Chain of Responsibility

6

The Chain of Responsibility is a behavior pattern. Again, it involves sending a message to a large number of objects. Remember – with the Observer pattern, you sent a message to all objects that are registered as listeners. Using the chain of responsibility, a message is passed down a chain of objects; the first object that can process that message gets the responsibility. It is also possible for a message to be processed by several objects in succession, as well as for the message to be modified on its way through the chain.

6.1 A Real World Example

Imagine you want to apply for a driver's license. So you go to the first room in the corridor at the citizen center. There you ask whether you can apply for a driving license. The official in the first room does not process driving licenses, but hunting licenses, so he is not responsible; therefore he sends you to the second room. There, too, you ask and either get help or are sent one room over. At some point you will meet an officer who is responsible for driving licenses and he will help you.

This already makes one feature of the pattern clear: you have a system of objects that could **all** potentially process the message.

6.2 First Code Example: Grocery Shopping

Your vision of the future is taking shape: You have a refrigerator with an RFID receiver. The fill level of your beer bottles is checked regularly. Even if the number of slices of sausage and cheese drops, it will be registered. And if eggs and bread are running low, the

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_6

fridge will know that too. Of course, since you're a motivated programmer, you don't have time to go to the supermarket to shop yourself. Fortunately, stores in the area have banded together to form an electronic supply chain. When refrigerators report that a customer is running low on a product, each of the retailers checks to see if he can bring the customer new food. If not, they pass the order on to the next possible retailer.

6.2.1 The Foodstuffs Required

In the first step, you define which foods are to be supplied by the electronic supply chain. In the example project CoR_1 this should be bread, cheese, sausage, eggs and beer. You define these goods in an enumeration.

```
public enum Ware {
    CHEESE("Cheese"), SAUSAGES("Sausages"),
    EGGS("Eggs"), BEER("Beer"), BREAD("Bread");

    private final String description;

    private Ware(String description) {
        this.description = description;
    }
    @Override
    public String toString() {
        return description;
    }
}
```

In this project, the job, the "problem", is passed through a variable from the enumeration. Do you remember the Observer pattern? There you used a listener and an event object. This principle also fits with the Chain of Responsibility: you create an event object and pass it to the first link in the chain. This link decides whether it can process the event; if not, the event is forwarded. I will show you below that the Java class library does exactly this.

In the next section you will get to know the sellers of the goods.

6.2.2 The Sellers

As traders I define the beverages store, the bakery and the farm shop of Old McDonald, where you can get fresh cheese, sausages and eggs. All merchants must have identical methods in addition to their actual tasks (purchasing, manufacturing, etc.): sell and pass on the order if it cannot be served. So you define an abstract superclass that holds a reference

to the next merchant in the chain. You call the `setNext()` method when you want to add another merchant to the chain. Each dealer first checks whether it itself already references a following dealer. If so, it will not store the new merchant, but pass it to the next merchant, which will now check if it is the last one in the chain. If so, it will save the new merchant as the following merchant. In addition, each trader must also provide a `sell()` method – you declare this in the superclass, but don't define it yet.

```
public abstract class AbstractMerchant {
    private AbstractMerchant next;

    public abstract void sell(Groceries article);

    public void setNext(AbstractMerchant merchant) {
        if (next == null)
            next = merchant;
        else
            next.setNext(merchant);
    }

    protected void forward(Groceries article) {
        if (next != null)
            next.sell(article);
    }
}
```

As an example of a specific merchant, I am only printing the bakery here. The other merchants can be found in the sample project. The `sell()` method is representative of an arbitrarily complex problem solution. However, the principle behind it is always the same: If I can solve the problem, I solve it, otherwise I pass it on.

```
public class Bakery extends AbstractMerchant {
    // … abridged

    @Override
    public void sell(Groceries article) {
        if (article == Groceries.BREAD)
          System.out.println(name + " sells " + article);
        else
            forward(article);
    }
}
```

Food is defined, the dealers are ready. The test class is still missing.

6.2.3 The Client

The main method of the test class is the client that demonstrates how to handle the chain. First, you create any number of merchants and link them:

```
public class Testclass {
    public static void main(String[] args) {
        var farmshop = new FarmShop("Old McDonald's");
        var bakery = new Bakery("Ben's Bakery");
        var liquorstore =
                new LiquorStore("BeeBee's Beer & Wine");
        bakery.setNext(liquorstore);
        liquorstore.setNext(farmshop);
        bakery.sell(Groceries.CHEESE);
        farmshop.sell(Groceries.EGGS);
        bakery.sell(Groceries.SAUSAGES);
        bakery.sell(Groceries.BREAD);
        liquorstore.sell(Groceries.BEER);
    }
}
```

Now the client can pass a purchase order to the first link in the chain. The order is now passed on to a dealer until one recognizes his responsibility and delivers the desired goods.

