7 Mediator

The mediator pattern enables the flexible interaction of several objects/classes that may not know each other. The mediation function can be varied very flexibly, but can also lead to very extensive implementations. In contrast to the Observer or the Chain of Responsibility, the Mediator also has its own tasks to perform. This gives him a central role. However, there is also the danger that it becomes very extensive and even confusing.

7.1 Distinction from the Observer Pattern

To start with, I would like to contrast Mediator and Observer. The purpose of Observer is to relate an event source to any number of observers. The relationship was uni- or bidirectional: The event source knows the interface of the observers and maybe vice versa. However, it becomes problematic when each observer also wants to be an event source for every other observer. This might be the case, for example, if you are programming a chat. There are an unspecified number of participants, all of whom are interested in the events – messages – of the others. If you were to program a chat room using the Observer Pattern, each participant would have to keep a list of all the other participants. New participants – in the Mediator Pattern, we talk about colleagues – would have to register with all existing participants and in turn reference all existing participants. Extrapolate how many relationships you have with – say – 33 participants, i.e. colleagues. You need a different approach for that.

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_7

7.2 Task of the Mediator Pattern

If 33 colleagues all knew each other, you would have an inefficient spider web of over 1000 relationships. The mediator now has the task of decoupling event sources and observers. He mediates between the participants.

The mediator is scalable. I would like to create the following scenario as an illustration:

You can have a class in your program that manages the orders of an online shop; but instead of, or even in addition to, this class, the "Orders" unit can also be a whole department of employees who enter additional orders in the telephone service. The "Orders" unit reports a new order for processing to the mediator – which can again be either a single class or a department of employees.

The mediator asks customer management – which again can be a class or a whole department – whether the customer is already registered. If not, the mediator asks customer management to create a new record; if the customer is already on file, the mediator asks accounting (you know: class or department) whether the customer has reliably paid his bills. Depending on the accounting department's answer, the mediator instructs the shipping unit to ship the goods either by invoice or by cash on delivery. Now, if 1 day there is a need to add another unit to the system, only the mediator needs to be adjusted. Let the case arise that the head department wants to be informed of all purchases above a certain sales value. All you have to do is create a message in the mediator; the other units would not know anything about it.

7.3 Mediator in Action – An Example

The following simulation shows you the mediator in action. You have one unit that produces wine; this unit is a simplified producer, in reality winemakers, cooper and the glycol industry are behind this unit. Another unit represents costumers or retailers, the consumers. Consumers buy wine at the respective low price. Since there are x consumers and y producers, the dependencies are reduced to the middleman, the mediator. The mediator receives the consumer's requests and forwards them to the producers. Finally, the producers tell the mediator what price they want; the mediator forwards this information to the consumers. In addition to a method for negotiating the price, the mediator needs other methods for registering and unregistering consumers and producers. The sample code can be found in the WineSim project.

7.3.1 Definition of a Consumer

A consumer must have a method with which it can register with the mediator; I will dispense with the possibility of de-registering so as not to inflate the example unnecessarily.

In any case, the consumer must have a method to request wine. This method is given the quantity of units – for example bottles:

```
public interface ConsumerIF {
    double requestPrice(int quantity);

    void register(MediatorIF mediator);
}
```

A consumer registers with the mediator and submits its request to the mediator.

```
public class PrivateCustomer implements ConsumerIF {
    private final String name;

    private mediatorIF mediator;


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

    // … abridged


    @Override
    public double requestPrice(int quantity) {
        System.out.
        println(name + " requests " + Quantity
            + " bottles of wine.");
        double totalPrice = mediator.getQuote(units);
        return totalPrice;
    }
}
```

In the next section, let's take a look at the producers of the wine.

7.3.2 Definition of a Producer

Producers must provide two methods – one for registering with the mediator and one for the mediator to address his requests to.

```
public interface ProducerIF {
    double getQuote(int quantity);
    void register(MediatorIF mediator);
}
```

A producer calculates the price for a unit on the basis of a random number and depending on the requested number of units and communicates this price to the mediator.

```
public class ProducerImpl implements ProducerIF {
    private mediatorIF mediator;
    private final String name;

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

    @Override
    public double getQuote(int quantity) {
        double discountFactor = 1.0;
        if (quantity > 100)
            discountFactor = 0.7;
        else if (units > 50)
            discountFactor = 0.8;
        else
            discountFactor = 0.9;
        double price = Math.random() * 9 + 1;
        price *= discountFactor;
        String strPrice =
        NumberFormat.getCurrencyInstance().format(price);
        System.out.
            println(″Producer ″ + name + ″ asks ″
                + strPrice + ″ per bottle.“);
        return price * units;
    }
}
```

Note that here, although the producer quotes the price per bottle (outputs it to the console) when submitting an offer, he then returns the total price to his caller. Of course, the mediator has to take this into account in his calculation. We'll deal with that in the next section.

