAdapter")  
public class RazorpayAdapter implements PaymentGateway {  
    private final RazorpaySDK razorpay = new RazorpaySDK();  
  
    @Override  
    public void pay(double amount) {  
        razorpay.doRazorTransaction(amount * 100); // Convert to paise  
    }  
}
```

4. Service That Uses Adapter

```

@Service
public class PaymentService {

    private final Map<String, PaymentGateway> gateways;

    @Autowired
    public PaymentService(Map<String, PaymentGateway> gateways) {
        this.gateways = gateways;
    }

    public void processPayment(String method, double amount) {
        PaymentGateway gateway = gateways.get(method);
        if (gateway != null) {
            gateway.pay(amount);
        } else {
            throw new IllegalArgumentException("Unsupported payment method: "
+ method);
        }
    }
}

```

@Component("stripeAdapter") and
@Component("razorpayAdapter") will inject the adapters using
a bean map.

5. Controller

```

@RestController
@RequestMapping("/api/payments")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;
}

```

```

    @PostMapping
    public ResponseEntity<String> makePayment(
        @RequestParam String method,
        @RequestParam double amount) {
        paymentService.processPayment(method, amount);
        return ResponseEntity.ok("Payment of ₹" + amount + " done via " + method);
    }
}

```

Summary

| Component | Purpose |
|---|--|
| <code>PaymentGateway</code> | Target interface used by Spring app |
| <code>StripePaymentSDK</code> , <code>RazorpaySDK</code> | Adaptees (external/legacy services) |
| <code>StripeAdapter</code> , <code>RazorpayAdapter</code> | Adapter classes that convert adaptee interface to target |
| <code>PaymentService</code> | Uses the unified <code>PaymentGateway</code> interface |
| <code>PaymentController</code> | Client that triggers the adapter-based logic |

Use Cases in Spring Boot

- Integrating **third-party SDKs or APIs**.
- Bridging **legacy services** with new Spring interfaces.
- Switching between **multiple implementations** (e.g., database, caching layers).
- Creating **testable** and **loosely coupled** service interfaces.

2.

The **Bridge Design Pattern** is a **structural pattern** used to **decouple abstraction from implementation**, so that the two can vary independently.



Real-World Motivation

Imagine you're developing a drawing app where you have:

- Different **shapes** (Circle, Square) — abstraction.
- Multiple **rendering engines** (vector, raster) — implementation.

Instead of creating a class for every combination (`VectorCircle`, `RasterCircle`, etc.), you separate shape logic and rendering logic using the Bridge pattern.



Structure

- **Abstraction**: High-level interface (e.g., `Shape`)
- **Refined Abstraction**: Extends abstraction (e.g., `Circle`)
- **Implementor**: Interface for implementation (e.g., `Renderer`)
- **Concrete Implementor**: Implements Implementor (e.g., `VectorRenderer`, `RasterRenderer`)

1. Implementor Interface

```
public interface Renderer {  
    void render(String shapeType);  
}
```

2. Concrete Implementors

```
public class VectorRenderer implements Renderer {  
    @Override  
    public void render(String shapeType) {  
        System.out.println("Drawing " + shapeType + " as vectors.");  
    }  
}
```

```
    }
}

public class RasterRenderer implements Renderer {
    @Override
    public void render(String shapeType) {
        System.out.println("Drawing " + shapeType + " as pixels.");
    }
}
```

3. Abstraction

```
public abstract class Shape {
    protected Renderer renderer;

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

    public abstract void draw();
}
```

4. Refined Abstractions

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

    @Override
    public void draw() {
        renderer.render("Circle");
    }
}
```

```

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

    @Override
    public void draw() {
        renderer.render("Square");
    }
}

```

5. Demo

```

public class BridgePatternDemo {
    public static void main(String[] args) {
        Renderer vector = new VectorRenderer();
        Renderer raster = new RasterRenderer();

        Shape circle = new Circle(vector);
        Shape square = new Square(raster);

        circle.draw(); // Output: Drawing Circle as vectors.
        square.draw(); // Output: Drawing Square as pixels.
    }
}

```

Use Cases of Bridge Pattern

| Use Case | Explanation |
|---|---|
| <input checked="" type="checkbox"/> UI frameworks | Separating platform-independent UI logic from platform-specific rendering (e.g., AWT vs Swing). |

| | |
|---|--|
| <input checked="" type="checkbox"/> Database drivers | JDBC provides an abstraction over different databases. The <code>DriverManager</code> bridges the JDBC API to specific DB drivers. |
| <input checked="" type="checkbox"/> File format exporters | A document can be exported as PDF, Word, HTML using different export strategies. |
| <input checked="" type="checkbox"/> Remote device control | A remote (abstraction) can operate TV, radio, projector (implementation). |
| <input checked="" type="checkbox"/> Messaging apps | Send messages (text, image, video) via various channels (SMS, Email, Push Notification). |
| <input checked="" type="checkbox"/> Game development | Game objects (e.g., units) can be rendered using different graphics engines (OpenGL, DirectX). |

🔑 When to Use

- You want to avoid a **combinatorial explosion** of classes.
- You want to **decouple abstraction from implementation** for better flexibility.
- You expect to change or extend **both abstraction and implementation** independently.
- **Adapter pattern:** You already have a `StripeSDK` and want to adapt it to your internal interface `PaymentProcessor`.
- **Bridge pattern:** You're designing the system from scratch to support multiple payment types (Stripe, Razorpay, PayPal) and multiple payment methods (card, wallet, UPI) with **separate hierarchies**.

Implementing the **Bridge Design Pattern in Spring Boot** is useful when you want to **decouple abstraction from implementation**, enabling them to vary independently. This is especially helpful in systems that require **flexible architecture**, such as notification systems, file exporters, or multi-database systems.

✓ Real-World Example: Notification System

We want to support **different notification types** (e.g., alert, marketing) and **different delivery channels** (e.g., email, SMS, push). Instead of creating a class for each combination (like `EmailAlertNotification`, `SMSMarketingNotification`, etc.), we'll **bridge** the abstraction (`Notification`) from its implementation (`MessageSender`).

Structure

- **Abstraction:** `Notification`
 - **Refined Abstraction:** `AlertNotification`, `MarketingNotification`
 - **Implementor Interface:** `MessageSender`
 - **Concrete Implementors:** `EmailSender`, `SMSender`
-

Implementation in Spring Boot

1. Implementor Interface

```
public interface MessageSender {  
    void sendMessage(String message);  
}
```

2. Concrete Implementors

```
@Component("emailSender")  
public class EmailSender implements MessageSender {  
    @Override  
    public void sendMessage(String message) {  
        System.out.println("Sending Email: " + message);  
    }  
}  
  
{@Component("smsSender")  
public class SMSender implements MessageSender {  
    @Override
```

```
public void sendMessage(String message) {  
    System.out.println("Sending SMS: " + message);  
}  
}
```

3. Abstraction

```
public abstract class Notification {  
  
    protected final MessageSender messageSender;  
  
    public Notification(MessageSender messageSender) {  
        this.messageSender = messageSender;  
    }  
  
    public abstract void notifyUser(String message);  
}
```

4. Refined Abstractions

```
public class AlertNotification extends Notification {  
    public AlertNotification(MessageSender messageSender) {  
        super(messageSender);  
    }  
  
    @Override  
    public void notifyUser(String message) {  
        messageSender.sendMessage("[ALERT] " + message);  
    }  
}  
  
public class MarketingNotification extends Notification {  
    public MarketingNotification(MessageSender messageSender) {  
        super(messageSender);  
    }
```

```

    }

    @Override
    public void notifyUser(String message) {
        messageSender.sendMessage("[MARKETING] " + message);
    }
}

```

5. Bridge Config via Spring Boot

```

@Service
public class NotificationService {

    private final Map<String, MessageSender> messageSenders;

    @Autowired
    public NotificationService(Map<String, MessageSender> messageSenders)
    {
        this.messageSenders = messageSenders;
    }

    public void sendNotification(String type, String channel, String message) {
        MessageSender sender = messageSenders.get(channel);
        if (sender == null) {
            throw new IllegalArgumentException("Unsupported channel: " + channel);
        }

        Notification notification;
        switch (type) {
            case "alert":
                notification = new AlertNotification(sender);
                break;
            case "marketing":
                notification = new MarketingNotification(sender);
                break;
        }
    }
}

```

```

        break;
    default:
        throw new IllegalArgumentException("Unsupported type: " + type);
    }

    notification.notifyUser(message);
}
}

```

6. REST Controller

```

@RestController
@RequestMapping("/notifications")
public class NotificationController {

    private final NotificationService notificationService;

    @Autowired
    public NotificationController(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    @PostMapping
    public ResponseEntity<String> send(
        @RequestParam String type,
        @RequestParam String channel,
        @RequestParam String message
    ) {
        notificationService.sendNotification(type, channel, message);
        return ResponseEntity.ok("Notification sent.");
    }
}

```

Sample Request

```
POST /notifications?type=alert&channel=emailSender&message=Disk+Usage+High
```

Use Cases of Bridge Pattern in Spring Boot

| Use Case | Description |
|--|---|
|  Notification systems | Decouple notification type from delivery mechanism |
|  Report generation | Report format (PDF, CSV) vs data source (SQL, NoSQL) |
|  Messaging systems | Message type (text, image) vs delivery channel (email, push, SMS) |
|  File storage | Storage type (public, private) vs provider (S3, local, Azure) |
|  Integration layers | Business logic vs external APIs (bridge logic & adapter style) |

3. Decorator (Wrapper)

The **Decorator Design Pattern** (also known as the **Wrapper Pattern**) is a structural pattern used to dynamically add behavior or responsibilities to an object without modifying its code. This is especially useful when you want to extend functionality in a **flexible and reusable way, without subclassing**.

Core Concepts

| Role | Description |
|---|--|
|  Component | The original interface or abstract class (e.g.,  Notifier) |
|  ConcreteComponent | The original implementation (e.g.,  EmailNotifier) |
|  Decorator | The base class for decorators that implements  Component |
|  ConcreteDecorator | Adds additional behavior (e.g.,  SMSDecorator ,  SlackDecorator) |

Java Example: Notification System with Decorators

We want to notify users via **email**, but also allow optional **SMS** and **Slack** notifications **without altering the original notifier**.

1. **Notifier** Interface (Component)

```
public interface Notifier {  
    void send(String message);  
}
```

2. **EmailNotifier** (ConcreteComponent)

```
public class EmailNotifier implements Notifier {  
    @Override  
    public void send(String message) {  
        System.out.println("Sending Email: " + message);  
    }  
}
```

3. **NotifierDecorator** (Base Decorator)

```
public abstract class NotifierDecorator implements Notifier {  
    protected Notifier wrappee;  
  
    public NotifierDecorator(Notifier notifier) {  
        this.wrappee = notifier;  
    }  
  
    @Override  
    public void send(String message) {  
        wrappee.send(message);  
    }  
}
```

```
    }  
}
```

4. **SMSDecorator** and **SlackDecorator** (Concrete Decorators)

```
public class SMSDecorator extends NotifierDecorator {  
    public SMSDecorator(Notifier notifier) {  
        super(notifier);  
    }  
  
    @Override  
    public void send(String message) {  
        super.send(message);  
        System.out.println("Sending SMS: " + message);  
    }  
}  
  
public class SlackDecorator extends NotifierDecorator {  
    public SlackDecorator(Notifier notifier) {  
        super(notifier);  
    }  
  
    @Override  
    public void send(String message) {  
        super.send(message);  
        System.out.println("Sending Slack Message: " + message);  
    }  
}
```

5. Demo Usage

```
public class Main {  
    public static void main(String[] args) {  
        Notifier basicNotifier = new EmailNotifier();
```

```

// Wrap email with SMS
Notifier smsNotifier = new SMSDecorator(basicNotifier);

// Wrap SMS+Email with Slack
Notifier fullNotifier = new SlackDecorator(smsNotifier);

fullNotifier.send("Server is down!");
}

}

```

Output

Sending Email: Server is down!
 Sending SMS: Server is down!
 Sending Slack Message: Server is down!



Real-World Use Cases of Decorator Pattern

| Use Case | Description |
|----------------------|---|
| Logging | Add logging around service methods |
| Caching | Wrap DAO or service methods with a caching layer |
| Validation | Decorate data processing pipelines |
| Spring Boot Filters | Servlet filters and interceptors act like decorators |
| Input/Output Streams | <code>BufferedInputStream</code> , <code>DataInputStream</code> in Java I/O |



Spring Boot Context Example (Optional)

In Spring, you can use the decorator pattern with beans using proxies. For example:

```

@Component
@Primary

```

```

public class LoggingUserService implements UserService {
    private final UserService delegate;

    public LoggingUserService(@Qualifier("actualUserService") UserService d
elegate) {
        this.delegate = delegate;
    }

    @Override
    public void createUser(String name) {
        System.out.println("Logging: Creating user " + name);
        delegate.createUser(name);
    }
}

```

Summary

- **Decorator = Wrapper + Behavior**
- Promotes **Open/Closed Principle**: open for extension, closed for modification.
- Great alternative to subclassing when multiple combinations of features are needed.

In **Spring Boot**, the **Decorator Design Pattern** is often used to enhance or modify the behavior of services **without changing the original implementation**. This is typically done using **delegation**, **bean overriding**, and **proxies**.

Real-World Example: Decorator Pattern in Spring Boot

Use Case: Logging + Notification Enhancer

We have a `NotificationService` that sends basic notifications. We want to **add features like logging and retry logic** without changing its original code.

1. Base Interface

```
public interface NotificationService {  
    void send(String message);  
}
```

2. Core Implementation

```
@Component("baseNotificationService")  
public class EmailNotificationService implements NotificationService {  
    @Override  
    public void send(String message) {  
        System.out.println("✉️ Sending Email: " + message);  
    }  
}
```

3. Logging Decorator

```
@Component  
@Primary // This will override the base implementation  
public class LoggingNotificationDecorator implements NotificationService {  
  
    private final NotificationService delegate;  
  
    public LoggingNotificationDecorator(@Qualifier("baseNotificationService")  
                                         NotificationService delegate) {  
        this.delegate = delegate;  
    }  
  
    @Override  
    public void send(String message) {  
        System.out.println("📝 Logging: About to send message");  
        delegate.send(message);  
        System.out.println("📝 Logging: Message sent");  
    }  
}
```

```
    }  
}
```

4. Retry Decorator (Optional, Stackable)

```
@Component("retryingNotificationService")  
public class RetryNotificationDecorator implements NotificationService {  
  
    private final NotificationService delegate;  
  
    public RetryNotificationDecorator(@Qualifier("loggingNotificationDecorato  
r") NotificationService delegate) {  
        this.delegate = delegate;  
    }  
  
    @Override  
    public void send(String message) {  
        int attempts = 0;  
        boolean success = false;  
  
        while (!success && attempts < 3) {  
            try {  
                delegate.send(message);  
                success = true;  
            } catch (Exception e) {  
                attempts++;  
                System.out.println("⚠ Retry attempt " + attempts);  
            }  
        }  
  
        if (!success) {  
            System.out.println("✗ Failed to send message after 3 attempts");  
        }  
    }  
}
```

```
    }  
}
```

You can wire this retry decorator manually into a config or use it directly.

5. Using the Decorated Bean

```
@RestController  
@RequiredArgsConstructor  
public class NotificationController {  
  
    private final NotificationService notificationService; // Spring injects the @Primary  
  
    @PostMapping("/send")  
    public ResponseEntity<String> sendNotification(@RequestParam String message) {  
        notificationService.send(message);  
        return ResponseEntity.ok("Notification processed");  
    }  
}
```

Notes

| Concept | Usage |
|------------------------------|--|
| <code>@Primary</code> | Tells Spring to inject this bean when multiple beans implement an interface. |
| <code>@Qualifier</code> | Lets you refer to a specific bean when injecting dependencies. |
| <code>@Component</code> | Each decorator is a component and wired like a chain. |
| <code>Bean Overriding</code> | You can create decorator chains via <code>@Configuration</code> too. |

Benefits

- Follows **Open/Closed Principle** (no changes to the core class).

- Easy to **add or remove behavior** like logging, caching, retry, etc.
- Works beautifully with **Spring DI and bean lifecycle**.



Alternative: Decorator via Configuration

Instead of relying on `@Component`, you can configure your decorator chain:

```
@Configuration
public class NotificationConfig {

    @Bean
    public NotificationService notificationService() {
        return new LoggingNotificationDecorator(
            new RetryNotificationDecorator(
                new EmailNotificationService()
            )
        );
    }
}
```



Common Spring Boot Use Cases

| Decorator Use Case | Example |
|--------------------|--|
| Logging | Log service method calls |
| Metrics | Time a method and log to Prometheus |
| Security | Add access control before calling real service |
| Retry | Retry failed remote calls |
| Caching | Cache method responses |
| Feature toggles | Dynamically switch behavior using flags |

4. `Composite`

The **Composite Design Pattern** is a structural pattern used to treat **individual objects** and **compositions of objects** uniformly. It allows you to build tree-like structures where nodes can be either **leaf** (basic objects) or **composites** (containers of other objects).

Key Intent

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly."