6.2.3.1 Extension of the Project

If you look at the project critically, you will surely notice a weak point: If there is no supplier for eggs, for example, the request will not be processed at all. Comment out the line in which the farm shop is created and restart the project. You find that the request is simply ignored. So it makes sense to provide a default behavior. To do this, extend the `AbstractMerchant` class. You can find the code for this in the example project CoR_2.

```
public abstract class AbstractMerchant {
    // … abridged
    private void printMessage(Purchase purchase) {
        System.out.println("Unfortunately, no merchant is able to
sell " + purchase
+ "can deliver.");
    }
    protected void forward(Purchase purchase) {
        if (next != null)
            next.sell(purchase);
        else
            printMessage(purchase);
    }
}
```

6.2.3.2 Variations of the Pattern

Would it have made a difference if you had created a list – ArrayList or LinkedList – and stored the merchants in it? You could have iterated over the list with a loop and stopped the loop from the moment one of the traders sells the goods. But the Chain offers a lot more possibilities than an aggregate.

Designing Hierarchies and Selecting any Entry Point

In the trader example, there are many traders, but they are not in any hierarchy among themselves. Let us think through a completely different example. In a company, the number of hierarchical levels increases in proportion to the size of the company. In a company with ten employees, the line to the boss is likely to still be very direct – flat hierarchies are indeed flat. If a company has 100 employees, you need at least one level of department head and below that perhaps a level for unit heads. Companies with 500 employees have, in probably this order: the board of directors, the department heads, the sub-department heads, the unit heads, the group heads, and finally the team heads. Such a company needs strict rules to justify this hierarchy. An example could be: When someone goes on a business trip, the group leader must approve the trip if it does not exceed 1 day and is within the company's local area. If a trip is longer than a day, the unit manager must sign off on the trip. If a long distance trip is coming up that is longer than a day, the department head must be involved. If you are an ordinary employee, you will always give your travel request to your team leader, who is the first in the chain. The team leader will check their responsibility and either approve your travel request or, if they are not responsible, pass it on. However, if you yourself are a sub-department head and want to travel for two days, you will probably not hand in your travel request to one of your subordinate unit heads, but to your next superior, the department head.

For the chain of responsibility, this means that each client can choose a different entry point for its message.

Links in the Chain Modify the Request

However, I would like to elaborate on two more special features of the pattern and take up the example with the traders again. You can define the sell method as you like. In the following example, it should be possible for you to pass an object of type Purchasing. The Purchasing class is an auxiliary class. It has two attributes: the goods themselves, as you already know from project CoR_1, and the quantity. When a merchant sells the goods, he calls the method `sell()`; the method `stillDemanding()` returns whether the customer wants to buy more of the same article.

```
public class Purchasing {
    protected final ware;
    private int quantity;
    Purchase(Groceries article, int quantity) {
        this.article = article;
        this.quantity = quantity;
    }
```

```
    public void sellGood() {
        quantity--;
    }
    public boolean stillDemanding() {
        return quantity > 0;
    }
    @Override
        public String toString() {
        return article.toString();
    }
}
```

A trader now sells the desired goods until either he himself can no longer offer any or the quantity demanded is zero. The question of whether he has any goods at all is decided by a random generator. Using the example of the bakery, let's look at the changes.

```
public class Bakery extends AbstractMerchant {
    // … abridged
    @Override
    public void sell(purchase purchase) {
        if (purchase.article == Groceries.BREAD)
            while (isAvailable() &&
                   purchase.stillDemanding()) {
               System.out.println(name + " sells " +
                  purchase.article);
               purchase.sellGood();
            }
            if (purchase.stillDemanding())
                forward(purchase);
        }

    private boolean isAvailable() {
        double number = Math.random() * 10;
        return number >= 5;
    }
}
```

So each link in the chain can now modify the request – in this case: the purchase.

6.3 An Example from the Class Library

Where can this pattern be demonstrated in Java? Take a look at the example project CoR_3. The main method builds a new window. An instance of the class `JFrame` contains an instance of the class `JPanel` and this in turn contains an instance of the class `JLabel`.

Furthermore, there is an EventListener that, when a component is clicked with the mouse, outputs on the console which component was clicked.

```
private final MouseAdapter adapter = new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        var source = e.getSource();
        System.out.println(source.getClass());
    }
};
```

This EventListener is assigned to all three components:

```
myLabel.addMouseListener(adapter);
myFrame.addMouseListener(adapter);
myPanel.addMouseListener(adapter);
```

Run the program and observe which source is output each time you click the window, panel, or label. If you click the label, the label is named as the event source. If you click on the panel, the panel is and in the other cases, the window.

Comment out the first line so that no event handler is assigned to the label. If you now click on it at runtime, the panel is output as the event source. Each component checks its responsibility and passes the event on to the surrounding container if necessary – until the window can no longer hand over its responsibility. A second characteristic of the pattern can be seen here: Processing goes **from the specific to the general**, from the leaves to the root. You will get to know the composite pattern further on, where the processing goes from the root to the leaves, i.e. vice versa.