7.3.3 Mediator Interface

The methods of the mediator are declared in the interface `MediatorIF`.

```
public interface MediatorIF {
    double offerDetermine(int units);
    void addProducer(ProducerIF producer);
    void removeProcuder(ProducerIF producer);
    void addConsumer(ConsumerIF consumer);
    void removeConsumer(ConsumerIF consumer);
}
```

A specific mediator, for example a wholesaler, must implement these methods.

```
public class Wholesale implements MediatorIF {
    private final List<ProducerIF> producers =
                                   new ArrayList<>();
    private final List<ConsumerIF> consumers =
                                   new ArrayList<>();

    // … abridged
    @Override
    public double getQuote(int quantity) {
        List<Double> quotes = new ArrayList<>();

        for (ProducerIF tempProducer : producers) {
          Double quote = tempProducer.getQuote(quantity);
            quotes.add(quote);
        }
        var price = Collections.min(offers);
        var strPrice =
            NumberFormat.getCurrencyInstance().
                format(price);
        System.out.println("The best offer for "
                + quantity
                + " bottles is: " + strPrice);
        return price;
    }
}
```

In the example, the list of consumers is not currently used by the mediator. But you can think about what you need to add if a producer wants to use the mediator to request the addresses of customers to send an advertisement (assuming the customers have given their consent). Now we can test the project.

7.3.4 Testing the Mediator Pattern

How can you test the interaction of the classes? You create any number of consumers and any number of producers and register them all with the mediator. Then you let the consumers request the price for a certain number of bottles. The mediator asks all the producers for quotes, determines the cheapest one, and forwards it to the consumers.

```
public class Test class {
    public static void main(String[] args) {
        // create a mediator
        MediatorIF wholesale = new wholesale();

        // create two customers
    ConsumerIF jim = new PrivateCustomer(″Jim Collins″);
    ConsumerIF jack = new PrivateCustomer(″Jack Meyers″);

        // create three suppliers
        ProducerIF vineyard_1 =
                        new ProducerImpl(″Vineyard 1″);
        ProducerIF vineyard_2 =
                        new ProducerImpl(″Vineyard 2″);
        ProducerIF vineyard_3 =
                        new ProducerImpl(″Vineyard 3″);

        // register with the mediator
        jim.register(wholesale);
        jack.register(wholesale);
        vineyard_1.register(wholesale);
        vineyard_2.register(wholesale);
        vineyard_3.register(wholesale);

        // Generate enquiries and obtain quotations
        int quantity = 50;
        double price = jim.requestPrice(quantity);
        System.out.println(″\n‟);


        unit = 10;
        price = jack.requestPrice(quantity);
    }
}
```

This leads to the following output (with random prices, of course):

```
Jim Collins requests 50 bottles of wine.
Producer Vineyard 1 asks 6,69 € per bottle.
Producer Vineyard 2 asks 4,52 € per bottle.
Producer Vineyard 3 asks 5,75 € per bottle.
The best offer for 50 bottles is: 225,76 €

Jack Meyers requests 10 bottles of wine.
Producer Vineyard 1 asks 7,65 € per bottle.
Producer Vineyard 2 asks 2,42 € per bottle.
Producer Vineyard 3 asks 1,73 € per bottle.
The best offer for 10 bottles is: 17,30 €
```

7.4 Mediator in Action – The Second Example

The mediator pattern is also very common in the implementation of user interfaces. In the model-view-controller model, the controller is precisely the mediator that mediates between the view/use and the abstract model of an application. Let’s look at an example of this.

