Mediator Design Pattern

The **Mediator Design Pattern** is a **behavioral pattern** that promotes **loose coupling** by preventing objects from referring to each other explicitly. Instead, they communicate via a **mediator object**.

🧠 Definition

Mediator defines an object that **encapsulates how a set of objects interact**. This helps:

- Reduce dependencies between communicating objects (colleagues).
- Centralize complex communication logic.

🧾 Real-life Analogy: Air Traffic Control

Planes (colleagues) do **not communicate with each other directly**. Instead, they communicate through **ATC (Mediator)**. This avoids collision and chaos.

✅ Participants

Role	Description
Mediator	Interface to coordinate communication between objects
ConcreteMediator	Implements communication logic
Colleague	Objects that communicate via the mediator

✅ Java Code Example — Chat Room

Step 1: Mediator Interface

```
public interface ChatMediator {
    void showMessage(String message, User user);
}
```

Step 2: Concrete Mediator

```
public class ChatRoom implements ChatMediator {
    @Override
    public void showMessage(String message, User user) {
        System.out.println("[" + user.getName() + "] : " + message);
    }
}
```

Step 3: Colleague (User)

```
public class User {
    private String name;
    private ChatMediator chatMediator;
    public User(String name, ChatMediator mediator) {
        this.name = name;
        this.chatMediator = mediator;
    }

    public String getName() {
        return name;
    }

    public void send(String message) {
        chatMediator.showMessage(message, this);
    }
}
```

Step 4: Client Code

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        ChatMediator mediator = new ChatRoom();

        User rajeev = new User("Rajeev", mediator);
        User farheen = new User("Farheen", mediator);

        rajeev.send("Hello!");
        farheen.send("Hi! How are you?");
    }
}
```

✅ Use Cases

Use Case	Description
Chat systems	Mediator routes messages between users
UI components	Mediator manages complex interactions between UI widgets
Aircraft control systems	ATC handles coordination between airplanes
Workflow engines	Mediator directs the flow between services/tasks

---

✅ Java (JDK) Internal Example

1. `java.util.Timer` & `TimerTask`

- `Timer` acts like a **mediator** that schedules `TimerTask` executions.
- The `TimerTask` does not control execution — it's triggered by `Timer`.

2. `ExecutorService`

- It mediates between a **thread pool** and submitted `Runnable`/`Callable` tasks.
- `ExecutorService` is the **coordinator**; threads and tasks don't manage each other directly.

✅ Spring Boot Mediator Pattern Implementation

✅ Scenario: Event-Driven Workflow

Let’s build a mediator that handles various **user actions** (register, notify, log), without each service knowing about the others.

---

1. Define an Interface

```
public interface UserActionHandler {
    void handle(String username);
}
```

2. Implement Different Services (Colleagues)

```
@Component
public class EmailNotificationHandler implements UserActionHandler {
    public void handle(String username) {
        System.out.println("Sending email to " + username);
    }
}

@Component
public class AuditLogHandler implements UserActionHandler {
    public void handle(String username) {
        System.out.println("Logging action for " + username);
    }
}
```

3. Create Mediator

```
@Component
public class UserActionMediator {

    private final List<UserActionHandler> handlers;

    @Autowired
    public UserActionMediator(List<UserActionHandler> handlers) {
        this.handlers = handlers;
    }

    public void mediate(String username) {
        handlers.forEach(handler -> handler.handle(username));
    }
}
```

4. REST Controller

```
@RestController
public class UserController {

    @Autowired
    private UserActionMediator mediator;

    @PostMapping("/register")
    public String registerUser(@RequestParam String username) {
        // Simulate user registration
        System.out.println("User " + username + " registered.");
        mediator.mediate(username); // Notify services
        return "User registered!";
    }
}
```

✅ Why Use Mediator?

Without Mediator	With Mediator
Colleagues communicate directly	Centralized communication logic
Hard to scale or extend	Add new handler easily
High coupling between services	Decoupled, testable components

---

✅ Comparison with Related Patterns

Pattern	Key Difference
Observer	One-to-many; loosely coupled notification
Chain of Responsibility	Pass request along chain till one handles
Command	Encapsulate request as object
Facade	Simplifies interface; doesn’t mediate communication

✅ Summary

- **Mediator centralizes communication** between components (like UI widgets or services).
- Used in **chat**, **UI systems**, **workflow engines**, and **Spring Boot** applications with multiple decoupled services.
- In Spring, you often **combine Mediator with DI (via `@Autowired List<Interface>`)**, and optionally with Events (`ApplicationEventPublisher`) for broader decoupling.