Real-World Analogy

Think of a **file system**:

- A `File` is a leaf node.
 - A `Directory` can contain both files and subdirectories (which can contain more).
-

Java Implementation: File System Example

1. Component Interface

```
public interface FileSystemComponent {  
    void showDetails(String indent);  
}
```

2. Leaf: File

```
public class FileLeaf implements FileSystemComponent {  
    private final String name;  
  
    public FileLeaf(String name) {  
        this.name = name;
```

```
}

@Override
public void showDetails(String indent) {
    System.out.println(indent + "- File: " + name);
}
}
```

3. Composite: Directory

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

public class DirectoryComposite implements FileSystemComponent {
    private final String name;
    private final List<FileSystemComponent> children = new ArrayList<>();

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

    public void add(FileSystemComponent component) {
        children.add(component);
    }

    public void remove(FileSystemComponent component) {
        children.remove(component);
    }

    @Override
    public void showDetails(String indent) {
        System.out.println(indent + "+ Directory: " + name);
        for (FileSystemComponent component : children) {
            component.showDetails(indent + "  ");
        }
    }
}
```

```

    }
}

package com.sai.designPatterns.composite;

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

public class CompositeAccount extends Account {
    private float totalBalance;
    private List<Account> accountList = new ArrayList<Account>();

    public float getBalance() {
        totalBalance = 0;
        for (Account f : accountList) {
            totalBalance = totalBalance + f.getBalance();
        }
        return totalBalance;
    }

    public void addAccount(Account acc) {
        accountList.add(acc);
    }

    public void removeAccount(Account acc) {
        accountList.remove(acc);
    }
}

```

4. Usage

```

public class Main {
    public static void main(String[] args) {
        FileSystemComponent file1 = new FileLeaf("resume.pdf");

```

```

FileSystemComponent file2 = new FileLeaf("todo.txt");
FileSystemComponent file3 = new FileLeaf("budget.xlsx");

DirectoryComposite documents = new DirectoryComposite("Documents");
documents.add(file1);
documents.add(file2);

DirectoryComposite work = new DirectoryComposite("Work");
work.add(file3);
work.add(documents);

work.showDetails(""); // start with empty indent
}
}

```

Output

- + Directory: Work
 - File: budget.xlsx
- + Directory: Documents
 - File: resume.pdf
 - File: todo.txt

Use Cases of Composite Pattern

| Domain | Use Case |
|------------------|--|
| UI Toolkits | Components like buttons, panels, and windows treated uniformly |
| File Systems | Files and directories |
| XML/HTML Parsing | Elements and nested elements |
| Drawing Apps | Shapes vs Groups of Shapes |
| Menus | MenuItem and Submenu hierarchy |

| | |
|-----------------|--|
| Rule Engines | Nested logical rules (AND/OR) |
| Product Bundles | Products and product bundles (e.g., in e-commerce) |

Benefits

- Treat **individual and composite objects uniformly**.
- Supports recursive structures naturally.
- Makes adding new components easy.

Drawbacks

- Can make the design overly general.
- Type-checking for leaves vs. composites might be necessary in stricter environments.

Use Case Example: Organization Structure (Employees and Departments)

- A `Department` can contain sub-departments or employees.
- `Employee` is a leaf node.
- `Department` is a composite.

This is a textbook example of the Composite Design Pattern.

Final Structure

```
composite-pattern-springboot/
|
|   -- controller/
|       |   -- OrgController.java
|   -- model/
|       |   -- OrgComponent.java
```

```
|   └── Employee.java  
|   └── Department.java  
└── service/  
    └── OrgService.java  
└── CompositePatternApplication.java
```

Step-by-Step Guide



OrgComponent.java (Component)

```
package com.example.model;  
  
public abstract class OrgComponent {  
    protected String name;  
  
    public OrgComponent(String name) {  
        this.name = name;  
    }  
  
    public abstract void showDetails();  
}
```



Employee.java (Leaf)

```
package com.example.model;  
  
public class Employee extends OrgComponent {  
  
    private String role;  
  
    public Employee(String name, String role) {  
        super(name);  
        this.role = role;  
    }
```

```
}

@Override
public void showDetails() {
    System.out.println("Employee: " + name + ", Role: " + role);
}
}
```



Department.java (Composite)

```
package com.example.model;

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

public class Department extends OrgComponent {

    private List<OrgComponent> components = new ArrayList<>();

    public Department(String name) {
        super(name);
    }

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

    public void removeComponent(OrgComponent component) {
        components.remove(component);
    }

    @Override
    public void showDetails() {
        System.out.println("Department: " + name);
        for (OrgComponent component : components) {
```

```
        component.showDetails();
    }
}
}
```



OrgService.java

```
package com.example.service;

import com.example.model.Department;
import com.example.model.Employee;
import com.example.model.OrgComponent;
import org.springframework.stereotype.Service;

@Service
public class OrgService {

    public OrgComponent createOrgStructure() {
        Employee emp1 = new Employee("Alice", "Developer");
        Employee emp2 = new Employee("Bob", "Tester");

        Department devDept = new Department("Development");
        devDept.addComponent(emp1);
        devDept.addComponent(emp2);

        Employee emp3 = new Employee("Carol", "HR");

        Department hrDept = new Department("HR");
        hrDept.addComponent(emp3);

        Department headOffice = new Department("Head Office");
        headOffice.addComponent(devDept);
        headOffice.addComponent(hrDept);

        return headOffice;
    }
}
```

```
    }  
}
```



OrgController.java

```
package com.example.controller;  
  
import com.example.model.OrgComponent;  
import com.example.service.OrgService;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class OrgController {  
  
    private final OrgService orgService;  
  
    public OrgController(OrgService orgService) {  
        this.orgService = orgService;  
    }  
  
    @GetMapping("/org")  
    public String getOrgStructure() {  
        OrgComponent root = orgService.createOrgStructure();  
        root.showDetails(); // You could alternatively return a custom DTO  
        return "Organization structure printed in logs.";  
    }  
}
```



CompositePatternApplication.java

```
package com.example;  
  
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CompositePatternApplication {
    public static void main(String[] args) {
        SpringApplication.run(CompositePatternApplication.class, args);
    }
}
```

✓ Output on calling `/org` endpoint:

```
Department: Head Office
Department: Development
Employee: Alice, Role: Developer
Employee: Bob, Role: Tester
Department: HR
Employee: Carol, Role: HR
```

🧠 Summary

- **Pattern Used:** Composite Pattern
- **Leaf:** Employee
- **Composite:** Department
- **Client:** OrgService and OrgController
- **Framework:** Spring Boot

5. Filter (Criteria)

The **Filter (Criteria) Design Pattern**, a structural pattern (often treated as behavioral as well), allows you to filter a set of objects using different criteria and chaining them using logical operations such as AND, OR, NOT.

Step-by-Step Implementation in Java:

1. Create a **Person** class

```
public class Person {  
    private String name;  
    private String gender;  
    private String maritalStatus;  
  
    public Person(String name, String gender, String maritalStatus) {  
        this.name = name;  
        this.gender = gender;  
        this.maritalStatus = maritalStatus;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getGender() {  
        return gender;  
    }  
  
    public String getMaritalStatus() {  
        return maritalStatus;  
    }  
}
```

2. Create a **Criteria** interface

```
import java.util.List;  
  
public interface Criteria {
```

```
    List<Person> meetCriteria(List<Person> persons);
}
```

3. Implement concrete criteria classes

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

public class CriteriaMale implements Criteria {
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> malePersons = new ArrayList<>();
        for (Person person : persons) {
            if (person.getGender().equalsIgnoreCase("MALE")) {
                malePersons.add(person);
            }
        }
        return malePersons;
    }
}

public class CriteriaFemale implements Criteria {
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> femalePersons = new ArrayList<>();
        for (Person person : persons) {
            if (person.getGender().equalsIgnoreCase("FEMALE")) {
                femalePersons.add(person);
            }
        }
        return femalePersons;
    }
}

public class CriteriaSingle implements Criteria {
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> singlePersons = new ArrayList<>();
```

```

        for (Person person : persons) {
            if (person.getMaritalStatus().equalsIgnoreCase("SINGLE")) {
                singlePersons.add(person);
            }
        }
        return singlePersons;
    }
}

```

4. Create combinational criteria classes

```

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

public class AndCriteria implements Criteria {
    private Criteria criteria;
    private Criteria otherCriteria;

    public AndCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
        this.otherCriteria = otherCriteria;
    }

    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
        return otherCriteria.meetCriteria(firstCriteriaPersons);
    }
}

public class OrCriteria implements Criteria {
    private Criteria criteria;
    private Criteria otherCriteria;

    public OrCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
    }
}

```

```

        this.otherCriteria = otherCriteria;
    }

public List<Person> meetCriteria(List<Person> persons) {
    List<Person> firstList = criteria.meetCriteria(persons);
    List<Person> secondList = otherCriteria.meetCriteria(persons);

    for (Person person : secondList) {
        if (!firstList.contains(person)) {
            firstList.add(person);
        }
    }
    return firstList;
}
}

```

5. Test the Filter Pattern

```

import java.util.Arrays;
import java.util.List;

public class FilterPatternDemo {
    public static void main(String[] args) {
        List<Person> persons = Arrays.asList(
            new Person("Robert", "Male", "Single"),
            new Person("John", "Male", "Married"),
            new Person("Laura", "Female", "Married"),
            new Person("Diana", "Female", "Single"),
            new Person("Mike", "Male", "Single"),
            new Person("Bobby", "Male", "Single")
        );
    }
}

```

```

Criteria male = new CriteriaMale();
Criteria female = new CriteriaFemale();
Criteria single = new CriteriaSingle();

```

```

Criteria singleMale = new AndCriteria(single, male);
Criteria singleOrFemale = new OrCriteria(single, female);

System.out.println("Males:");
printPersons(male.meetCriteria(persons));

System.out.println("\nSingle Males:");
printPersons(singleMale.meetCriteria(persons));

System.out.println("\nSingle Or Females:");
printPersons(singleOrFemale.meetCriteria(persons));
}

public static void printPersons(List<Person> persons) {
    for (Person person : persons) {
        System.out.println("Person: [ Name: " + person.getName()
            + ", Gender: " + person.getGender()
            + ", Marital Status: " + person.getMaritalStatus() + "]");
    }
}
}

```

Output

Males:

Person: [Name: Robert, Gender: Male, Marital Status: Single]
 Person: [Name: John, Gender: Male, Marital Status: Married]
 Person: [Name: Mike, Gender: Male, Marital Status: Single]
 Person: [Name: Bobby, Gender: Male, Marital Status: Single]

Single Males:

Person: [Name: Robert, Gender: Male, Marital Status: Single]
 Person: [Name: Mike, Gender: Male, Marital Status: Single]
 Person: [Name: Bobby, Gender: Male, Marital Status: Single]

Single Or Females:

```
Person: [ Name: Robert, Gender: Male, Marital Status: Single ]  
Person: [ Name: Laura, Gender: Female, Marital Status: Married ]  
Person: [ Name: Diana, Gender: Female, Marital Status: Single ]  
Person: [ Name: Mike, Gender: Male, Marital Status: Single ]  
Person: [ Name: Bobby, Gender: Male, Marital Status: Single ]
```

6. Proxy

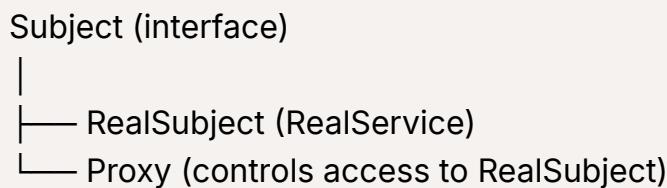
The **Proxy Design Pattern** provides a surrogate or placeholder for another object to control access to it. In Java, this is often used for:

- **Access control** (e.g. protecting real objects)
- **Lazy initialization** (e.g. expensive object creation)
- **Remote proxies** (e.g. RMI)
- **Logging / auditing / caching**

Real-world Analogy

Think of a **bank ATM** as a proxy to your **bank account** — the ATM controls access and enforces authentication before you interact with your actual account.

Structure



Example: Internet Access Proxy

1. Internet.java — Subject Interface

```
interface Internet {  
    void connectTo(String serverHost) throws Exception;  
}
```

2. **ReallInternet.java** — Real Object

```
public class ReallInternet implements Internet {  
    @Override  
    public void connectTo(String serverHost) {  
        System.out.println("Connecting to " + serverHost);  
    }  
}
```

3. **ProxyInternet.java** — Proxy Implementation

```
import java.util.Arrays;  
import java.util.List;  
  
public class ProxyInternet implements Internet {  
  
    private Internet reallInternet = new ReallInternet();  
    private static final List<String> bannedSites = Arrays.asList(  
        "facebook.com", "instagram.com", "tiktok.com"  
    );  
  
    @Override  
    public void connectTo(String serverHost) throws Exception {  
        if (bannedSites.contains(serverHost.toLowerCase())) {  
            throw new Exception("Access Denied to " + serverHost);  
        }  
        reallInternet.connectTo(serverHost);  
    }  
}
```

```
    }  
}
```

4. **ProxyPatternDemo.java** — Test Class

```
public class ProxyPatternDemo {  
    public static void main(String[] args) {  
        Internet internet = new ProxyInternet();  
  
        try {  
            internet.connectTo("google.com");  
            internet.connectTo("facebook.com");  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Output

```
Connecting to google.com  
Access Denied to facebook.com
```

🔍 When to Use

| Scenario | Proxy Type |
|-------------------------|-------------------------|
| Restrict access | Protection Proxy |
| Lazy load heavy objects | Virtual Proxy |
| Add logs or cache | Logging / Caching Proxy |
| Connect remote service | Remote Proxy |

Tip

Java's built-in `java.lang.reflect.Proxy` and Spring AOP (like `@Transactional`) also use **dynamic proxies**.

Use Case: Logging Access to a Service

We'll:

- Create a `VideoService` that streams videos.
- Wrap it with a `LoggingProxyVideoService` that logs each call.
- Use Spring to inject the proxy instead of the real service.

Project Structure

```
proxy-pattern-springboot/
├── ProxyPatternApplication.java
└── service/
    ├── VideoService.java      (interface)
    ├── RealVideoService.java  (actual service)
    └── LoggingProxyVideoService.java (proxy)
└── controller/
    └── VideoController.java
```

1. `VideoService.java` (Interface)

```
package com.example.service;

public interface VideoService {
    String streamVideo(String title);
}
```

2. RealVideoService.java (Actual Service)

```
package com.example.service;

import org.springframework.stereotype.Service;

@Service("realVideoService")
public class RealVideoService implements VideoService {
    @Override
    public String streamVideo(String title) {
        return "Streaming video: " + title;
    }
}
```

3. LoggingProxyVideoService.java (Proxy)

```
package com.example.service;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

@Service
public class LoggingProxyVideoService implements VideoService {

    private final VideoService realService;

    public LoggingProxyVideoService(@Qualifier("realVideoService") VideoService
        realService) {
        this.realService = realService;
    }

    @Override
    public String streamVideo(String title) {
```

```
        System.out.println("[LOG] Requesting video: " + title);
        String result = realService.streamVideo(title);
        System.out.println("[LOG] Finished streaming: " + title);
        return result;
    }
}
```

4. VideoController.java

```
package com.example.controller;

import com.example.service.VideoService;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/video")
public class VideoController {

    private final VideoService videoService;

    public VideoController(VideoService videoService) {
        this.videoService = videoService;
    }

    @GetMapping("/watch")
    public String watchVideo(@RequestParam String title) {
        return videoService.streamVideo(title);
    }
}
```

5. ProxyPatternApplication.java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProxyPatternApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProxyPatternApplication.class, args);
    }
}
```

Run and Test

```
GET http://localhost:8080/video/watch?title=Inception
```

Terminal Log:

```
[LOG] Requesting video: Inception
[LOG] Finished streaming: Inception
```

API Response:

```
Streaming video: Inception
```

Notes

- We're **injecting the proxy** as the main bean (`LoggingProxyVideoService`) using Spring's `@Qualifier` and constructor injection.
- You can enhance the proxy to add **auth, rate-limiting, caching, or circuit breaking**.

Alternatives in Spring

| Feature | Proxy Tool |
|------------------------|---|
| AOP Logging | <code>@Aspect</code> with <code>@Around</code> advice |
| Declarative Proxy | Spring <code>@Bean</code> method returning a proxy |
| Circuit Breaking | <code>Resilience4j</code> , <code>Hystrix</code> |
| Declarative REST Proxy | <code>FeignClient</code> |

Use Case: Logging Access to a Service

We'll:

- Create a `VideoService` that streams videos.
- Wrap it with a `LoggingProxyVideoService` that logs each call.
- Use Spring to inject the proxy instead of the real service.