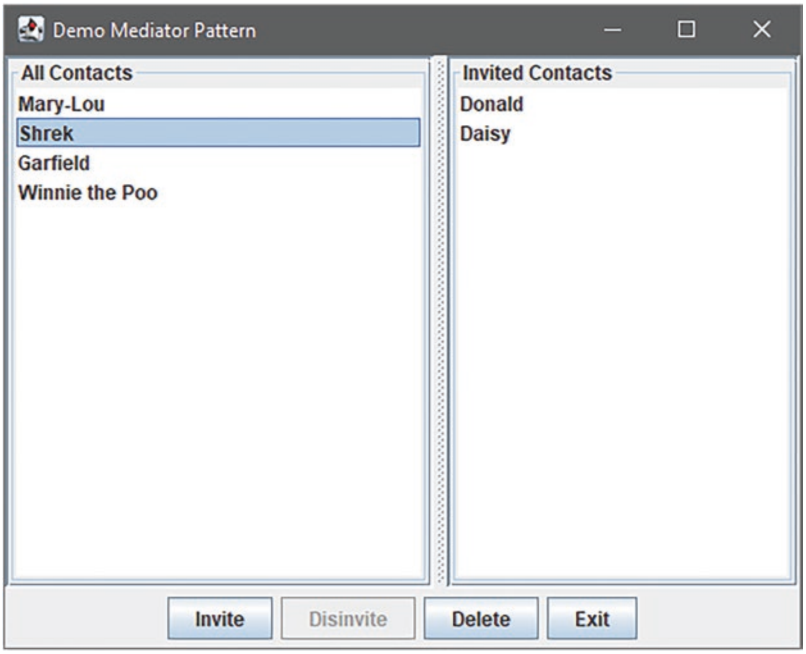

Fig. 7.1 GUI with Mediator from the SwingExample example

7.4.1 Mediator in GUI Programming

You have a GUI that shows two lists. The left list shows all contacts in your address book; the right lists all contacts who will be invited to the next party. For this example, we need four buttons. One button moves a contact to the list of invited contacts, another button moves a contact back to the general list of all contacts. A third button deletes a selected contact and the fourth button exits the program. The screenshot in Fig. 7.1 shows how the finished program should look like. The source code can be found in the project SwingExample.

Various dependencies are to be defined:

- The Exit button is always activated, regardless of the dependencies described below. If you click on it, the program is terminated.
- If a contact is marked on the right list, the button Do not invite is activated. Clicking this button moves the contact to the left list.
- If a contact is selected on the left list, the Invite button is activated.
- When the user clicks on this button, the selected contact is moved to the right list.
- Only one contact can be selected in both lists.
- The Delete button is activated when an entry is selected in one of the lists.
- If no contact is selected – neither in the right nor in the left list – no button – except Exit – is activated.
- When a contact has been moved or deleted, all markers are cleared and all buttons are disabled except for the Exit button.

7.4.2 Structure of the GUI

I print the source code of the project only in abbreviated form and limit myself to the essential key points. Please analyze the source code further on your own. The two lists are instances of the class `JList`. The four buttons are instances of the class `JButton`. All components register with an instance of the class Mediator, which is the core of the project. It stores references to all components involved. It also defines methods that are called when an event is fired. Let's look at the flow using the Invite button as an example. When the Invite button is activated, the Invite`()` method of the Mediator is called. Within this class, all the buttons – except for the exit button – are first disabled. Then the method gets the models of the two `JList instances` and moves the selected entry into the list of invited contacts.

```
class Mediator {
    private JButton btnInvite;
    private JButton btnDisinvite;
    private JButton btnDelete;
```

```
        private JList allContactsList;
        private JList invitedContactsList;

        // … abridged
        void invite() {
            btnInvite.setEnabled(false);
            btnDelete.setEnabled(false);
            btnDisinvite.setEnabled(false);

            var selectedItem =
                (String) allContactsList.getSelectedValue();

            var tempModel = allContactsList.getModel();
            var allContactsModel =
                (AllContactsModel) tempModel;

            tempModel = invitedContactsList.getModel();
            var invitedContactsModel =
                (InvitedContactsModel) tempModel;

            allContactsModel.removeData(selectedItem);
            invitedContactsModel.addData(selectedItem);

            allContactsList.clearSelection();
            invitedContactsList.clearSelection();
        }
    }
```

The Invite button is deactivated after it is created and is registered with the Mediator. The ActionListener passed to the button provides for the Invite`()` method of the Mediator to be called when the button is activated.

```
    public class GUI {
        private final JFrame frmMain = new JFrame();

        public GUI() throws Exception {
            // … abridged
            var btnInvite = new JButton("Invite");
            btnInvite.setEnabled(false);
            mediator.registerInviteButton(btnInvite);
            btnInvite.addActionListener((ActionEvent e) -> {
                mediator.invite();
            });
        }
    }
```

Note that I've used lambda notation here. This saves me from typing ...`new ActionListener` and `@Override public void actionPerformed` ... and makes the code much easier to read.

In this way, all components are created and positioned on the GUI. Each component registers with the Mediator and calls a specific method there.

7.5 Mediator – The UML Diagram

For the example project WeinSim you can see the UML diagram in Fig. 7.2.

