25 Behavioral: Mediator Pattern — Chat-Room Assignment

Design a mini chat-room where **users never talk directly**; every message flows through a **Mediator**.  
 All classes must contain concise JavaDoc, and you’ll finish with a short **Reflection** on pros/cons.

.

├── analysis

│ └── mediator\_need.md ← why mediator > direct references

├── src/main/java

│ └── chat

│ ├── mediator

│ │ ├── ChatMediator.java

│ │ └── ChatRoom.java

│ ├── colleague

│ │ ├── User.java

│ │ └── ChatUser.java

│ └── MediatorDemo.java

├── src/test/java/chat

│ ├── BroadcastTest.java

│ ├── JoinLaterTest.java

│ └── SelfEchoTest.java

├── reflection.md

└── README.md

#### **1 Mediator interface**

package chat.mediator;

import chat.colleague.User;

/\*\*

\* Mediator contract for the chat domain.

\* Implementations route messages and maintain the list of participants.

\*/

public interface ChatMediator {

/\*\* Broadcasts {@code message} from {@code sender} to all other users. \*/

void send(String message, User sender);

/\*\* Registers a new {@link User} with the chat room. \*/

void add(User user);

}

#### **2 Concrete mediator**

package chat.mediator;

import chat.colleague.User;

import java.util.ArrayList;

import java.util.List;

/\*\* Concrete mediator that naïvely broadcasts to every user except sender. \*/

public class ChatRoom implements ChatMediator {

private final List<User> participants = new ArrayList<>();

@Override public void send(String msg, User sender) {

for (User u : participants) if (u != sender) u.receive(msg, sender);

}

@Override public void add(User user) { participants.add(user); }

}

#### **3 Abstract colleague**

package chat.colleague;

import chat.mediator.ChatMediator;

/\*\*

\* Participant of a chat room. All communication is delegated to the mediator.

\*/

public abstract class User {

protected final ChatMediator room;

protected final String name;

protected User(ChatMediator room, String name){

this.room = room;

this.name = name;

}

/\*\* Sends a text message to the room. \*/

public abstract void send(String message);

/\*\* Called by the mediator when another user sends a message. \*/

public abstract void receive(String message, User from);

@Override public String toString(){ return name; }

}

#### **4 Concrete colleague**

package chat.colleague;

import chat.mediator.ChatMediator;

/\*\* Basic console-printing user. \*/

public class ChatUser extends User {

public ChatUser(ChatMediator room, String name){ super(room, name); }

@Override public void send(String message){

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

room.send(message, this);

}

@Override public void receive(String message, User from){

System.out.printf("%s receives from %s: %s%n", name, from, message);

}

}

#### **5 Demo client**

package chat;

import chat.colleague.\*; import chat.mediator.\*;

public class MediatorDemo {

public static void main(String[] args){

ChatMediator room = new ChatRoom();

User alice = new ChatUser(room, "Alice");

User bob = new ChatUser(room, "Bob");

User chad = new ChatUser(room, "Chad");

room.add(alice); room.add(bob); room.add(chad);

alice.send("Hello everyone!");

bob.send("Hi Alice!");

}

}

Output

Alice sends: Hello everyone!

Bob receives from Alice: Hello everyone!

Chad receives from Alice: Hello everyone!

Bob sends: Hi Alice!

Alice receives from Bob: Hi Alice!

Chad receives from Bob: Hi Alice!

#### **6 Tests (outline)**

/\* BroadcastTest.java \*/

ChatMediator room = new ChatRoom();

StringBuilder log = new StringBuilder();

User a = new ChatUser(room,"A"){ @Override public void receive(String m,User f){log.append(m);} };

User b = new ChatUser(room,"B");

room.add(a); room.add(b);

b.send("ping");

assertEquals("ping", log.toString());

/\* JoinLaterTest.java \*/

User c = new ChatUser(room,"C");

room.add(c); // after first message

a.send("second");

## **reflection.md**

Using Mediator eliminates the **n² web** of references each time a new user joins – only the room knows everyone.  
 *Pros*

* **Loose coupling** – users depend on the mediator interface only.
* **Single place for rules** – muting, filters, private groups can be added inside ChatRoom without touching users.
* **Scalability** – new user types simply implement receive/send.

*Cons*

* **God-object risk** – ChatRoom might balloon with moderation, persistence, presence, etc.; split into smaller mediators if that happens.
* **Latency/overhead** – indirect call vs direct method call is negligible here but matters for high-frequency systems.

For chat-style apps with many peers and evolving policies, Mediator provides a clean, maintainable structure.