Project Structure

```
proxy-pattern-springboot/
├── ProxyPatternApplication.java
└── service/
    ├── VideoService.java      (interface)
    ├── RealVideoService.java  (actual service)
    └── LoggingProxyVideoService.java (proxy)
└── controller/
    └── VideoController.java
```

1. `VideoService.java` (Interface)

```
package com.example.service;

public interface VideoService {
```

```
    String streamVideo(String title);  
}
```

2. RealVideoService.java (Actual Service)

```
java  
CopyEdit  
package com.example.service;  
  
import org.springframework.stereotype.Service;  
  
@Service("realVideoService")  
public class RealVideoService implements VideoService {  
    @Override  
    public String streamVideo(String title) {  
        return "Streaming video: " + title;  
    }  
}
```

3. LoggingProxyVideoService.java (Proxy)

```
java  
CopyEdit  
package com.example.service;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class LoggingProxyVideoService implements VideoService {  
  
    private final VideoService realService;
```

```
public LoggingProxyVideoService(@Qualifier("realVideoService") VideoService realService) {
    this.realService = realService;
}

@Override
public String streamVideo(String title) {
    System.out.println("[LOG] Requesting video: " + title);
    String result = realService.streamVideo(title);
    System.out.println("[LOG] Finished streaming: " + title);
    return result;
}
```

4. VideoController.java

```
java
CopyEdit
package com.example.controller;

import com.example.service.VideoService;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/video")
public class VideoController {

    private final VideoService videoService;

    public VideoController(VideoService videoService) {
        this.videoService = videoService;
    }
}
```

```
@GetMapping("/watch")
public String watchVideo(@RequestParam String title) {
    return videoService.streamVideo(title);
}
```

5. ProxyPatternApplication.java

```
java
CopyEdit
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProxyPatternApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProxyPatternApplication.class, args);
    }
}
```

✓ Run and Test

```
bash
CopyEdit
GET http://localhost:8080/video/watch?title=Inception
```

Terminal Log:

```
bash
CopyEdit
[LOG] Requesting video: Inception
[LOG] Finished streaming: Inception
```

API Response:

```
CSS
CopyEdit
Streaming video: Inception
```

🧠 Notes

- We're **injecting the proxy** as the main bean (`LoggingProxyVideoService`) using Spring's `@Qualifier` and constructor injection.
- You can enhance the proxy to add **auth, rate-limiting, caching, or circuit breaking**.

💡 Alternatives in Spring

| Feature | Proxy Tool |
|------------------------|---|
| AOP Logging | <code>@Aspect</code> with <code>@Around</code> advice |
| Declarative Proxy | Spring <code>@Bean</code> method returning a proxy |
| Circuit Breaking | <code>Resilience4j</code> , <code>Hystrix</code> |
| Declarative REST Proxy | <code>FeignClient</code> |

Would you like to see an **AOP-based proxy** as well (even more idiomatic in Spring)?

You said:

proxy design pattern using spring

ChatGPT said:

Here's how to implement the **Proxy Design Pattern using Spring Framework** (without Spring Boot if preferred, or within Boot if needed), to **intercept and control access** to a real service.

We'll show two ways:

1. **Manual Proxy with `@Component` and `@Qualifier`** (Classic proxy pattern).
2. **Spring AOP-based Proxy** (cleaner and idiomatic Spring way).

🔧 Use Case Example: Access Control to a Service

✓ Approach 1: Manual Proxy via `@Component`

📁 Structure

```
php
CopyEdit
com.example.proxy
└── Internet.java      (interface)
└── RealInternet.java  (real object)
└── ProxyInternet.java (proxy object)
└── AppConfig.java     (Spring config)
```

1. `Internet.java` (Subject interface)

```
java
CopyEdit
public interface Internet {
    void connectTo(String url) throws Exception;
```

```
}
```

2. **ReallInternet.java** (Real service)

```
java
CopyEdit
import org.springframework.stereotype.Component;

@Component("reallInternet")
public class ReallInternet implements Internet {
    @Override
    public void connectTo(String url) {
        System.out.println("Connected to " + url);
    }
}
```

3. **ProxyInternet.java** (Proxy class)

```
java
CopyEdit
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class ProxyInternet implements Internet {

    private final Internet reallInternet;
    private final List<String> bannedSites = List.of("facebook.com", "tiktok.co
m");
}
```

```

public ProxyInternet(@Qualifier("realInternet") Internet realInternet) {
    this.realInternet = realInternet;
}

@Override
public void connectTo(String url) throws Exception {
    if (bannedSites.contains(url.toLowerCase())) {
        throw new Exception("Access Denied to " + url);
    }
    realInternet.connectTo(url);
}
}

```

4. **AppConfig.java (Spring app config for testing)**

```

java
CopyEdit
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.example.proxy")
public class AppConfig {
    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(AppConfig.class);
        Internet internet = context.getBean(ProxyInternet.class);

        try {
            internet.connectTo("google.com");
            internet.connectTo("facebook.com");
        } catch (Exception e) {

```

```
        System.out.println(e.getMessage());
    }

    context.close();
}
}
```

✓ Output

```
bash
CopyEdit
Connected to google.com
Access Denied to facebook.com
```

✓ Approach 2: Spring AOP-Based Proxy (Preferred in large apps)

🔍 Use Case: Log before and after a method runs.

1. Add Spring AOP Dependency

If you're using Maven:

```
xml
CopyEdit
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

2. Define a Service

```
java
CopyEdit
@Service
public class VideoService {
    public void stream(String title) {
        System.out.println("Streaming video: " + title);
    }
}
```

3. Define an Aspect (the proxy behavior)

```
java
CopyEdit
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.VideoService.stream(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[LOG] Starting: " + joinPoint.getSignature().getName
());
    }

    @After("execution(* com.example.service.VideoService.stream(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("[LOG] Finished: " + joinPoint.getSignature().getName
());
    }
}
```

4. Call it from your main or controller

```
java
CopyEdit
@RestController
public class VideoController {
    private final VideoService videoService;

    public VideoController(VideoService videoService) {
        this.videoService = videoService;
    }

    @GetMapping("/watch")
    public String watch(@RequestParam String title) {
        videoService.stream(title);
        return "OK";
    }
}
```

✓ Output

```
less
CopyEdit
[LOG] Starting: stream
Streaming video: Avatar
[LOG] Finished: stream
```

⬅ END Summary

| Approach | Use Case |
|--------------|--|
| Manual Proxy | Full control, low-level logic |
| Spring AOP | Cross-cutting concerns (log, auth, perf) |

9. Flyweight

The **Flyweight Design Pattern** is a **structural pattern** used to **minimize memory usage** or **computational expenses** by sharing as much data as possible with similar objects.

Intent

Use sharing to support a large number of fine-grained objects efficiently.

When to Use

- You have **many objects** that consume **a lot of memory**.
 - Most object data is **intrinsic (shared)** and only a small part is **extrinsic (context-specific)**.
 - You want to **reuse** objects instead of creating new ones each time.
-

Key Concepts

- **Flyweight**: The shared object.
 - **Intrinsic State**: Shared and independent of context.
 - **Extrinsic State**: Supplied by client, depends on context.
 - **FlyweightFactory**: Manages flyweight instances.
-

Real-World Use Cases

| Use Case | Description |
|----------|-------------|
| | |

| | |
|-----------------------------------|--|
| Text Editor (characters) | Characters share font info (intrinsic), but position is extrinsic. |
| Game Trees (chess, go) | Reuse similar game pieces instead of creating new ones. |
| Icons in UI | One icon object rendered in many places with different positions. |
| Particle systems | Reuse particle shape, size etc., change position/color only. |
| Map rendering (e.g., Google Maps) | Reuse marker shapes/icons across locations. |

Java Implementation

Scenario: Drawing multiple circles with only a few shared colors.

1. **Shape.java (Flyweight Interface)**

```
public interface Shape {
    void draw(int x, int y);
}
```

2. **Circle.java (Concrete Flyweight)**

```
public class Circle implements Shape {
    private final String color; // Intrinsic (shared)
    private int x;             // Extrinsic
    private int y;             // Extrinsic

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

    @Override
    public void draw(int x, int y) {
```

```
        this.x = x;
        this.y = y;
        System.out.printf("Drawing %s circle at (%d, %d)%n", color, x, y);
    }
}
```

3. **ShapeFactory.java** (Flyweight Factory)

```
import java.util.HashMap;
import java.util.Map;

public class ShapeFactory {
    private static final Map<String, Shape> circleMap = new HashMap<>();

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

        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Created circle of color: " + color);
        }
        return circle;
    }
}
```

4. **FlyweightDemo.java** (Client Code)

```
public class FlyweightDemo {
    public static void main(String[] args) {
        String[] colors = { "Red", "Green", "Blue" };

        for (int i = 0; i < 10; i++) {
            String color = colors[i % colors.length];
```

```

        Shape circle = ShapeFactory.getCircle(color);
        circle.draw((int)(Math.random() * 100), (int)(Math.random() * 100));
    }
}
}

```

Output Example

Created circle of color: Red
 Drawing Red circle at (22, 11)
 Created circle of color: Green
 Drawing Green circle at (55, 91)
 Created circle of color: Blue
 Drawing Blue circle at (72, 34)
 Drawing Red circle at (99, 67)
 ...

Summary

| Element | Role in Flyweight |
|--------------|---------------------------|
| Shape | Flyweight interface |
| Circle | Concrete Flyweight |
| ShapeFactory | Creates/reuses Flyweights |
| x, y | Extrinsic data |
| color | Intrinsic (shared) data |

Benefits

-  Reduces memory usage
-  Improves performance for large datasets
-  Encourages sharing and reuse

Goal

We'll simulate a **UI icon system** where the icon's **shape and type** are intrinsic (shared) and **position** is extrinsic (context-specific). We'll use Spring to manage and share the flyweight beans.

Project Structure

```
com.example.flyweight
├── Icon.java          // Flyweight interface
├── SharedIcon.java    // Concrete Flyweight
├── IconFactory.java   // Flyweight factory using Spring
├── IconUsageService.java // Client
└── FlyweightApp.java   // Main class (SpringBootApplication)
```

Step-by-Step Spring Implementation

1. **Icon.java** – Flyweight interface

```
public interface Icon {
    void draw(int x, int y);
}
```

2. **SharedIcon.java** – Concrete Flyweight

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype") // Ensure different beans for different icons, if needed
public class SharedIcon implements Icon {
```

```

private String type; // Intrinsic state

public void setType(String type) {
    this.type = type;
}

@Override
public void draw(int x, int y) {
    System.out.printf("Drawing '%s' icon at (%d, %d)%n", type, x, y);
}
}

```

3. **IconFactory.java** – Factory using Spring Context

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.Map;

@Component
public class IconFactory {

    @Autowired
    private ApplicationContext context;

    private final Map<String, Icon> cache = new HashMap<>();

    public Icon getIcon(String type) {
        if (!cache.containsKey(type)) {
            SharedIcon icon = context.getBean(SharedIcon.class);
            icon.setType(type);
            cache.put(type, icon);
        }
        return cache.get(type);
    }
}

```

```
        System.out.println("Created new icon of type: " + type);
    }
    return cache.get(type);
}
}
```

4. **IconUsageService.java** – Client using Flyweights

```
java
CopyEdit
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class IconUsageService {

    @Autowired
    private IconFactory iconFactory;

    public void drawUI() {
        String[] types = {"folder", "file", "trash"};

        for (int i = 0; i < 10; i++) {
            String type = types[i % types.length];
            Icon icon = iconFactory.getIcon(type);
            icon.draw(i * 10, i * 5); // x, y are extrinsic
        }
    }
}
```

5. **FlyweightApp.java** – Main Spring Boot App

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class FlyweightApp implements CommandLineRunner {

    private final IIconUsageService usageService;

    public FlyweightApp(IIconUsageService usageService) {
        this.usageService = usageService;
    }

    public static void main(String[] args) {
        SpringApplication.run(FlyweightApp.class, args);
    }

    @Override
    public void run(String... args) {
        usageService.drawUI();
    }
}
```

✓ Output

```
Created new icon of type: folder
Drawing 'folder' icon at (0, 0)
Created new icon of type: file
Drawing 'file' icon at (10, 5)
Created new icon of type: trash
Drawing 'trash' icon at (20, 10)
Drawing 'folder' icon at (30, 15)
Drawing 'file' icon at (40, 20)
...
...
```



Summary

| Component | Role |
|------------------|-------------------------------|
| Icon | Flyweight interface |
| SharedIcon | Intrinsic state (shared icon) |
| IconFactory | Flyweight manager |
| IconUsageService | Client using icons |
| Spring Context | Manages bean creation |

✓ Benefits of Using Spring

- Easy caching & management
- Leverage dependency injection
- Cleaner separation of concerns
- Prototype scope for fine-grained control

✓ Goal

We'll simulate a **UI icon system** where the icon's **shape and type** are intrinsic (shared) and **position** is extrinsic (context-specific). We'll use Spring to manage and share the flyweight beans.

Project Structure

```
com.example.flyweight
├── Icon.java          // Flyweight interface
├── SharedIcon.java    // Concrete Flyweight
├── IconFactory.java   // Flyweight factory using Spring
├── IconUsageService.java // Client
└── FlyweightApp.java   // Main class (SpringBootApplication)
```



Step-by-Step Spring Implementation

1. `Icon.java` – Flyweight interface

```
public interface Icon {  
    void draw(int x, int y);  
}
```

2. `SharedIcon.java` – Concrete Flyweight

```
import org.springframework.context.annotation.Scope;  
import org.springframework.stereotype.Component;  
  
@Component  
@Scope("prototype") // Ensure different beans for different icons, if needed  
public class SharedIcon implements Icon {  
  
    private String type; // Intrinsic state  
  
    public void setType(String type) {  
        this.type = type;  
    }  
  
    @Override  
    public void draw(int x, int y) {  
        System.out.printf("Drawing '%s' icon at (%d, %d)%n", type, x, y);  
    }  
}
```

3. `IconFactory.java` – Factory using Spring Context

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.Map;

@Component
public class IconFactory {

    @Autowired
    private ApplicationContext context;

    private final Map<String, Icon> cache = new HashMap<>();

    public Icon getIcon(String type) {
        if (!cache.containsKey(type)) {
            SharedIcon icon = context.getBean(SharedIcon.class);
            icon.setType(type);
            cache.put(type, icon);
            System.out.println("Created new icon of type: " + type);
        }
        return cache.get(type);
    }
}

```

4. **IconUsageService.java** – Client using Flyweights

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class IconUsageService {

```

```

@Autowired
private IconFactory iconFactory;

public void drawUI() {
    String[] types = {"folder", "file", "trash"};

    for (int i = 0; i < 10; i++) {
        String type = types[i % types.length];
        Icon icon = iconFactory.getIcon(type);
        icon.draw(i * 10, i * 5); // x, y are extrinsic
    }
}
}

```

5. FlyweightApp.java – Main Spring Boot App

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class FlyweightApp implements CommandLineRunner {

    private final IconUsageService usageService;

    public FlyweightApp(IconUsageService usageService) {
        this.usageService = usageService;
    }

    public static void main(String[] args) {
        SpringApplication.run(FlyweightApp.class, args);
    }

    @Override
    public void run(String... args) {

```

```
        usageService.drawUI();  
    }  
}
```

✓ Output

```
Created new icon of type: folder  
Drawing 'folder' icon at (0, 0)  
Created new icon of type: file  
Drawing 'file' icon at (10, 5)  
Created new icon of type: trash  
Drawing 'trash' icon at (20, 10)  
Drawing 'folder' icon at (30, 15)  
Drawing 'file' icon at (40, 20)  
...
```

📚 Summary

| Component | Role |
|------------------|-------------------------------|
| Icon | Flyweight interface |
| SharedIcon | Intrinsic state (shared icon) |
| IconFactory | Flyweight manager |
| IconUsageService | Client using icons |
| Spring Context | Manages bean creation |

✓ Benefits of Using Spring

- ✓ Easy caching & management
- ✓ Leverage dependency injection
- ✓ Cleaner separation of concerns
- ✓ Prototype scope for fine-grained control



Enhancement Plan

We'll modify `IconFactory` to include:

- A `createdCount` to track newly created icons.
- A `reusedCount` to track reused icons.
- A new method `printStats()` to display runtime metrics.



Updated `IconFactory.java`

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.Map;

@Component
public class IconFactory {

    @Autowired
    private ApplicationContext context;

    private final Map<String, Icon> cache = new HashMap<>();
    private int createdCount = 0;
    private int reusedCount = 0;

    public Icon getIcon(String type) {
        if (!cache.containsKey(type)) {
            SharedIcon icon = context.getBean(SharedIcon.class);
            icon.setType(type);
            cache.put(type, icon);
        }
        return cache.get(type);
    }

    public void printStats() {
        System.out.println("Created icons: " + createdCount);
        System.out.println("Reused icons: " + reusedCount);
    }
}
```

```

        createdCount++;
        System.out.println("Created new icon of type: " + type);
    } else {
        reusedCount++;
        System.out.println("Reused icon of type: " + type);
    }
    return cache.get(type);
}

public void printStats() {
    System.out.println("\n--- IconFactory Runtime Stats ---");
    System.out.println("Total icon types created: " + createdCount);
    System.out.println("Total icon types reused: " + reusedCount);
    System.out.println("Cache size (distinct icons): " + cache.size());
}
}

```

Updated **IconUsageService.java**

We'll add a call to `printStats()` at the end of `drawUI()`.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class IconUsageService {

    @Autowired
    private IconFactory iconFactory;

    public void drawUI() {
        String[] types = {"folder", "file", "trash"};

        for (int i = 0; i < 10; i++) {
            String type = types[i % types.length];

```