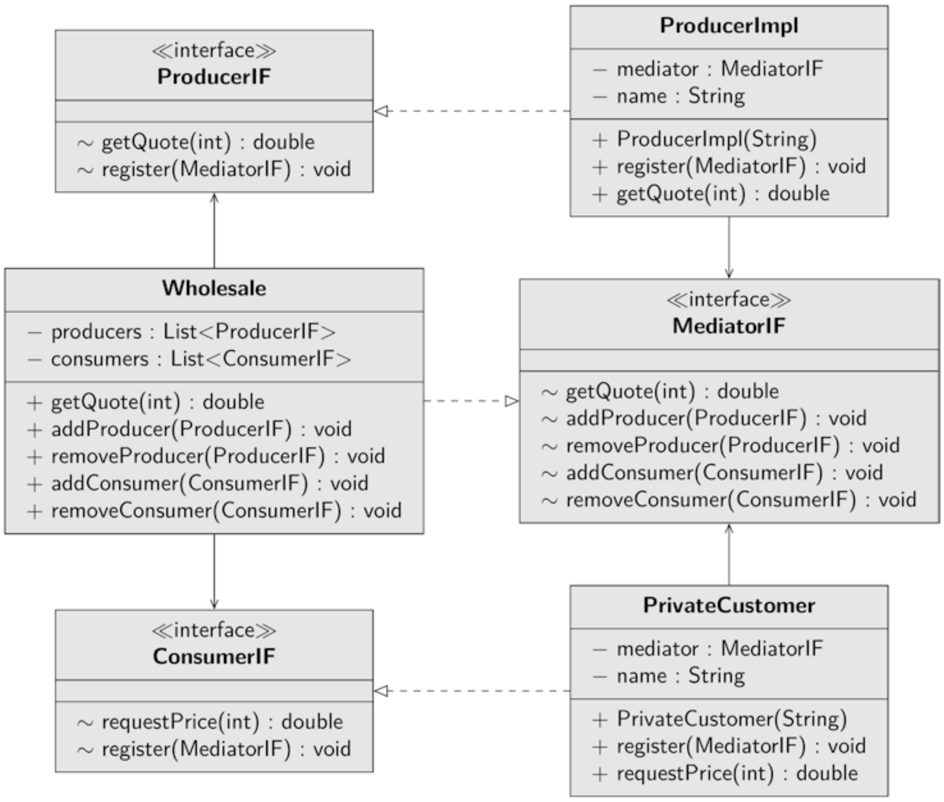
≪interface≫
ProducerIF
∼ getQuote(int) : double
∼ register(MediatorIF) : void
ProducerImpl
− mediator : MediatorIF
− name : String
+ ProducerImpl(String)
+ register(MediatorIF) : void
+ getQuote(int) : double
Wholesale
− producers : List<ProducerIF>
− consumers : List<ConsumerIF>
+ getQuote(int) : double
+ addProducer(ProducerIF) : void
+ removeProducer(ProducerIF) : void
+ addConsumer(ConsumerIF) : void
+ removeConsumer(ConsumerIF) : void
≪interface≫
MediatorIF
∼ getQuote(int) : double
∼ addProducer(ProducerIF) : void
∼ removeProducer(ProducerIF) : void
∼ addConsumer(ConsumerIF) : void
∼ removeConsumer(ConsumerIF) : void
≪interface≫
ConsumerIF
∼ requestPrice(int) : double
∼ register(MediatorIF) : void
PrivateCustomer
− mediator : MediatorIF
− name : String
+ PrivateCustomer(String)
+ register(MediatorIF) : void
+ requestPrice(int) : double

Fig. 7.2 UML diagram of the Mediator Pattern (example project WeinSim)

7.6 Criticism of Mediator

Let's take a critical look at the project SwingExample. What stands out? On the one hand, as with the wine trading simulation, you can see the advantage that the colleague classes involved don't know each other – and don't need to know each other. They are loosely coupled, which is always an indication of a good design. You can insert new colleagues at any time – the changes to the source code are limited to the mediator.

On the other hand, the disadvantage should not be concealed. If you analyze the Mediator class, you will find a relatively large class. And this is exactly the danger of the mediator: you risk developing a "god class"[1] that grows rapidly and becomes confusing as the number of components involved increases.

7.7 Summary

Go through the chapter again in key words:

- The Mediator Pattern loosens the binding between objects.
- Many participants – colleagues – depend on each other.
- Colleagues don't know each other.
- The mediator mediates between the colleagues.
- The mediator carries out its own tasks and can become very extensive.

7.8 Description of Purpose

The Gang of Four describes the purpose of the "Mediator" pattern as follows:

> Define an object that encapsulates the interaction of a set of objects within itself. Mediators promote loose coupling by preventing objects from explicitly referring to each other. They allow you to vary the interaction of objects independently of them.

[1] "God-class" is what it's called because only the good Lord still keeps track of the code.