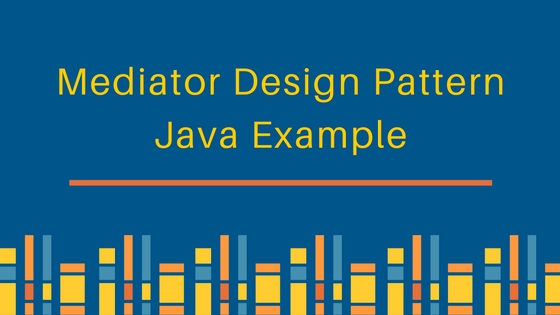
**Mediator Design Pattern**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/mediator-design-pattern.jpg)

According to GoF, mediator pattern intent is:

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it’s own logic to provide way of communication.

The system objects that communicate each other are called Colleagues. Usually we have an [interface or abstract class](https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java) that provides the contract for communication and then we have concrete implementation of mediators.

For our example, we will try to implement a chat application where users can do group chat. Every user will be identified by it’s name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.

**Mediator Pattern Interface**

First of all we will create Mediator interface that will define the contract for concrete mediators.

ChatMediator.java

interface ChatMediator {

public void sendMessage(String msg, User user);

void addUser(User user);

}

**Mediator Pattern Colleague Interface**

Users can send and receive messages, so we can have User interface or abstract class. I am creating User as abstract class like below.

User.java

public abstract class User {

protected ChatMediator mediator;

protected String name;

public User(ChatMediator med, String name){

this.mediator=med;

this.name=name;

}

public abstract void send(String msg);

public abstract void receive(String msg);

}

Notice that User has a reference to the mediator object, it’s required for the communication between different users.

**Concrete Mediator**

Now we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users.

ChatMediatorImpl.java

import java.util.ArrayList;

import java.util.List;

public class ChatMediatorImpl implements ChatMediator {

private List<User> users;

public ChatMediatorImpl(){

this.users=new ArrayList<>();

}

@Override

public void addUser(User user){

this.users.add(user);

}

@Override

public void sendMessage(String msg, User user) {

for(User u : this.users){

//message should not be received by the user sending it

if(u != user){

u.receive(msg);

}

}

}

}

**Mediator Design Pattern Concrete Colleague**

Now we can create concrete User classes to be used by client system.

UserImpl.java

public class UserImpl extends User {

public UserImpl(ChatMediator med, String name) {

super(med, name);

}

@Override

public void send(String msg){

System.out.println(this.name+": Sending Message="+msg);

mediator.sendMessage(msg, this);

}

@Override

public void receive(String msg) {

System.out.println(this.name+": Received Message:"+msg);

}

}

Notice that send() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator.

**Mediator Pattern Example Client Program Code**

Let’s test this our chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message.

ChatClient.java

public class ChatClient {

public static void main(String[] args) {

ChatMediator mediator = new ChatMediatorImpl();

User user1 = new UserImpl(mediator, "Pankaj");

User user2 = new UserImpl(mediator, "Lisa");

User user3 = new UserImpl(mediator, "Saurabh");

User user4 = new UserImpl(mediator, "David");

mediator.addUser(user1);

mediator.addUser(user2);

mediator.addUser(user3);

mediator.addUser(user4);

user1.send("Hi All");

}

}

Notice that client program is very simple and it has no idea how the message is getting handled and if mediator is getting user or not.

Output of the mediator pattern example program is:

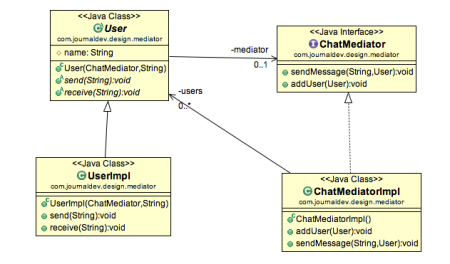
Pankaj: Sending Message=Hi All

Lisa: Received Message:Hi All

Saurabh: Received Message:Hi All

David: Received Message:Hi All

**Mediator Pattern Class Diagram**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/mediator-pattern.png)

**Mediator Pattern Example in JDK**

* [java.util.Timer](https://www.journaldev.com/1050/java-timer-timertask-example) class scheduleXXX() methods
* [Java Concurrency Executor](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) execute() method.
* java.lang.reflect.Method invoke() method.

**Mediator Design Pattern Important Points**

* Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
* Java Message Service (JMS) uses Mediator pattern along with [**Observer pattern**](https://www.journaldev.com/1739/observer-design-pattern-in-java) to allow applications to subscribe and publish data to other applications.
* We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

That’s all for mediator design pattern and it’s implementation in java.