```

        Icon icon = iconFactory.getIcon(type);
        icon.draw(i * 10, i * 5);
    }

    // Print stats after rendering
    iconFactory.printStats();
}
}

```

✓ Sample Output

```

Created new icon of type: folder
Drawing 'folder' icon at (0, 0)
Created new icon of type: file
Drawing 'file' icon at (10, 5)
Created new icon of type: trash
Drawing 'trash' icon at (20, 10)
Reused icon of type: folder
Drawing 'folder' icon at (30, 15)

...
--- IconFactory Runtime Stats ---
Total icon types created: 3
Total icon types reused: 7
Cache size (distinct icons): 3

```

✓ Summary of What We Added

| Metric | Purpose |
|--------------|---------------------------------|
| createdCount | Count of distinct icons created |
| reusedCount | Count of times cache was hit |
| printStats() | Shows the runtime metrics |

Would you like to expose this as a **REST API endpoint** (e.g., `/icons/stats`) or integrate it with **Spring Actuator metrics**?

✓ Goal

Create a REST controller with:

- `GET /icons/stats` → returns icon creation and reuse metrics as JSON.
 - Optionally: expose `GET /icons/draw-ui` to simulate drawing and trigger stats.
-

🔧 Step-by-Step Implementation

✓ 1. Add `IconStatsResponse.java` DTO

```
public class IconStatsResponse {  
    private int createdCount;  
    private int reusedCount;  
    private int cacheSize;  
  
    public IconStatsResponse(int createdCount, int reusedCount, int cacheSize)  
    {  
        this.createdCount = createdCount;  
        this.reusedCount = reusedCount;  
        this.cacheSize = cacheSize;  
    }  
  
    public int getCreatedCount() {  
        return createdCount;  
    }  
  
    public int getReusedCount() {  
        return reusedCount;  
    }  
}
```

```
}

public int getCacheSize() {
    return cacheSize;
}

}
```

✓ 2. Modify `IconFactory.java` to expose metric values

```
public int getCreatedCount() {
    return createdCount;
}

public int getReusedCount() {
    return reusedCount;
}

public int getCacheSize() {
    return cache.size();
}
```

✓ 3. Create `IconController.java`

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/icons")
public class IconController {

    @Autowired
    private IconFactory iconFactory;

    @Autowired
```

```
private IconUsageService iconUsageService;

@GetMapping("/stats")
public IconStatsResponse getStats() {
    return new IconStatsResponse(
        iconFactory.getCreatedCount(),
        iconFactory.getReusedCount(),
        iconFactory.getCacheSize()
    );
}

@PostMapping("/draw-ui")
public String drawUI() {
    iconUsageService.drawUI();
    return "UI drawn with icons. Check /icons/stats for metrics.";
}
```

Example Requests

➤ Simulate Drawing

```
POST /icons/draw-ui
```

Response:

```
UI drawn with icons. Check /icons/stats for metrics.
```

➤ Get Runtime Stats

```
GET /icons/stats
```

Response:

```
{
  "createdCount": 3,
  "reusedCount": 7,
  "cacheSize": 3
}
```

✓ Optional Enhancements

| Feature | Add-on |
|----------------------------|----------------------------|
| Use Spring Actuator | /actuator/metrics |
| Log to a DB or Grafana | Metrics exporter |
| Reset counts (for tests) | Add @PostMapping("/reset") |
| Auto cache expiry | Use @Cacheable with TTL |

10. **Facade**

🧩 What is the Facade Pattern?

The **Facade pattern** provides a simplified interface to a complex subsystem. It's useful when:

- You want to **hide system complexity** from the client.
- You want to **provide a unified interface** to multiple components.

✓ Real-Life Example: Online Order System

Subsystems:

- **InventoryService** – checks product availability
- **PaymentService** – processes payments
- **ShippingService** – handles delivery

The **Facade**: **OrderFacade** coordinates all of them.

Project Structure

```
src/
    └── controller/
        └── OrderController.java
    └── facade/
        └── OrderFacade.java
    └── services/
        ├── InventoryService.java
        ├── PaymentService.java
        └── ShippingService.java
```

1. Create Subsystems (Services)

InventoryService.java

```
package com.example.services;

import org.springframework.stereotype.Service;

@Service
public class InventoryService {
    public boolean checkStock(String productId) {
        System.out.println("Checking stock for: " + productId);
        return true; // assume in stock
    }
}
```

PaymentService.java

```
package com.example.services;

import org.springframework.stereotype.Service;
```

```
@Service
public class PaymentService {
    public boolean processPayment(String userId, double amount) {
        System.out.println("Processing payment of $" + amount + " for user: " +
userId);
        return true; // assume payment successful
    }
}
```

ShippingService.java

```
package com.example.services;

import org.springframework.stereotype.Service;

@Service
public class ShippingService {
    public void shipProduct(String productId, String userId) {
        System.out.println("Shipping " + productId + " to user: " + userId);
    }
}
```

✓ 2. Create the Facade

OrderFacade.java

```
package com.example.facade;

import com.example.services.InventoryService;
import com.example.services.PaymentService;
import com.example.services.ShippingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
public class OrderFacade {

    @Autowired
    private InventoryService inventoryService;

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private ShippingService shippingService;

    public String placeOrder(String productId, String userId, double amount) {
        if (!inventoryService.checkStock(productId)) {
            return "Product out of stock!";
        }

        if (!paymentService.processPayment(userId, amount)) {
            return "Payment failed!";
        }

        shippingService.shipProduct(productId, userId);
        return "Order placed successfully!";
    }
}
```

✓ 3. Create REST Controller

OrderController.java

```
package com.example.controller;

import com.example.facade.OrderFacade;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
```

```

@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderFacade orderFacade;

    @PostMapping("/place")
    public String placeOrder(@RequestParam String productId,
                            @RequestParam String userId,
                            @RequestParam double amount) {
        return orderFacade.placeOrder(productId, userId, amount);
    }
}

```

Test via Postman or cURL

POST http://localhost:8080/orders/place?productId=123&userId=abc&amount=50

Output:

Checking stock for: 123
 Processing payment of \$50.0 for user: abc
 Shipping 123 to user: abc
 Order placed successfully!

Summary

| Layer | Role |
|-----------------|---|
| OrderController | Accepts request from user |
| OrderFacade | Simplifies interactions with subsystems |

| | |
|----------|---|
| Services | Do the actual work (inventory, payment, shipping) |
|----------|---|

🎯 Use Case: Online Order System

You have subsystems:

- `InventoryService` – checks if product is in stock
- `PaymentService` – handles payment
- `ShippingService` – manages delivery

The `OrderFacade` class simplifies interaction with all these services.

✓ 1. Define the Subsystems

InventoryService.java

```
public class InventoryService {  
    public boolean checkStock(String productId) {  
        System.out.println("Checking stock for: " + productId);  
        return true; // Always available for demo  
    }  
}
```

PaymentService.java

```
public class PaymentService {  
    public boolean processPayment(String userId, double amount) {  
        System.out.println("Processing payment of $" + amount + " for user: " +  
            userId);  
        return true; // Payment always succeeds  
    }  
}
```

```
    }  
}
```

ShippingService.java

```
public class ShippingService {  
    public void shipProduct(String productId, String userId) {  
        System.out.println("Shipping product " + productId + " to user: " + userId);  
    }  
}
```

✓ 2. Create the Facade

OrderFacade.java

```
public class OrderFacade {  
  
    private InventoryService inventoryService;  
    private PaymentService paymentService;  
    private ShippingService shippingService;  
  
    public OrderFacade() {  
        this.inventoryService = new InventoryService();  
        this.paymentService = new PaymentService();  
        this.shippingService = new ShippingService();  
    }  
  
    public String placeOrder(String productId, String userId, double amount) {  
        if (!inventoryService.checkStock(productId)) {  
            return "Order failed: Product out of stock!";  
        }  
  
        if (!paymentService.processPayment(userId, amount)) {
```

```
        return "Order failed: Payment failed!";
    }

    shippingService.shipProduct(productId, userId);
    return "Order placed successfully!";
}
}
```

✓ 3. Main Method to Run It

Main.java

```
public class Main {
    public static void main(String[] args) {
        OrderFacade orderFacade = new OrderFacade();

        String result = orderFacade.placeOrder("P123", "U456", 99.99);
        System.out.println(result);
    }
}
```

✓ Output

```
Cecking stock for: P123
Processing payment of $99.99 for user: U456
Shipping product P123 to user: U456
Order placed successfully!
```

✓ Summary

| Class | Responsibility |
|------------------|-----------------------------|
| InventoryService | Checks product availability |

| | |
|-----------------|-------------------------------------|
| PaymentService | Handles user payment |
| ShippingService | Ships the product |
| OrderFacade | Combines above steps via one method |
| Main | Uses the Facade |

Structural

1. Chain Of Responsibility

The **Chain of Responsibility (CoR)** design pattern allows you to **pass a request along a chain of handlers** until one of them handles it. It's useful when multiple objects can handle a request and you want to **decouple the sender and receiver**.

✓ Real-World Use Case

Logging System – where logs are passed from `DEBUG` → `INFO` → `ERROR`, and each level decides whether to process or pass it along.

Other use cases:

- Event processing pipelines
- Authentication filters
- Form validation chains
- Middleware in web frameworks

✓ Goal: Logging Handler Example in Java

Chain:

`DebugLogger` → `InfoLogger` → `ErrorLogger`

Each logger decides if it can handle the log level; if not, it passes it forward.

File Structure

```
src/
├── Logger.java      (abstract handler)
├── DebugLogger.java (concrete handler)
├── InfoLogger.java
├── ErrorLogger.java
└── Main.java        (to run the chain)
```

1. Abstract Logger (Handler)

```
public abstract class Logger {
    public static final int DEBUG = 1;
    public static final int INFO = 2;
    public static final int ERROR = 3;

    protected int level;
    protected Logger nextLogger;

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

    public void logMessage(int level, String message) {
        if (this.level <= level) {
            write(message);
        }
        if (nextLogger != null) {
            nextLogger.logMessage(level, message);
        }
    }
}
```

```
    protected abstract void write(String message);  
}
```

✓ 2. Concrete Loggers

DebugLogger.java

```
public class DebugLogger extends Logger {  
    public DebugLogger() {  
        this.level = DEBUG;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("[DEBUG]: " + message);  
    }  
}
```

InfoLogger.java

```
public class InfoLogger extends Logger {  
    public InfoLogger() {  
        this.level = INFO;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("[INFO]: " + message);  
    }  
}
```

ErrorLogger.java

```
public class ErrorLogger extends Logger {  
    public ErrorLogger() {  
        this.level = ERROR;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("[ERROR]: " + message);  
    }  
}
```

✓ 3. Run the Chain

Main.java

```
public class Main {  
    private static Logger getLoggerChain() {  
        Logger errorLogger = new ErrorLogger();  
        Logger infoLogger = new InfoLogger();  
        Logger debugLogger = new DebugLogger();  
  
        debugLogger.setNextLogger(infoLogger);  
        infoLogger.setNextLogger(errorLogger);  
  
        return debugLogger;  
    }  
  
    public static void main(String[] args) {  
        Logger loggerChain = getLoggerChain();  
  
        loggerChain.logMessage(Logger.DEBUG, "Debugging application.");  
        loggerChain.logMessage(Logger.INFO, "Information message.");  
        loggerChain.logMessage(Logger.ERROR, "An error occurred!");  
    }  
}
```

```
    }  
}
```

✓ Output

[DEBUG]: Debugging application.

[INFO]: Information message.

[ERROR]: An error occurred!

Each message is passed through the chain; each handler decides whether to act.

🧠 Summary

| Concept | Description |
|-------------|---|
| Pattern | Chain of Responsibility |
| Purpose | Decouple sender and receivers |
| Key Benefit | Flexibility in processing, avoids tight coupling |
| Real Uses | Loggers, Authentication filters, Validation pipelines |

The **Chain of Responsibility (CoR)** design pattern allows you to **pass a request along a chain of handlers** until one of them handles it. It's useful when multiple objects can handle a request and you want to **decouple the sender and receiver**.

✓ Real-World Use Case

Logging System – where logs are passed from DEBUG → INFO → ERROR, and each level decides whether to process or pass it along.

Other use cases:

- Event processing pipelines
- Authentication filters

- Form validation chains
 - Middleware in web frameworks
-

✓ Goal: Logging Handler Example in Java

Chain:

DebugLogger → InfoLogger → ErrorLogger

Each logger decides if it can handle the log level; if not, it passes it forward.

📁 File Structure

```
src/
    ├── Logger.java      (abstract handler)
    ├── DebugLogger.java (concrete handler)
    ├── InfoLogger.java
    ├── ErrorLogger.java
    └── Main.java        (to run the chain)
```

✓ 1. Abstract Logger (Handler)

```
public abstract class Logger {
    public static final int DEBUG = 1;
    public static final int INFO = 2;
    public static final int ERROR = 3;

    protected int level;
    protected Logger nextLogger;

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

```
public void logMessage(int level, String message) {  
    if (this.level <= level) {  
        write(message);  
    }  
    if (nextLogger != null) {  
        nextLogger.logMessage(level, message);  
    }  
}  
  
protected abstract void write(String message);  
}
```

✓ 2. Concrete Loggers

DebugLogger.java

```
public class DebugLogger extends Logger {  
    public DebugLogger() {  
        this.level = DEBUG;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("[DEBUG]: " + message);  
    }  
}
```

InfoLogger.java

```
java  
CopyEdit  
public class InfoLogger extends Logger {  
    public InfoLogger() {  
        this.level = INFO;
```

```
}

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

ErrorLogger.java

```
java
CopyEdit
public class ErrorLogger extends Logger {
    public ErrorLogger() {
        this.level = ERROR;
    }

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

✓ 3. Run the Chain

Main.java

```
java
CopyEdit
public class Main {
    private static Logger getLoggerChain() {
        Logger errorLogger = new ErrorLogger();
```

```

Logger infoLogger = new InfoLogger();
Logger debugLogger = new DebugLogger();

debugLogger.setNextLogger(infoLogger);
infoLogger.setNextLogger(errorLogger);

return debugLogger;
}

public static void main(String[] args) {
    Logger loggerChain = getLoggerChain();

    loggerChain.logMessage(Logger.DEBUG, "Debugging application.");
    loggerChain.logMessage(Logger.INFO, "Information message.");
    loggerChain.logMessage(Logger.ERROR, "An error occurred!");
}
}

```

Output

markdown
[CopyEdit](#)
[DEBUG]: Debugging application.
[INFO]: Information message.
[ERROR]: An error occurred!

Each message is passed through the chain; each handler decides whether to act.

Summary

| Concept | Description |
|---------|-------------------------|
| Pattern | Chain of Responsibility |

| | |
|-------------|---|
| Purpose | Decouple sender and receivers |
| Key Benefit | Flexibility in processing, avoids tight coupling |
| Real Uses | Loggers, Authentication filters, Validation pipelines |

Would you like to see this adapted for a **Spring Boot-based middleware or validator chain** example?

You said:

use spring boot

ChatGPT said:

Let's implement the **Chain of Responsibility** pattern in **Spring Boot** with a real-world scenario.

✓ Use Case: Request Validation Chain

You want to **process a user registration request** through multiple steps:

1. **Check if email is present**
2. **Check if password is strong enough**
3. **Check if user already exists**

Each validation is implemented as a **Spring bean**, and chained dynamically using the Chain of Responsibility pattern.

Project Structure

```
src/
  └── chain/
    ├── RequestValidator.java
    ├── EmailValidator.java
    ├── PasswordValidator.java
    └── UserExistValidator.java
  └── controller/
    └── RegisterController.java
```

```
|── model/
|   └── RegisterRequest.java
└── SpringBootApp.java
```

✓ 1. Define Base Handler (RequestValidator)

```
package com.example.chain;

import com.example.model.RegisterRequest;

public abstract class RequestValidator {
    protected RequestValidator next;

    public RequestValidator linkWith(RequestValidator nextValidator) {
        this.next = nextValidator;
        return nextValidator;
    }

    public void validate(RegisterRequest request) {
        handle(request);
        if (next != null) {
            next.validate(request);
        }
    }

    protected abstract void handle(RegisterRequest request);
}
```

✓ 2. Create Validators

EmailValidator.java

```
@Component
public class EmailValidator extends RequestValidator {
    @Override
    protected void handle(RegisterRequest request) {
        if (request.getEmail() == null || !request.getEmail().contains("@")) {
            throw new RuntimeException("Invalid email");
        }
    }
}
```

PasswordValidator.java

```
@Component
public class PasswordValidator extends RequestValidator {
    @Override
    protected void handle(RegisterRequest request) {
        if (request.getPassword() == null || request.getPassword().length() < 6) {
            throw new RuntimeException("Password too short");
        }
    }
}
```

UserExistValidator.java

```
@Component
public class UserExistValidator extends RequestValidator {
    @Override
    protected void handle(RegisterRequest request) {
        if ("existing@example.com".equals(request.getEmail())) {
            throw new RuntimeException("User already exists");
        }
    }
}
```

✓ 3. Create **RegisterRequest** DTO

```
package com.example.model;

public class RegisterRequest {
    private String email;
    private String password;

    // Getters and setters
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

✓ 4. Create Controller to Use the Chain

```
@RestController
@RequestMapping("/api")
public class RegisterController {

    private final EmailValidator emailValidator;
    private final PasswordValidator passwordValidator;
    private final UserExistValidator userExistValidator;

    public RegisterController(EmailValidator emailValidator,
                            PasswordValidator passwordValidator,
                            UserExistValidator userExistValidator) {
        this.emailValidator = emailValidator;
        this.passwordValidator = passwordValidator;
        this.userExistValidator = userExistValidator;

        // Link the chain
    }
}
```

```
        emailValidator.linkWith(passwordValidator).linkWith(userExistValidator);
    }

    @PostMapping("/register")
    public ResponseEntity<String> register(@RequestBody RegisterRequest request) {
        emailValidator.validate(request);
        return ResponseEntity.ok("Registration valid");
    }
}
```

✓ 5. Main Application

```
@SpringBootApplication
public class SpringBootApp {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootApp.class, args);
    }
}
```

✓ Sample Test with curl

```
curl -X POST http://localhost:8080/api/register \
-H "Content-Type: application/json" \
-d '{"email":"test@example.com", "password":"123456"}'
```

✓ Returns: Registration valid

Chain Flow

If email is invalid → throws

Else → goes to password check

Else → goes to existing user check

Else → success!