The example project CoR_4 goes one step further. The GlassPane of the window is activated and gets the EventListener assigned.

```
Component glass = myFrame.getGlassPane();
glass.setVisible(true);
glass.addMouseListener(adapter);
```

Now it doesn't matter where you click – it's always the GlassPane that processes the MouseEvent. In practice, you can intercept user input and mouse clicks this way, for example, when your program is performing a large calculation. You effectively cover the user interface with the GlassPane to protect it from unwanted access. Once you set the "visibility" of the pane back to `false`, the covered interface is accessible again. The GlassPane is an object of type JPanel, so on the console this type is always named as the event source. Now this example doesn't really add anything new to the theme of the pattern, but it's a handy tip that I like to put here.

6.4 Chain of Responsibility – The UML Diagram

Figure 6.1 shows the UML diagram for the example project CoR_2.

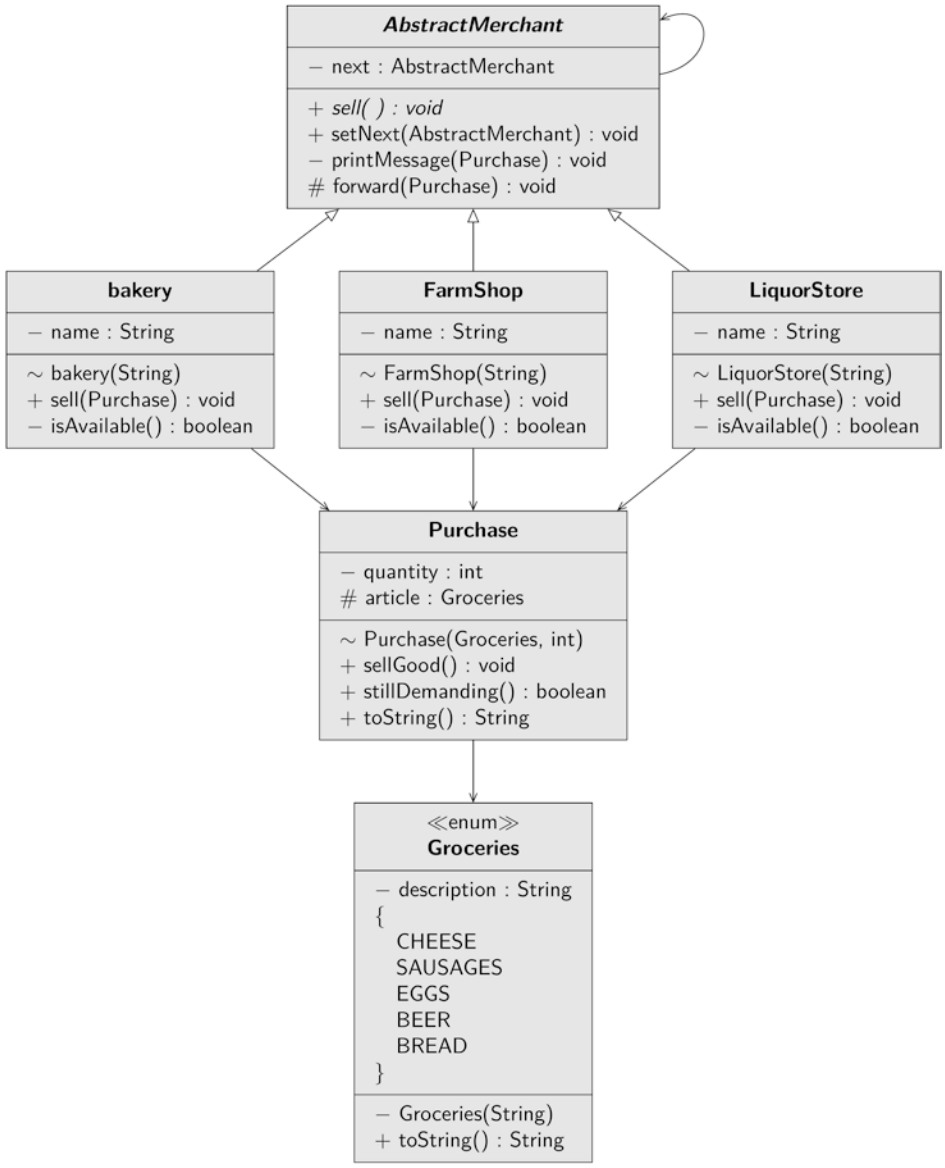

Fig. 6.1 UML diagram of the Chain of Responsibility Pattern (example project CoR_2)

6.5 Summary

Go through the chapter again in key words:

- Any number of objects can potentially solve a problem.
- These objects are concatenated in a chain or organized in a tree.
- A request is only passed to one link in the chain.
- If the object cannot solve the problem, it passes the request on.
- An object can modify a request.
- Every other object checks its jurisdiction; either it responds to the request or it forwards it.
- There is a risk that a request will go completely unanswered; therefore, a default behavior should always be implemented.

6.6 Description of Purpose

The Gang of Four describes the purpose of the Chain of Responsibility pattern as follows:

> Avoid coupling the trigger of a request to its receiver by allowing more than one object to complete the request. Chain the receiving objects and route the request along the chain until one object completes it.