Summary

| Concept | Applied With |
|-------------------------|---|
| Chain of Responsibility | Spring Beans (Validators) |
| Chain Linked In | Controller class |
| Real-World Use Case | Request validation, filter chains, middleware |

2. Command (Action Or Transaction)

The **Command Design Pattern** encapsulates a request as an object, thereby allowing you to **parameterize clients with queues, requests, and operations**, and support **undoable operations**.

Real-World Use Case

Example: Remote Control System (like a smart home system)

You want to:

- Turn a light on/off
- Start/stop a fan
- Schedule commands
- Support undo

Each action (like turning on/off devices) can be encapsulated as a command object.

Participants in Command Pattern

| Role | Class |
|-----------------|-------------------------|
| Command | Command interface |
| ConcreteCommand | LightOnCommand , etc. |
| Receiver | Light , Fan , etc. |
| Invoker | RemoteControl |
| Client | Main class to bind them |

✓ Java Implementation

1. Command Interface

```
public interface Command {
    void execute();
    void undo(); // optional: for undo functionality
}
```

2. Receiver Classes (Devices)

Light.java

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

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

Fan.java

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

3. Concrete Command Classes

LightOnCommand.java

```
public class LightOnCommand implements Command {  
    private final Light light;  
  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }  
  
    public void execute() {  
        light.turnOn();  
    }  
  
    public void undo() {  
        light.turnOff();  
    }  
}
```

FanStartCommand.java

```
public class FanStartCommand implements Command {  
    private final Fan fan;
```

```
public FanStartCommand(Fan fan) {  
    this.fan = fan;  
}  
  
public void execute() {  
    fan.start();  
}  
  
public void undo() {  
    fan.stop();  
}  
}
```

4. Invoker: [RemoteControl](#)

```
public class RemoteControl {  
    private Command command;  
  
    public void setCommand(Command command) {  
        this.command = command;  
    }  
  
    public void pressButton() {  
        command.execute();  
    }  
  
    public void pressUndo() {  
        command.undo();  
    }  
}
```

5. Client: [Main.java](#)

```

public class Main {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Fan ceilingFan = new Fan();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command fanStart = new FanStartCommand(ceilingFan);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);
        remote.pressButton(); // Light is ON
        remote.pressUndo(); // Light is OFF

        remote.setCommand(fanStart);
        remote.pressButton(); // Fan started
        remote.pressUndo(); // Fan stopped
    }
}

```

Output

Light is ON
 Light is OFF
 Fan started
 Fan stopped

Benefits of the Command Pattern

| Feature | Benefit |
|--------------------------------|--|
| Decouples sender from receiver | Remote doesn't need to know device internals |
| Supports undo | Built-in undo method |

| | |
|-----------------------------|--------------------------------------|
| Can queue/schedule commands | Useful in task schedulers |
| Reusability | Commands are reusable and composable |

✓ Summary

- Use **Command pattern** when you need to encapsulate operations or actions as objects.
- Works great for **UI buttons, job queues, undo/redo features, or transactional operations.**

✓ Use Case: Device Command Execution via REST API

We'll simulate a system where the client can send commands like:

- Turn **light on/off**
- Start/stop **fan**

These commands are encapsulated as Command objects and executed through a controller.

📦 Structure Overview

```
src/
  └── command/
    ├── Command.java
    ├── LightOnCommand.java
    ├── LightOffCommand.java
    ├── FanStartCommand.java
    ├── FanStopCommand.java
    └── CommandExecutor.java
  └── devices/
```

```
|   └── Light.java  
|   └── Fan.java  
├── controller/  
|   └── DeviceController.java  
├── dto/  
|   └── CommandRequest.java  
└── SpringBootApp.java
```

✓ 1. Command Interface

```
package com.example.command;  
  
public interface Command {  
    String execute();  
}
```

✓ 2. Device Classes (Receivers)

Light.java

```
package com.example.devices;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Light {  
    public String turnOn() {  
        return "Light turned ON";  
    }  
  
    public String turnOff() {  
        return "Light turned OFF";  
    }  
}
```

```
    }  
}
```

Fan.java

```
package com.example.devices;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Fan {  
    public String start() {  
        return "Fan started";  
    }  
  
    public String stop() {  
        return "Fan stopped";  
    }  
}
```

✓ 3. Concrete Commands

LightOnCommand.java

```
package com.example.command;  
  
import com.example.devices.Light;  
  
public class LightOnCommand implements Command {  
    private final Light light;  
  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }
```

```
@Override  
public String execute() {  
    return light.turnOn();  
}  
}
```

FanStopCommand.java (similarly create **LightOffCommand**, **FanStartCommand**)

```
package com.example.command;  
  
import com.example.devices.Fan;  
  
public class FanStopCommand implements Command {  
    private final Fan fan;  
  
    public FanStopCommand(Fan fan) {  
        this.fan = fan;  
    }  
  
    @Override  
    public String execute() {  
        return fan.stop();  
    }  
}
```

✓ 4. **CommandExecutor.java** – Invoker

```
package com.example.command;  
  
import org.springframework.stereotype.Component;  
  
@Component
```

```
public class CommandExecutor {  
    public String executeCommand(Command command) {  
        return command.execute();  
    }  
}
```

✓ 5. **CommandRequest.java** – DTO

```
package com.example.dto;  
  
public class CommandRequest {  
    private String device;  
    private String action;  
  
    // getters and setters  
    public String getDevice() { return device; }  
    public void setDevice(String device) { this.device = device; }  
  
    public String getAction() { return action; }  
    public void setAction(String action) { this.action = action; }  
}
```

✓ 6. **DeviceController.java**

```
package com.example.controller;  
  
import com.example.command.*;  
import com.example.devices.Fan;  
import com.example.devices.Light;  
import com.example.dto.CommandRequest;  
import org.springframework.web.bind.annotation.*;  
  
@RestController
```

```

@RequestMapping("/api/device")
public class DeviceController {

    private final Light light;
    private final Fan fan;
    private final CommandExecutor executor;

    public DeviceController(Light light, Fan fan, CommandExecutor executor) {
        this.light = light;
        this.fan = fan;
        this.executor = executor;
    }

    @PostMapping("/command")
    public String executeCommand(@RequestBody CommandRequest request)
    {
        Command command;

        switch (request.getDevice().toLowerCase()) {
            case "light":
                command = request.getAction().equalsIgnoreCase("on")
                    ? new LightOnCommand(light)
                    : new LightOffCommand(light);
                break;

            case "fan":
                command = request.getAction().equalsIgnoreCase("start")
                    ? new FanStartCommand(fan)
                    : new FanStopCommand(fan);
                break;

            default:
                return "Invalid device";
        }

        return executor.executeCommand(command);
    }
}

```

```
    }  
}
```

✓ 7. **SpringBootApp.java**

```
package com.example;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class SpringBootApp {  
    public static void main(String[] args) {  
        SpringApplication.run(SpringBootApp.class, args);  
    }  
}
```

✓ Test with CURL

Light ON

```
curl -X POST http://localhost:8080/api/device/command \  
-H "Content-Type: application/json" \  
-d '{"device": "light", "action": "on"}'
```

💡 Output:

```
Light turned ON
```

Fan STOP

```
curl -X POST http://localhost:8080/api/device/command \
-H "Content-Type: application/json" \
-d '{"device": "fan", "action": "stop"}'
```

⌚ Output:

Fan stopped

🧠 Summary

| Component | Role |
|-----------------|------------------------|
| Command | Encapsulates a request |
| Light , Fan | Receivers |
| CommandExecutor | Invoker |
| Controller | Client |
| DTO | Input abstraction |

To integrate a **Queue** with the **Command Design Pattern in Spring Boot**, you can enhance the `CommandExecutor` to enqueue and process commands asynchronously or in batch — like a **task scheduler**, **job queue**, or **command dispatcher**.

✓ Updated Use Case

We'll modify our previous Spring Boot implementation to:

1. Queue commands.
2. Execute them **in order** (FIFO).
3. Optionally simulate delayed/background execution.

✓ Step-by-Step Enhancements



1. Modify **CommandExecutor** to Queue Commands

```
package com.example.command;

import jakarta.annotation.PostConstruct;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

import java.util.LinkedList;
import java.util.Queue;

@Component
public class CommandExecutor {

    private final Queue<Command> commandQueue = new LinkedList<>();

    @PostConstruct
    public void startExecutionLoop() {
        // Start a background thread to execute commands
        new Thread(this::processCommands).start();
    }

    public void enqueue(Command command) {
        synchronized (commandQueue) {
            commandQueue.offer(command);
            commandQueue.notify(); // Wake up the processor thread
        }
    }

    private void processCommands() {
        while (true) {
            Command command;
            synchronized (commandQueue) {
                while (commandQueue.isEmpty()) {
                    try {
                        commandQueue.wait(); // Wait for a command to arrive
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                command = commandQueue.poll();
            }
            if (command != null) {
                command.execute();
            }
        }
    }
}
```

```
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return;
        }
    }
    command = commandQueue.poll();
}

if (command != null) {
    String result = command.execute();
    System.out.println("[Executed] " + result);
}
}
```

2. Modify Controller to Enqueue Instead of Execute Immediately

```
@PostMapping("/command")
public String enqueueCommand(@RequestBody CommandRequest request) {
    Command command;

    switch (request.getDevice().toLowerCase()) {
        case "light":
            command = request.getAction().equalsIgnoreCase("on")
                ? new LightOnCommand(light)
                : new LightOffCommand(light);
            break;
        case "fan":
            command = request.getAction().equalsIgnoreCase("start")
                ? new FanStartCommand(fan)
                : new FanStopCommand(fan);
            break;
        default:
```

```
        return "Invalid device or action";  
    }  
  
    executor.enqueue(command);  
    return "Command enqueued: " + request.getDevice() + " " + request.getAction();  
}
```

Sample Output

When you call the endpoint:

```
curl -X POST http://localhost:8080/api/device/command \  
-H "Content-Type: application/json" \  
-d '{"device": "light", "action": "on"}'
```

Console logs:

```
Command enqueued: light on  
[Executed] Light turned ON
```

Optional: Use `@Async` for true background tasks

Enable `@Async` execution by adding:

`SpringBootApp.java`

```
@SpringBootApplication  
@EnableAsync  
public class SpringBootApp { ... }
```

Then modify `processCommands()` to use `@Async` methods if you want parallel execution (not FIFO).

Summary: Why Use Queue in Command Pattern?

| Benefit | Description |
|----------------------------------|--|
| Decouples invocation & execution | Useful in background job systems |
| FIFO processing | Ensures order of command execution |
| Supports async workloads | Schedule, batch, or throttle command dispatching |

3. Iterator

The **Iterator pattern** allows you to **traverse a collection** (like a list, set, or custom data structure) **one element at a time**, without knowing its internal structure.

Key Components

1. **Iterator Interface** – defines the methods for accessing elements (`hasNext()`, `next()`).
 2. **Concrete Iterator** – implements the  `Iterator` interface.
 3. **Aggregate Interface** – defines a method to create an iterator.
 4. **Concrete Aggregate** – returns an instance of the concrete iterator.
-

Java Example

Step 1: Iterator Interface

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

Step 2: Aggregate Interface

```
public interface Collection<T> {  
    Iterator<T> createIterator();  
}
```

Step 3: Concrete Collection

```
public class NameRepository implements Collection<String> {  
    private String[] names = { "Sai", "Ashish", "Ravi", "Priya" };  
  
    @Override  
    public Iterator<String> createIterator() {  
        return new Nameliterator();  
    }  
  
    private class Nameliterator implements Iterator<String> {  
        int index = 0;  
  
        @Override  
        public boolean hasNext() {  
            return index < names.length;  
        }  
  
        @Override  
        public String next() {  
            return hasNext() ? names[index++] : null;  
        }  
    }  
}
```

Step 4: Client Usage

```
public class Main {  
    public static void main(String[] args) {  
        NameRepository repo = new NameRepository();
```

```
Iterator<String> iterator = repo.createIterator();

while (iterator.hasNext()) {
    System.out.println(iterator.next());
}

}
```

Use Cases

- **Custom collection frameworks** – when creating your own data structures.
- **Traversal abstraction** – when you want to hide the internal structure (tree, graph, etc.).
- **Multiple traversals** – forward, reverse, or filtered iteration.
- **Undo functionality** – iterate over a history stack.

Advantages

- Promotes **encapsulation** by hiding collection internals.
- Allows **multiple traversals**.
- Clean separation between collection and traversal logic.

Real-World Examples in Java

- Java's `Iterator<E>` interface in the `java.util` package.

```
List<String> list = Arrays.asList("A", "B", "C");
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

Implementing the **Iterator Design Pattern** in a **Spring Boot application** is similar to the standard Java implementation—but it's typically used in contexts like:

- Processing elements from a database or repository.
- Iterating over services or tasks.
- Streamlining batch operations.

Let's walk through a **realistic Spring Boot example** where we simulate **iterating over a list of users** from a database using a custom iterator.

Use Case: User Iterator to Process Users One by One

Step 1: Define the `User` Entity

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    // getters and setters
}
```

Step 2: JPA Repository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> { }
```

Step 3: Create the Iterator Interface

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

Step 4: User Collection and Iterator

```
public class UserCollection {  
    private List<User> users;  
  
    public UserCollection(List<User> users) {  
        this.users = users;  
    }  
  
    public Iterator<User> getIterator() {  
        return new UserIterator();  
    }  
  
    private class UserIterator implements Iterator<User> {  
        private int index = 0;  
  
        @Override  
        public boolean hasNext() {  
            return index < users.size();  
        }  
  
        @Override  
        public User next() {  
            return hasNext() ? users.get(index++) : null;  
        }  
    }  
}
```

Step 5: Service to Use the Iterator

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public void processUsers() {
        List<User> allUsers = userRepository.findAll();
        UserCollection userCollection = new UserCollection(allUsers);
        Iterator<User> iterator = userCollection.getIterator();

        while (iterator.hasNext()) {
            User user = iterator.next();
            System.out.println("Processing user: " + user.getName());
        }
    }
}
```

Step 6: Trigger from Controller

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/process")
    public String processUsers() {
        userService.processUsers();
        return "User processing complete!";
    }
}
```

```
}
```

✓ When to Use in Spring Boot

- **Streaming results** (e.g., `@Transactional` with custom iterators).
- **Paginated processing** using iterators.
- **Batch jobs or ETL pipelines**.
- **Workflow or task orchestration** over a list of steps/entities.

4. `Interpreter`

🧠 Interpreter Design Pattern in Java (Design Patterns)

The **Interpreter pattern** is a **behavioral design pattern** that is used to **define a grammar** for a language and **interpret sentences** in that language using classes representing grammar rules.



Definition

The Interpreter Pattern defines a representation for a grammar and provides an interpreter to deal with this grammar. It is used when we want to evaluate language expressions.



Use Case

- **Arithmetic expression evaluators** (`2 + 3 * 4`)
- **Rule engines**
- **SQL parsing**
- **Mini-language processing**

- **Regex parsing**
 - **Search filters**
-



Components of Interpreter Pattern

| Component | Description |
|-----------------------|--|
| Expression | Abstract interface for all expressions |
| TerminalExpression | Represents atomic expressions (e.g., numbers) |
| NonTerminalExpression | Represents operators or rules (e.g., add, multiply) |
| Context | Contains information global to the interpreter (can be optional) |

✓ Java Example: Arithmetic Expression Evaluator

Step 1: Expression Interface

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

Step 2: TerminalExpression (Number)

```
public class Number implements Expression {  
    private int number;  
  
    public Number(int number) {  
        this.number = number;  
    }  
  
    @Override  
    public int interpret() {  
        return number;  
    }  
}
```

```
    }  
}
```

Step 3: NonTerminalExpressions (Add, Subtract)

```
public class Add implements Expression {  
    private Expression left, right;  
  
    public Add(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    @Override  
    public int interpret() {  
        return left.interpret() + right.interpret();  
    }  
}  
  
public class Subtract implements Expression {  
    private Expression left, right;  
  
    public Subtract(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    @Override  
    public int interpret() {  
        return left.interpret() - right.interpret();  
    }  
}
```

Step 4: Client Usage

```

public class InterpreterPatternDemo {
    public static void main(String[] args) {
        // Represents: (5 + 3) - 2
        Expression expr = new Subtract(
            new Add(new Number(5), new Number(3)),
            new Number(2)
        );

        System.out.println("Result: " + expr.interpret()); // Output: 6
    }
}

```

✓ Real-Life Use Cases

| Domain | Example |
|--|---|
| Spring Expression Language (SpEL) | Used for evaluating expressions in config, security rules, etc. |
| SQL Engines | Parsing and interpreting queries |
| Rule Engines (e.g., Drools) | Evaluating custom business logic |
| Custom Query Filters | "price > 100 AND stock < 50" |

! Pros and Cons

✓ Pros

- Easy to add new expressions or operations
- Good for small domain-specific languages (DSLs)
- Clean, object-oriented grammar representation

✗ Cons

- Becomes complex for large grammars
- Lots of small classes (can lead to class explosion)

- Not suitable for performance-critical code

5. Mediator

It allows you to encapsulate the interaction between objects (colleagues) into a separate mediator object, improving code maintainability and scalability.

The **Mediator Design Pattern** is a behavioral design pattern that **centralizes complex communications** and control logic between related objects in a system, promoting **loose coupling** by preventing objects from referring to each other explicitly.

Purpose of Mediator Pattern

It allows you to encapsulate the interaction between objects (colleagues) into a separate mediator object, improving code maintainability and scalability.

Key Components

| Component | Description |
|---|---|
| Mediator | Interface that defines communication methods |
| ConcreteMediator | Implements the mediator and coordinates interactions |
| Colleague | Abstract class/component that interacts via the mediator |
| ConcreteColleague | Implements behavior and delegates communication to mediator |

✓ Java Implementation Example

🎯 Use Case: Chatroom where users send messages through a central mediator

Step 1: Mediator Interface

```
public interface ChatMediator {  
    void sendMessage(String message, User sender);  
    void addUser(User user);  
}
```

Step 2: Concrete Mediator

```
import java.util.ArrayList;  
import java.util.List;  
  
public class ChatRoom implements ChatMediator {  
    private List<User> users = new ArrayList<>();  
  
    @Override  
    public void sendMessage(String message, User sender) {  
        for (User user : users) {  
            if (user != sender) {  
                user.receive(message);  
            }  
        }  
    }  
  
    @Override  
    public void addUser(User user) {  
        users.add(user);  
    }  
}
```

Step 3: Abstract Colleague

```
public abstract class User {  
    protected ChatMediator mediator;  
    protected String name;  
  
    public User(ChatMediator mediator, String name) {  
        this.mediator = mediator;  
        this.name = name;  
    }  
  
    public abstract void send(String message);  
    public abstract void receive(String message);  
}
```

Step 4: Concrete Colleague

```
public class ChatUser extends User {  
  
    public ChatUser(ChatMediator mediator, String name) {  
        super(mediator, name);  
    }  
  
    @Override  
    public void send(String message) {  
        System.out.println(name + " sends: " + message);  
        mediator.sendMessage(message, this);  
    }  
  
    @Override  
    public void receive(String message) {  
        System.out.println(name + " received: " + message);  
    }  
}
```

```
    }  
}
```

Step 5: Client Demo

```
public class MediatorPatternDemo {  
    public static void main(String[] args) {  
        ChatMediator mediator = new ChatRoom();  
  
        User user1 = new ChatUser(mediator, "Alice");  
        User user2 = new ChatUser(mediator, "Bob");  
        User user3 = new ChatUser(mediator, "Charlie");  
  
        mediator.addUser(user1);  
        mediator.addUser(user2);  
        mediator.addUser(user3);  
  
        user1.send("Hello everyone!");  
    }  
}
```

Output

```
Alice sends: Hello everyone!  
Bob received: Hello everyone!  
Charlie received: Hello everyone!
```

Real-World Use Cases

| Scenario | Example |
|------------------------------------|---|
| UI Components Communication | A form where buttons, fields, and checkboxes update each other via a mediator |

| | |
|-----------------------------------|---|
| Chat Systems | User-to-user or group chats |
| Air Traffic Control System | Planes communicate with the control tower, not directly with each other |
| Workflow Engines | Tasks communicate via a central controller |

✓ Benefits

- Reduces coupling between components
- Centralizes logic for object communication
- Improves code readability and maintainability

✗ Drawbacks

- Mediator can become too complex and turn into a "God Object" if not handled carefully

🧠 Use Case: Order Notification System

When a new order is placed:

- **Email service** should notify the customer.
- **Inventory service** should update stock.
- **Shipping service** should prepare shipment.

🔄 Instead of each service calling others directly, we'll use a **mediator** to coordinate them — achieving **loose coupling**.

📦 Project Structure

```

src/
  └── main/
    └── java/
      └── com/example/mediator/
  
```

```
|── mediator/
|   ├── Mediator.java
|   └── OrderMediator.java
├── services/
|   ├── EmailService.java
|   ├── InventoryService.java
|   └── ShippingService.java
└── controller/
    └── OrderController.java
└── Application.java
```

✓ 1. Define the Mediator Interface

```
package com.example.mediator.mediator;

import com.example.mediator.model.Order;

public interface Mediator {
    void notifyServices(Order order);
}
```

✓ 2. Define the Services (Colleagues)

```
package com.example.mediator.services;

import com.example.mediator.model.Order;
import org.springframework.stereotype.Service;

@Service
public class EmailService {
    public void sendConfirmation(Order order) {
        System.out.println("Email sent to customer for Order: " + order.getId());
```

```
    }
}

package com.example.mediator.services;

import com.example.mediator.model.Order;
import org.springframework.stereotype.Service;

@Service
public class InventoryService {
    public void updateStock(Order order) {
        System.out.println("Inventory updated for Product: " + order.getProduct());
    }
}
```

```
package com.example.mediator.services;

import com.example.mediator.model.Order;
import org.springframework.stereotype.Service;

@Service
public class ShippingService {
    public void prepareShipment(Order order) {
        System.out.println("Shipment prepared for Order: " + order.getId());
    }
}
```

✓ 3. Define the Order Model

```
package com.example.mediator.model;

public class Order {
```

```
private String id;  
private String product;  
  
// Constructor, getters  
public Order(String id, String product) {  
    this.id = id;  
    this.product = product;  
}  
  
public String getId() { return id; }  
public String getProduct() { return product; }  
}
```

✓ 4. Implement the Concrete Mediator

```
package com.example.mediator.mediator;  
  
import com.example.mediator.model.Order;  
import com.example.mediator.services.EmailService;  
import com.example.mediator.services.InventoryService;  
import com.example.mediator.services.ShippingService;  
import org.springframework.stereotype.Component;  
  
@Component  
public class OrderMediator implements Mediator {  
  
    private final EmailService emailService;  
    private final InventoryService inventoryService;  
    private final ShippingService shippingService;  
  
    public OrderMediator(EmailService emailService, InventoryService inventoryService, ShippingService shippingService) {  
        this.emailService = emailService;  
        this.inventoryService = inventoryService;
```

```

        this.shippingService = shippingService;
    }

@Override
public void notifyServices(Order order) {
    emailService.sendConfirmation(order);
    inventoryService.updateStock(order);
    shippingService.prepareShipment(order);
}
}

```

5. Controller to Trigger the Mediator

```

package com.example.mediator.controller;

import com.example.mediator.mediator.Mediator;
import com.example.mediator.model.Order;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/orders")
public class OrderController {

    private final Mediator orderMediator;

    public OrderController(Mediator orderMediator) {
        this.orderMediator = orderMediator;
    }

    @PostMapping("/create")
    public String createOrder(@RequestParam String id, @RequestParam String product) {
        Order order = new Order(id, product);
        orderMediator.notifyServices(order);
    }
}

```

```
        return "Order processed successfully.";  
    }  
}
```

✓ 6. Spring Boot Main Application

```
package com.example.mediator;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class MediatorApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(MediatorApplication.class, args);  
    }  
}
```

 **Sample Output on POST /orders/create?**
id=123&product=Laptop

```
Email sent to customer for Order: 123  
Inventory updated for Product: Laptop  
Shipment prepared for Order: 123
```

✓ Benefits in Spring Boot

- **Loose coupling** between services
- Easy to extend (add LoggingService etc.)
- Easy testing: test mediator behavior separately
- Follows **Single Responsibility Principle**

6. Memento

The **Memento Design Pattern** is a **behavioral pattern** used to capture and externalize an object's internal state so that it can be restored later **without violating encapsulation**.

Intent

“Without exposing the object’s internal structure, you can save and restore its state.

Key Components

| Role | Description |
|--|--|
| Originator | The object whose state needs to be saved and restored |
| Memento | Stores the internal state of the Originator ; it’s immutable to others |
| Caretaker | Maintains a list of mementos and controls save/restore without accessing their data |

Use Case Example in Java

Scenario: Text Editor with Undo Functionality

1. Memento Class (Immutable)

```
public class TextEditorMemento {  
    private final String content;  
  
    public TextEditorMemento(String content) {  
        this.content = content;  
    }
```

```
}

public String getContent() {
    return content;
}
}
```

◆ 2. Originator (Text Editor)

```
public class TextEditor {
    private String content = "";

    public void type(String words) {
        content += words;
    }

    public String getContent() {
        return content;
    }

    public TextEditorMemento save() {
        return new TextEditorMemento(content);
    }

    public void restore(TextEditorMemento memento) {
        this.content = memento.getContent();
    }
}
```

◆ 3. Caretaker (History)

```
import java.util.Stack;

public class EditorHistory {
```

```

private Stack<TextEditorMemento> history = new Stack<>();

public void save(TextEditor editor) {
    history.push(editor.save());
}

public void undo(TextEditor editor) {
    if (!history.isEmpty()) {
        editor.restore(history.pop());
    }
}

```

◆ 4. Client (Demo)

```

public class MementoPatternDemo {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        EditorHistory history = new EditorHistory();

        editor.type("Hello ");
        history.save(editor); // Save 1

        editor.type("World!");
        history.save(editor); // Save 2

        editor.type(" This will be undone.");

        System.out.println("Current Content: " + editor.getContent());

        history.undo(editor);
        System.out.println("After Undo 1: " + editor.getContent());

        history.undo(editor);
        System.out.println("After Undo 2: " + editor.getContent());
    }
}

```

```
    }  
}
```

✓ Output:

Current Content: Hello World! This will be undone.

After Undo 1: Hello World!

After Undo 2: Hello



Real-World Use Cases

| Use Case | Example |
|---------------------------------------|---|
| Undo/Redo in editors | Text editors (Notepad, Word, etc.) |
| Game Save/Load | Saving a game state to resume later |
| Workflow snapshots | Save intermediate workflow state for rollback |
| Versioning | Object version history (drafts, edits) |
| Database transactions (manual) | Rollback logic where a memento stores object state before a transaction |

✓ Benefits

- Preserves encapsulation
- Easy to implement undo/rollback features
- Cleaner separation between state management and business logic

✗ Drawbacks

- Can consume a lot of memory if many states are saved
- Caretaker must manage lifecycle carefully

✓ Project Structure

```
memento-spring-boot/
├── controller/
│   └── EditorController.java
├── model/
│   └── Memento.java
├── service/
│   ├── EditorService.java
│   └── HistoryService.java
└── MementoApplication.java
```

◆ 1. **Memento.java (Model)**

```
package com.example.memento.model;

public class Memento {
    private final String content;

    public Memento(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

◆ 2. **EditorService.java (Originator)**

```
package com.example.memento.service;

import com.example.memento.model.Memento;
import org.springframework.stereotype.Service;

@Service
public class EditorService {
    private String content = "";

    public void type(String newContent) {
        content += newContent;
    }

    public Memento save() {
        return new Memento(content);
    }

    public void restore(Memento memento) {
        this.content = memento.getContent();
    }

    public String getContent() {
        return content;
    }
}
```

◆ 3. HistoryService.java (Caretaker)

```
package com.example.memento.service;

import com.example.memento.model.Memento;
import org.springframework.stereotype.Service;
```

```
import java.util.Stack;

@Service
public class HistoryService {
    private final Stack<Memento> history = new Stack<>();

    public void save(Memento memento) {
        history.push(memento);
    }

    public Memento undo() {
        return history.isEmpty() ? new Memento("") : history.pop();
    }

    public boolean hasHistory() {
        return !history.isEmpty();
    }
}
```

◆ 4. **EditorController.java**

```
import com.example.memento.controller;

import com.example.memento.service.EditorService;
import com.example.memento.service.HistoryService;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/editor")
public class EditorController {

    private final EditorService editorService;
    private final HistoryService historyService;
```

```
public EditorController(EditorService editorService, HistoryService historyService) {
    this.editorService = editorService;
    this.historyService = historyService;
}

@PostMapping("/type")
public String type(@RequestParam String text) {
    editorService.type(text);
    return "Typed: " + text;
}

@PostMapping("/save")
public String save() {
    historyService.save(editorService.save());
    return "State saved.";
}

@PostMapping("/undo")
public String undo() {
    if (historyService.hasHistory()) {
        editorService.restore(historyService.undo());
        return "Undo successful.";
    }
    return "No history to undo.";
}

@GetMapping("/content")
public String content() {
    return editorService.getContent();
}
```

◆ 5. MementoApplication.java (Main App)

```
package com.example.memento;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MementoApplication {
    public static void main(String[] args) {
        SpringApplication.run(MementoApplication.class, args);
    }
}
```

✓ Sample API Usage

1. POST /editor/type?text=Hello
2. POST /editor/save
3. POST /editor/type?text=World!
4. GET /editor/content → Hello World!
5. POST /editor/undo
6. GET /editor/content → Hello

📦 Dependencies (in pom.xml)

```
<dependencies>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>
```

✓ Summary

| Component | Pattern Role |
|------------------|--------------|
| EditorService | Originator |
| Memento | Memento |
| HistoryService | Caretaker |
| EditorController | Client / API |

7. Null State

The **Null Object Design Pattern** provides an object as a surrogate for the absence of a real object. Instead of returning `null`, return a special object that implements the expected interface but does nothing (a “do-nothing” implementation). This helps eliminate `null` checks and `NullPointerExceptions`.

✓ When to Use

- To **avoid null checks** and **reduce conditional logic**.
 - To **provide a default behavior** when no object is present.
 - To **simplify code**, especially in object-oriented designs.
-

✓ Structure

- **Abstract Class / Interface**: Declares the common operations.
 - **Real Object**: Implements actual behavior.
 - **Null Object**: Implements the same interface with empty or default behavior.
 - **Client**: Uses interface without worrying about `null`.
-

✓ Example: Customer Lookup

Imagine a system where you retrieve a `Customer` by name. If not found, instead of returning `null`, we return a `NullCustomer`.

◆ 1. Define the Interface

```
public interface Customer {  
    String getName();  
    boolean isNull();  
}
```

◆ 2. Create Real Customer Class

```
public class RealCustomer implements Customer {  
    private final String name;  
  
    public RealCustomer(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public boolean isNull() {  
        return false;  
    }  
}
```

◆ 3. Create Null Customer Class

```
public class NullCustomer implements Customer {
```

```
@Override  
public String getName() {  
    return "Not Available";  
}  
  
@Override  
public boolean isNull() {  
    return true;  
}  
}
```

◆ 4. CustomerFactory

```
public class CustomerFactory {  
    private static final String[] names = {"Alice", "Bob", "Charlie"};  
  
    public static Customer getCustomer(String name) {  
        for (String n : names) {  
            if (n.equalsIgnoreCase(name)) {  
                return new RealCustomer(name);  
            }  
        }  
        return new NullCustomer();  
    }  
}
```

◆ 5. Client Code

```
public class Main {  
    public static void main(String[] args) {  
        Customer c1 = CustomerFactory.getCustomer("Bob");  
        Customer c2 = CustomerFactory.getCustomer("Unknown");  
  
        System.out.println(c1.getName()); // Output: Bob
```

```

        System.out.println(c2.getName()); // Output: Not Available
    }
}

```

✓ Advantages of Null Object Pattern

| Benefit | Description |
|---------------------------------|--|
| Avoids <code>null</code> checks | No need to check <code>if (x != null)</code> |
| Prevents NullPointerException | Null object handles default behavior |
| Clean and maintainable code | Promotes polymorphism over conditionals |
| Improves testability | Behavior can be easily mocked or replaced |

✓ Real-world Use Cases

| Context | Use Case |
|------------------|---|
| Repositories | <code>findById()</code> returns NullObject if not found |
| Logging | NoOpLogger avoids logging if disabled |
| Strategy Pattern | Fallback behavior when no strategy matches |
| User Sessions | Return a GuestUser instead of null session |
| Spring Boot | Use Optional beans or empty object patterns |

8. `Observer`

The **Observer Design Pattern** is a **behavioral design pattern** that defines a **one-to-many dependency** between objects so that when one object (the **Subject**) changes state, all its dependents (**Observers**) are notified and updated automatically.

Key Components

| Role | Description |
|-------------------------|---|
| Subject | Maintains a list of observers and notifies them |
| Observer | Defines an updating interface |
| ConcreteSubject | Stores state and notifies observers |
| ConcreteObserver | Implements the observer interface |

Real-world Analogy

A **YouTube Channel (Subject)** notifies all its **Subscribers (Observers)** when a new video is uploaded.

Example in Java: Notification System

Let's implement a simple system where **users subscribe** to a **news agency**, and are **notified** when news is published.

◆ 1. **Observer.java (Interface)**

```
public interface Observer {  
    void update(String message);  
}
```

◆ 2. **Subject.java (Interface)**

```
public interface Subject {  
    void subscribe(Observer observer);  
    void unsubscribe(Observer observer);  
    void notifyObservers(String message);  
}
```

◆ 3. **NewsAgency.java (ConcreteSubject)**

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

public class NewsAgency implements Subject {
    private final List<Observer> observers = new ArrayList<>();

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

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

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void publishNews(String news) {
        System.out.println("Publishing news: " + news);
        notifyObservers(news);
    }
}
```

◆ 4. **Subscriber.java** (ConcreteObserver)

```
public class Subscriber implements Observer {
    private final String name;
```

```
public Subscriber(String name) {  
    this.name = name;  
}  
  
@Override  
public void update(String message) {  
    System.out.println(name + " received update: " + message);  
}  
}
```

◆ 5. Main Class

```
public class Main {  
    public static void main(String[] args) {  
        NewsAgency agency = new NewsAgency();  
  
        Observer alice = new Subscriber("Alice");  
        Observer bob = new Subscriber("Bob");  
  
        agency.subscribe(alice);  
        agency.subscribe(bob);  
  
        agency.publishNews("Breaking News: Observer pattern rocks!");  
        agency.unsubscribe(alice);  
        agency.publishNews("Update: Alice unsubscribed.");  
    }  
}
```

✓ Output

```
Breaking News: Observer pattern rocks!  
Alice received update: Breaking News: Observer pattern rocks!  
Bob received update: Breaking News: Observer pattern rocks!
```

Publishing news: Update: Alice unsubscribed.
Bob received update: Update: Alice unsubscribed.

✓ Use Cases

| Use Case | Description |
|---------------------------|--|
| Event handling systems | UI components listening to user input |
| Messaging & Notification | Email, SMS, or push notifications |
| Stock market applications | Observers watch for stock price changes |
| Chat applications | Listeners receive messages in real time |
| Spring Framework | <code>ApplicationEventPublisher</code> and listeners |

Here's how you can implement the **Observer Design Pattern using Spring Boot**, simulating a **news publishing system** where:

- Admin can publish news.
- Users (observers) are notified.

✓ Overview

We'll build:

- `Observer` interface (`Subscriber`)
- `Subject` class (`NewsAgency`)
- A simple REST API to:
 - Register subscribers
 - Publish news
 - Notify all subscribers

✓ Step-by-Step Spring Boot Setup

◆ 1. Project Structure

```
src/main/java/
└── com/example/observer
    ├── Observer.java
    ├── Subscriber.java
    ├── NewsAgency.java
    ├── NewsController.java
    └── ObserverApplication.java
```

◆ 2. **Observer.java**

```
package com.example.observer;

public interface Observer {
    void update(String message);
    String getName();
}
```

◆ 3. **Subscriber.java**

```
package com.example.observer;

public class Subscriber implements Observer {
    private final String name;

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

    @Override
    public void update(String message) {
        System.out.println("[" + name + "] received: " + message);
    }
}
```

```
@Override  
public String getName() {  
    return name;  
}  
}
```

◆ 4. NewsAgency.java

```
package com.example.observer;  
  
import org.springframework.stereotype.Component;  
  
import java.util.ArrayList;  
import java.util.List;  
  
@Component  
public class NewsAgency {  
  
    private final List<Observer> subscribers = new ArrayList<>();  
  
    public void subscribe(Observer observer) {  
        subscribers.add(observer);  
    }  
  
    public void unsubscribe(String name) {  
        subscribers.removeIf(subscriber → subscriber.getName().equalsIgnoreCase(name));  
    }  
  
    public List<String> getSubscribers() {  
        return subscribers.stream().map(Observer::getName).toList();  
    }  
  
    public void publishNews(String news) {
```

```
        for (Observer subscriber : subscribers) {
            subscriber.update(news);
        }
    }
}
```

◆ 5. NewsController.java

```
import com.example.observer;

import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/news")
public class NewsController {

    private final NewsAgency newsAgency;

    public NewsController(NewsAgency newsAgency) {
        this.newsAgency = newsAgency;
    }

    @PostMapping("/subscribe")
    public String subscribe(@RequestParam String name) {
        newsAgency.subscribe(new Subscriber(name));
        return "Subscribed: " + name;
    }

    @DeleteMapping("/unsubscribe")
    public String unsubscribe(@RequestParam String name) {
        newsAgency.unsubscribe(name);
        return "Unsubscribed: " + name;
    }
}
```

```
@PostMapping("/publish")
public String publish(@RequestParam String message) {
    newsAgency.publishNews(message);
    return "News published: " + message;
}

@GetMapping("/subscribers")
public List<String> getSubscribers() {
    return newsAgency.getSubscribers();
}
}
```

◆ 6. **ObserverApplication.java**

```
package com.example.observer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ObserverApplication {
    public static void main(String[] args) {
        SpringApplication.run(ObserverApplication.class, args);
    }
}
```

✓ Sample API Usage (Postman or cURL)

```
# Subscribe
curl -X POST "http://localhost:8080/news/subscribe?name=Alice"
curl -X POST "http://localhost:8080/news/subscribe?name=Bob"
```

```
# Publish news
curl -X POST "http://localhost:8080/news/publish?message=Breaking+New
s%21"

# Unsubscribe
curl -X DELETE "http://localhost:8080/news/unsubscribe?name=Alice"

# Get all subscribers
curl "http://localhost:8080/news/subscribers"
```

✓ Console Output (on publish)

```
[Alice] received: Breaking News!
[Bob] received: Breaking News!
```

9. State

The **State Design Pattern** is a **behavioral pattern** that allows an object to **change its behavior when its internal state changes**. It appears as if the object changed its class.

✓ Intent

Encapsulate state-based behavior and delegate to the current state object.

✓ Real-world Analogy

Consider a **Vending Machine**:

- When money is inserted, it moves to the **Ready** state.
- If selection is made, it moves to **Dispensing** state.

- If out of stock, it enters **OutOfStock** state.

Key Participants

| Role | Description |
|--------------------------|---|
| Context | Maintains an instance of a ConcreteState subclass |
| State (interface) | Defines the interface for behavior associated with a state of Context |
| ConcreteStates | Each subclass implements behavior specific to the state |

Java Example: **TrafficLight** System

We'll implement a traffic light that changes from **Red** → **Green** → **Yellow** → **Red**...

◆ 1. **State.java** (State Interface)

```
public interface State {
    void handleRequest(TrafficLight context);
    String getColor();
}
```

◆ 2. **RedState.java**

```
public class RedState implements State {
    @Override
    public void handleRequest(TrafficLight context) {
        context.setState(new GreenState());
    }

    @Override
    public String getColor() {
        return "Red";
    }
}
```

◆ 3. **GreenState.java**

```
public class GreenState implements State {  
    @Override  
    public void handleRequest(TrafficLight context) {  
        context.setState(new YellowState());  
    }  
  
    @Override  
    public String getColor() {  
        return "Green";  
    }  
}
```

◆ 4. **YellowState.java**

```
public class YellowState implements State {  
    @Override  
    public void handleRequest(TrafficLight context) {  
        context.setState(new RedState());  
    }  
  
    @Override  
    public String getColor() {  
        return "Yellow";  
    }  
}
```

◆ 5. **TrafficLight.java (Context)**

```
public class TrafficLight {  
    private State currentState;
```

```

public TrafficLight() {
    currentState = new RedState(); // initial state
}

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

public void next() {
    currentState.handleRequest(this);
}

public String getCurrentColor() {
    return currentState.getColor();
}
}

```

◆ 6. Main Class

```

public class Main {
    public static void main(String[] args) {
        TrafficLight light = new TrafficLight();

        for (int i = 0; i < 6; i++) {
            System.out.println("Light: " + light.getCurrentColor());
            light.next();
        }
    }
}

```

✓ Output

[makefile](#)
[CopyEdit](#)

Light: Red
Light: Green
Light: Yellow
Light: Red
Light: Green
Light: Yellow

✓ Use Cases of State Design Pattern

| Use Case | Description |
|---------------------------------|--|
| Workflow/Process engines | e.g., Order states: Placed → Shipped → Delivered |
| Game development | Player states: Idle, Running, Jumping, Attacking |
| UI components | Button states: Enabled, Disabled, Hovered, Clicked |
| TCP connection states | OPEN, LISTEN, CLOSED, SYN_SENT, etc. |
| ATM machine | NoCard, HasCard, Authorized, OutOfService |

✓ 1. **State.java** (interface)

```
package com.example.statepattern.states;

import com.example.statepattern.context.TrafficLightContext;

public interface State {
    void handle(TrafficLightContext context);
    String getColor();
}
```

✓ 2. Concrete States

◆ **RedState.java**

```
package com.example.statepattern.states;

import com.example.statepattern.context.TrafficLightContext;
import org.springframework.stereotype.Component;

@Component
public class RedState implements State {
    @Override
    public void handle(TrafficLightContext context) {
        context.setState(context.getGreenState());
    }

    @Override
    public String getColor() {
        return "Red";
    }
}
```

◆ **GreenState.java**

```
package com.example.statepattern.states;

import com.example.statepattern.context.TrafficLightContext;
import org.springframework.stereotype.Component;

@Component
public class GreenState implements State {
    @Override
    public void handle(TrafficLightContext context) {
        context.setState(context.getYellowState());
    }

    @Override
```

```
public String getColor() {  
    return "Green";  
}  
}
```

◆ YellowState.java

```
package com.example.statepattern.states;  
  
import com.example.statepattern.context.TrafficLightContext;  
import org.springframework.stereotype.Component;  
  
@Component  
public class YellowState implements State {  
    @Override  
    public void handle(TrafficLightContext context) {  
        context.setState(context.getRedState());  
    }  
  
    @Override  
    public String getColor() {  
        return "Yellow";  
    }  
}
```

✓ 3. TrafficLightContext.java

```
package com.example.statepattern.context;  
  
import com.example.statepattern.states.*;  
import jakarta.annotation.PostConstruct;  
import org.springframework.stereotype.Component;  
  
@Component
```

```
public class TrafficLightContext {  
  
    private final RedState redState;  
    private final GreenState greenState;  
    private final YellowState yellowState;  
  
    private State currentState;  
  
    public TrafficLightContext(RedState red, GreenState green, YellowState yellow) {  
        this.redState = red;  
        this.greenState = green;  
        this.yellowState = yellow;  
    }  
  
    @PostConstruct  
    public void init() {  
        currentState = redState; // initial state  
    }  
  
    public void next() {  
        currentState.handle(this);  
    }  
  
    public String getCurrentStateColor() {  
        return currentState.getColor();  
    }  
  
    public void setState(State state) {  
        this.currentState = state;  
    }  
  
    public RedState getRedState() { return redState; }  
    public GreenState getGreenState() { return greenState; }  
}
```

```
    public YellowState getYellowState() { return yellowState; }
}
```

✓ 4. TrafficLightController.java

```
package com.example.statepattern.controller;

import com.example.statepattern.context.TrafficLightContext;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/traffic-light")
public class TrafficLightController {

    private final TrafficLightContext context;

    public TrafficLightController(TrafficLightContext context) {
        this.context = context;
    }

    @GetMapping("/state")
    public String getCurrentState() {
        return "Current State: " + context.getCurrentStateColor();
    }

    @PostMapping("/next")
    public String goToNextState() {
        context.next();
        return "Transitioned to: " + context.getCurrentStateColor();
    }
}
```

✓ 5. StatePatternApplication.java

```
package com.example.statepattern;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StatePatternApplication {
    public static void main(String[] args) {
        SpringApplication.run(StatePatternApplication.class, args);
    }
}
```

✓ Test Using cURL or Postman

```
# Get current state
curl http://localhost:8080/traffic-light/state

# Transition to next state
curl -X POST http://localhost:8080/traffic-light/next
```

✓ Sample Output

```
GET /state → Current State: Red
POST /next → Transitioned to: Green
POST /next → Transitioned to: Yellow
POST /next → Transitioned to: Red
```

10. Strategy

The **Strategy Design Pattern** is a **behavioral pattern** used to define a **family of algorithms**, encapsulate each one, and make them interchangeable at runtime. It enables selecting an algorithm's behavior at runtime.

✓ Real-World Analogy

Think of a **navigation app**:

- You can choose between different **route strategies** like:
 - Fastest route
 - Shortest distance
 - Avoid tolls

Each strategy is encapsulated and can be changed without modifying the navigation logic.

✓ Participants

| Component | Role |
|------------------|---|
| Strategy | Interface for all supported algorithms |
| ConcreteStrategy | Implementation of the algorithm |
| Context | Uses a <code>Strategy</code> object to call the algorithm |

✓ Java Example: Payment Strategy

We'll implement a payment system where users can choose between:

- Credit Card
 - PayPal
 - UPI
-

◆ 1. `PaymentStrategy.java` (**Strategy Interface**)

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

◆ 2. Concrete Strategies

CreditCardPayment.java

```
public class CreditCardPayment implements PaymentStrategy {  
    private String cardNumber;  
  
    public CreditCardPayment(String cardNumber) {  
        this.cardNumber = cardNumber;  
    }  
  
    @Override  
    public void pay(double amount) {  
        System.out.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);  
    }  
}
```

PayPalPayment.java

```
public class PayPalPayment implements PaymentStrategy {  
    private String email;  
  
    public PayPalPayment(String email) {  
        this.email = email;  
    }  
  
    @Override  
    public void pay(double amount) {  
        System.out.println("Paid ₹" + amount + " using PayPal: " + email);  
    }  
}
```

UPIPayment.java

```
public class UPIPayment implements PaymentStrategy {  
    private String upild;  
  
    public UPIPayment(String upild) {  
        this.upild = upild;  
    }  
  
    @Override  
    public void pay(double amount) {  
        System.out.println("Paid ₹" + amount + " using UPI: " + upild);  
    }  
}
```

◆ 3. **PaymentContext.java** (Context)

```
public class PaymentContext {  
    private PaymentStrategy paymentStrategy;  
  
    public void setPaymentStrategy(PaymentStrategy strategy) {  
        this.paymentStrategy = strategy;  
    }  
  
    public void processPayment(double amount) {  
        if (paymentStrategy == null) {  
            throw new IllegalStateException("Payment strategy not set");  
        }  
        paymentStrategy.pay(amount);  
    }  
}
```

◆ 4. Main Method

```

public class Main {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-
3456"));
        context.processPayment(500.0);

        context.setPaymentStrategy(new PayPalPayment("user@example.co
m"));
        context.processPayment(250.0);

        context.setPaymentStrategy(new UPIPayment("user@upi"));
        context.processPayment(100.0);
    }
}

```

Output

Credit Card: 1234-5678-9012-3456
Paid ₹250.0 using PayPal: user@example.com
Paid ₹100.0 using UPI: user@upi

Use Cases of Strategy Pattern

| Use Case | Example |
|-------------------------------|---|
| Payment gateways | Switch between different payment processors dynamically |
| Compression algorithms | Support ZIP, RAR, TAR, etc. |
| Sorting strategies | Choose QuickSort, MergeSort, BubbleSort based on dataset size |
| Validation strategies | Apply different validation based on user type or context |
| Route selection | GPS apps: fastest vs shortest vs scenic |

| | |
|----------------------------------|--|
| Tax calculations | Different tax rules for regions or countries |
| Authentication mechanisms | JWT, OAuth2, Basic Auth, API Key strategies |

✓ Goal

Create a Spring Boot app where:

- Multiple `PaymentStrategy` implementations exist.
- A user can choose a strategy via API (e.g., PayPal, Credit Card, UPI).
- Strategy is resolved at runtime.

✓ 1. Project Structure

```
spring-strategy-demo/
├── controller/
│   └── PaymentController.java
└── strategy/
    ├── PaymentStrategy.java
    ├── CreditCardPayment.java
    ├── PayPalPayment.java
    ├── UPIPayment.java
    └── PaymentStrategyFactory.java
└── SpringStrategyDemoApplication.java
```

✓ 2. Define the Strategy Interface

```
package com.example.strategy.strategy;

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

✓ 3. Concrete Strategies

```
// strategy/PayPalPayment.java
package com.example.strategy.strategy;

import org.springframework.stereotype.Component;

@Component("paypal")
public class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using PayPal");
    }
}
```

```
// strategy/CreditCardPayment.java
package com.example.strategy.strategy;

import org.springframework.stereotype.Component;

@Component("creditcard")
public class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card");
    }
}
```

```
// strategy/UPIPayment.java
package com.example.strategy.strategy;

import org.springframework.stereotype.Component;

@Component("upi")
```

```
public class UPIPayment implements PaymentStrategy {  
    @Override  
    public void pay(double amount) {  
        System.out.println("Paid ₹" + amount + " using UPI");  
    }  
}
```

✓ 4. Strategy Factory (Auto-wired by Bean name)

```
// StrategyFactory.java  
package com.example.strategy.strategy;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
import java.util.Map;  
  
@Component  
public class PaymentStrategyFactory {  
  
    private final Map<String, PaymentStrategy> strategyMap;  
  
    @Autowired  
    public PaymentStrategyFactory(Map<String, PaymentStrategy> strategyMa  
p) {  
        this.strategyMap = strategyMap;  
    }  
  
    public PaymentStrategy getStrategy(String type) {  
        PaymentStrategy strategy = strategyMap.get(type.toLowerCase());  
        if (strategy == null) {  
            throw new IllegalArgumentException("Invalid payment type: " + type);  
        }  
        return strategy;  
    }  
}
```

```
    }  
}
```

✓ 5. REST Controller

```
// controller/PaymentController.java  
package com.example.strategy.controller;  
  
import com.example.strategy.strategy.PaymentStrategy;  
import com.example.strategy.strategy.PaymentStrategyFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/api/payment")  
public class PaymentController {  
  
    @Autowired  
    private PaymentStrategyFactory strategyFactory;  
  
    @PostMapping  
    public String pay(@RequestParam double amount, @RequestParam String  
method) {  
        PaymentStrategy strategy = strategyFactory.getStrategy(method);  
        strategy.pay(amount);  
        return "Payment of ₹" + amount + " via " + method + " completed.";  
    }  
}
```

✓ 6. Main Class

```
package com.example.strategy;  
  
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringStrategyDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringStrategyDemoApplication.class, args);
    }
}
```

✓ Run & Test

Start the app and use **Postman** or **cURL**:

```
curl -X POST "http://localhost:8080/api/payment?amount=1500&method=paypal"
```

Output (console):

```
Paid ₹1500.0 using PayPal
```

✓ Benefits of Using Strategy with Spring

- No `if-else` or `switch-case`.
- Easy to add new strategies (just `@Component`).
- Runtime selection based on REST param.
- Clean, scalable, and testable.

11. [Template](#)

Template Method Design Pattern

✓ Intent

Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses override certain steps of the algorithm without changing its structure.

✓ What it means

You have a general algorithm with fixed steps, but some steps can vary depending on the subclass. You put the **fixed sequence** in a method (the *template method*) and let subclasses provide the implementation for the variable steps.

✓ Real-world analogy

Think of **making a tea or coffee**. The sequence is:

1. Boil water
2. Brew beverage (tea leaves or coffee powder)
3. Pour into cup
4. Add condiments (milk, sugar)

Steps 1, 3 are fixed, but steps 2 and 4 vary depending on the drink.

✓ Structure

| Component | Description |
|----------------------|--|
| AbstractClass | Defines the template method (algorithm skeleton). Contains some abstract operations to be implemented by subclasses. |
| ConcreteClass | Implements the variable steps of the algorithm. |

Java Example

1. Abstract Class (Template)

```
public abstract class Beverage {  
  
    // Template method defining the sequence  
    public final void prepare() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
  
    private void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    // Abstract methods to be implemented by subclasses  
    protected abstract void brew();  
  
    private void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    protected abstract void addCondiments();  
}
```

2. Concrete Classes

```
public class Tea extends Beverage {  
  
    @Override  
    protected void brew() {  
        System.out.println("Steeping the tea leaves");  
    }  
}
```

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

```
public class Coffee extends Beverage {  
  
    @Override  
    protected void brew() {  
        System.out.println("Dripping coffee through filter");  
    }  
  
    @Override  
    protected void addCondiments() {  
        System.out.println("Adding sugar and milk");  
    }  
}
```

3. Client code

```
public class Main {  
    public static void main(String[] args) {  
        Beverage tea = new Tea();  
        tea.prepare();  
  
        System.out.println();  
  
        Beverage coffee = new Coffee();  
        coffee.prepare();  
    }  
}
```

Output

```
Boiling water
Steeping the tea leaves
Pouring into cup
Adding lemon
```

```
Boiling water
Dripping coffee through filter
Pouring into cup
Adding sugar and milk
```

✓ Use Cases for Template Method Pattern

| Use Case | Explanation |
|--|--|
| Frameworks where basic workflow is fixed | E.g., JUnit testing framework uses template methods for test lifecycle (<code>setUp</code> , <code>tearDown</code>). |
| Algorithms with invariant structure but variable steps | Sorting algorithms that share common parts but vary the pivot choice. |
| Code reuse with common algorithm parts | Base classes define the skeleton, subclasses override details. |
| Workflow engines and batch jobs | Fixed steps with flexible sub-tasks inside a process. |
| UI frameworks | Rendering lifecycle with customizable rendering steps. |
| Parsing and compiling | Generic parse method with steps overridden by different language parsers. |

Scenario

We have a service that sends notifications. The basic flow (algorithm) to send a notification is:

1. Validate the notification data
2. Prepare the message

3. Send the notification

4. Log the result

Steps 1 and 4 are fixed for all notifications. Steps 2 and 3 vary for different notification types like **Email** and **SMS**.

Step-by-step Spring Boot implementation

1. Abstract Template Service

```
package com.example.notification.service;

public abstract class NotificationService {

    // Template method
    public final void sendNotification(String to, String message) {
        validate(to, message);
        String preparedMessage = prepareMessage(message);
        send(preparedMessage, to);
        log();
    }

    private void validate(String to, String message) {
        if (to == null || to.isEmpty()) {
            throw new IllegalArgumentException("Recipient cannot be empty");
        }
        if (message == null || message.isEmpty()) {
            throw new IllegalArgumentException("Message cannot be empty");
        }
    }

    // Steps to be implemented by subclasses
    protected abstract String prepareMessage(String message);

    protected abstract void send(String message, String to);
```

```
// Fixed logging step
private void log() {
    System.out.println("Notification sent successfully");
}
}
```

2. Concrete Notification Services

```
package com.example.notification.service;

import org.springframework.stereotype.Service;

@Service("emailNotificationService")
public class EmailNotificationService extends NotificationService {

    @Override
    protected String prepareMessage(String message) {
        return "Email Content: " + message;
    }

    @Override
    protected void send(String message, String to) {
        System.out.println("Sending Email to " + to + " with message: " + message);
    }
}
```

```
package com.example.notification.service;

import org.springframework.stereotype.Service;

@Service("smsNotificationService")
public class SmsNotificationService extends NotificationService {
```

```

@Override
protected String prepareMessage(String message) {
    return "SMS Content: " + message;
}

@Override
protected void send(String message, String to) {
    System.out.println("Sending SMS to " + to + " with message: " + message);
}

```

3. Controller to choose strategy dynamically

```

package com.example.notification.controller;

import com.example.notification.service.NotificationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/notify")
public class NotificationController {

    @Autowired
    @Qualifier("emailNotificationService")
    private NotificationService emailNotificationService;

    @Autowired
    @Qualifier("smsNotificationService")
    private NotificationService smsNotificationService;

    @PostMapping

```

```

public String sendNotification(@RequestParam String type,
                               @RequestParam String to,
                               @RequestParam String message) {

    NotificationService service;

    switch (type.toLowerCase()) {
        case "email":
            service = emailNotificationService;
            break;
        case "sms":
            service = smsNotificationService;
            break;
        default:
            throw new IllegalArgumentException("Invalid notification type");
    }

    service.sendNotification(to, message);
    return "Notification sent via " + type;
}

```

4. Main Spring Boot application

```

package com.example.notification;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class NotificationTemplateApp {
    public static void main(String[] args) {
        SpringApplication.run(NotificationTemplateApp.class, args);
    }
}

```

```
    }  
}
```

How to test

Start the app and send POST requests:

```
curl -X POST "http://localhost:8080/api/notify?type=email&to=user@example.com&message=HelloEmail"
```

Output console:

```
Sending Email to user@example.com with message: Email Content: HelloEmail  
Notification sent successfully
```

```
curl -X POST "http://localhost:8080/api/notify?type=sms&to=999999999999&message=HelloSMS"
```

Output console:

```
Sending SMS to 9999999999 with message: SMS Content: HelloSMS  
Notification sent successfully
```

✓ Summary

- Template Method defines fixed steps + customizable hooks.
- Spring `@Service` components implement different variants.
- Controller dynamically picks the correct implementation based on input.

12. `Visitor`

Visitor Design Pattern

✓ Intent

Separate an algorithm from the objects on which it operates.

Visitor lets you add further operations to objects without modifying them.

✓ Why use Visitor?

- When you want to perform operations across a complex object structure (like different types in a class hierarchy).
- Avoid cluttering classes with unrelated operations.
- Add new operations easily without changing existing classes.

✓ Structure

| Component | Description |
|------------------------|--|
| Visitor | Declares visit methods for each concrete element type. |
| ConcreteVisitor | Implements operations to be performed on elements. |
| Element | Defines an accept method that takes a visitor. |
| ConcreteElement | Implements accept to call visitor's visit method. |
| ObjectStructure | Collection or complex object with elements to accept visitors. |

Java Example: Shopping Cart with different Item types

1. Element interface

```
public interface ItemElement {  
    void accept(ShoppingCartVisitor visitor);  
}
```

2. Concrete Elements (different items)

```
public class Book implements ItemElement {  
    private int price;  
    private String isbnNumber;  
  
    public Book(int price, String isbn) {  
        this.price = price;  
        this.isbnNumber = isbn;  
    }  
  
    public int getPrice() { return price; }  
    public String getIsbnNumber() { return isbnNumber; }  
  
    @Override  
    public void accept(ShoppingCartVisitor visitor) {  
        visitor.visit(this);  
    }  
}
```

```
public class Fruit implements ItemElement {  
    private int pricePerKg;  
    private int weight;  
    private String name;  
  
    public Fruit(int pricePerKg, int weight, String name) {  
        this.pricePerKg = pricePerKg;  
        this.weight = weight;  
        this.name = name;  
    }  
  
    public int getPricePerKg() { return pricePerKg; }  
    public int getWeight() { return weight; }  
    public String getName() { return name; }
```

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

3. Visitor Interface

```
public interface ShoppingCartVisitor {  
    void visit(Book book);  
    void visit(Fruit fruit);  
}
```

4. Concrete Visitor implementing operation

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {  
  
    @Override  
    public void visit(Book book) {  
        int cost = book.getPrice();  
        System.out.println("Book ISBN::" + book.getIsbnNumber() + " cost = " +  
cost);  
    }  
  
    @Override  
    public void visit(Fruit fruit) {  
        int cost = fruit.getPricePerKg() * fruit.getWeight();  
        System.out.println(fruit.getName() + " cost = " + cost);  
    }  
}
```

5. Client code

```

public class ShoppingCartClient {
    public static void main(String[] args) {
        ItemElement[] items = new ItemElement[] {
            new Book(20, "1234"),
            new Book(100, "5678"),
            new Fruit(10, 2, "Banana"),
            new Fruit(5, 5, "Apple")
        };

        int total = 0;
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();

        for (ItemElement item : items) {
            item.accept(visitor); // double dispatch
        }
    }
}

```

Output

```

Book ISBN::1234 cost = 20
Book ISBN::5678 cost = 100
Banana cost = 20
Apple cost = 25

```

Summary

- **Visitor** decouples operations from object structure.
- Uses **double dispatch**:
 1. Object calls `accept(visitor)`.
 2. Visitor calls `visit(concreteElement)`.
- Easy to add new operations without modifying elements.

Visitor Pattern Use Cases

| Use Case | Explanation |
|--|---|
| Complex object structures | When you have complex object hierarchies and want to perform operations across them without cluttering the objects. |
| Adding operations without changing classes | When you need to add new functionality frequently to unrelated classes but want to avoid modifying their code. |
| Compilers and AST traversal | Visiting different types of nodes in Abstract Syntax Trees (AST) for operations like code generation, optimization, or type checking. |
| Serialization / Deserialization | Performing different serialization strategies on diverse object types. |
| UI rendering systems | Rendering or processing different UI elements without embedding logic in the elements themselves. |
| Multiple unrelated operations | If multiple unrelated operations must be performed on a fixed set of objects, visitor helps keep those operations clean and separate. |

Visitor Pattern in Spring Boot

Scenario

Imagine a **document processing system** with different types of documents:

- **Invoice**
- **Report**

We want to:

- Generate a **summary** for each document type.
- Generate a **detailed report** for each document type.

Using the Visitor pattern, we separate these operations from the document classes themselves.

Step 1: Define Document Elements (Elements)

```
package com.example.visitor.elements;

public interface Document {
    void accept(DocumentVisitor visitor);
}
```

```
package com.example.visitor.elements;

public class Invoice implements Document {
    private double amount;
    private String invoiceNumber;

    public Invoice(double amount, String invoiceNumber) {
        this.amount = amount;
        this.invoiceNumber = invoiceNumber;
    }

    public double getAmount() { return amount; }
    public String getInvoiceNumber() { return invoiceNumber; }

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

```
package com.example.visitor.elements;

public class Report implements Document {
```

```
private String title;  
private String content;  
  
public Report(String title, String content) {  
    this.title = title;  
    this.content = content;  
}  
  
public String getTitle() { return title; }  
public String getContent() { return content; }  
  
@Override  
public void accept(DocumentVisitor visitor) {  
    visitor.visit(this);  
}  
}
```

Step 2: Define Visitor Interface

```
package com.example.visitor.visitors;  
  
import com.example.visitor.elements.Invoice;  
import com.example.visitor.elements.Report;  
  
public interface DocumentVisitor {  
    void visit(Invoice invoice);  
    void visit(Report report);  
}
```

Step 3: Concrete Visitors

```
package com.example.visitor.visitors;  
  
import com.example.visitor.elements.Invoice;
```

```

import com.example.visitor.elements.Report;
import org.springframework.stereotype.Component;

@Component("summaryVisitor")
public class SummaryVisitor implements DocumentVisitor {

    @Override
    public void visit(Invoice invoice) {
        System.out.println("Invoice Summary: Invoice #" + invoice.getInvoiceNumber() + ", Amount: $" + invoice.getAmount());
    }

    @Override
    public void visit(Report report) {
        System.out.println("Report Summary: Title - " + report.getTitle());
    }
}

```

```

package com.example.visitor.visitors;

import com.example.visitor.elements.Invoice;
import com.example.visitor.elements.Report;
import org.springframework.stereotype.Component;

@Component("detailedVisitor")
public class DetailedVisitor implements DocumentVisitor {

    @Override
    public void visit(Invoice invoice) {
        System.out.println("Invoice Detailed Report:\nInvoice Number: " + invoice.getInvoiceNumber() + "\nAmount Due: $" + invoice.getAmount());
    }

    @Override
    public void visit(Report report) {

```

```
        System.out.println("Report Detailed Content:\nTitle: " + report.getTitle() +  
        "\nContent:\n" + report.getContent());  
    }  
}
```

Step 4: Spring Boot Controller to trigger visitors

```
package com.example.visitor.controller;  
  
import com.example.visitor.elements.Document;  
import com.example.visitor.elements.Invoice;  
import com.example.visitor.elements.Report;  
import com.example.visitor.visitors.DocumentVisitor;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/documents")  
public class DocumentController {  
  
    @Autowired  
    @Qualifier("summaryVisitor")  
    private DocumentVisitor summaryVisitor;  
  
    @Autowired  
    @Qualifier("detailedVisitor")  
    private DocumentVisitor detailedVisitor;  
  
    @GetMapping("/process")  
    public String processDocument(@RequestParam String type,  
                                 @RequestParam String visitorType) {  
  
        Document document;
```

```

// Create sample documents based on type
if ("invoice".equalsIgnoreCase(type)) {
    document = new Invoice(1500.75, "INV-123");
} else if ("report".equalsIgnoreCase(type)) {
    document = new Report("Annual Report", "This is the detailed content
of the annual report...");
} else {
    return "Invalid document type";
}

// Choose visitor
DocumentVisitor visitor;
if ("summary".equalsIgnoreCase(visitorType)) {
    visitor = summaryVisitor;
} else if ("detailed".equalsIgnoreCase(visitorType)) {
    visitor = detailedVisitor;
} else {
    return "Invalid visitor type";
}

// Accept visitor
document.accept(visitor);

return "Processed " + type + " with " + visitorType + " visitor";
}
}

```

Step 5: Main Spring Boot Application

```

package com.example.visitor;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

```

```
public class VisitorPatternApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(VisitorPatternApplication.class, args);  
    }  
}
```

Testing the Visitor in Spring Boot

- Start the app
- Call:

```
GET http://localhost:8080/documents/process?type=invoice&visitorType=summary
```

Console output:

```
Invoice Summary: Invoice #INV-123, Amount: $1500.75
```

```
GET http://localhost:8080/documents/process?type=report&visitorType=detailed
```

Console output:

```
Report Detailed Content:  
Title: Annual Report  
Content:  
This is the detailed content of the annual report...
```

Summary

- Visitor separates operations from document object structure.
- Spring beans implement different visitor strategies.

- Controller selects and applies visitors dynamically.
 - Easy to add new operations by creating new visitors without modifying documents.
-
-