CHAPTER 22

Chain-of-Responsibility Pattern

This chapter covers the chain-of-responsibility pattern.

GoF Definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Concept

In this pattern, you form a chain of objects where each object in the chain handles a particular kind of request. If an object cannot handle the request fully, it passes the request to the next object in the chain. The same process may follow until the end of a chain is reached. This kind of request handling mechanism gives you the flexibility to add a new processing object (handler) at the end of the chain. Figure 22-1 depicts such a chain with N number of handlers.

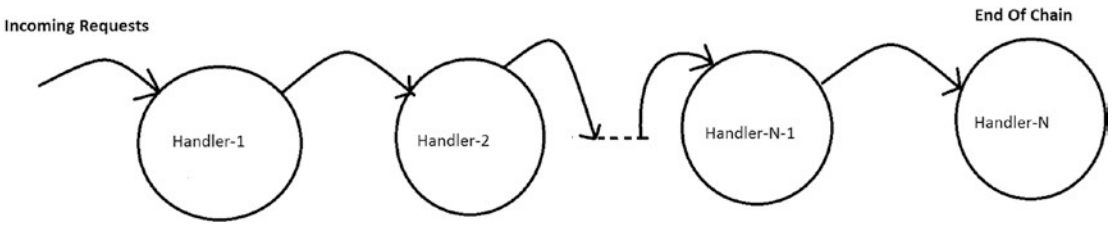

Figure 22-1. *The concept of a chain-of-responsibility pattern*

V. Sarcar, *Java Design Patterns*, https://doi.org/10.1007/978-1-4842-4078-6_22

Real-World Example

- Each organization employs customer care executives who receive feedback or complaints directly from the customers. If the employees cannot answer the customers' issues properly, they forward these issues/escalations to the appropriate departments in the organization. These departments do not try to fix an issue simultaneously. In the first phase of investigation, the department that seems responsible analyzes the case, and if they believe that the issue should be forwarded to another department, they do that.
- A similar scenario occurs when a patient visits a hospital. Doctors from one department can refer a patient to a different department for further diagnosis.

Computer-World Example

Consider a software application (e.g., a printer) that can send emails and faxes. So, customers can report faxing issues or email issues. Let's assume that these issues are handled by handlers. So, you introduce two different types of error handlers: `EmailErrorHandler` and `FaxErrorHandler`. You can assume that `EmailErrorHandler` handles email errors only; it cannot fix the fax errors. In a similar manner, `FaxErrorHandler` handles fax errors and does not care about email errors.

So, you may form a chain like this: whenever the application finds an error, it raises a ticket and forwards the error with a hope that one of the handlers will handle it. Let's assume that the request first comes to `FaxErrorhandler`. If this handler agrees that the error is a fax issue, it handles it; otherwise, the handler forwards the issue to `EmailErrorHandler`.

Note that the chain ends with `EmailErrorHandler`. But if you need to handle a different type of issue—for example, authentication issues (which can occur due to security vulnerabilities), you can make a handler called `AuthenticationErrorHandler` and place it after `EmailErrorHandler`. Now if an `EmailErrorHandler` cannot fix the issue completely, it forwards the issue to `AuthenticationErrorHandler`, and the chain ends there.

Note You are free to place these handlers in any order you choose in your application.

So, the bottom line is that the processing chain may end in any of the following scenarios.

- Any of these handlers could process the request completely and control comes back.
- A handler cannot handle the request completely, so it passes the request to the next handlers. This way, you reach the end of the chain. So, the request is handled there. But if the request cannot be processed there, you cannot pass it further. (You may need to take special care for such a situation.)

You notice a similar mechanism when you are implementing an exception handling mechanism with multiple catch blocks in your Java application. If an exception occurs in a `try` block, the first catch block tries to handle it. If it cannot handle that type of exception, the next `catch` block tries to handle it, and the same mechanism is followed until the exception is handled properly by handlers (`catch` blocks). If the last catch block in your application is unable to handle it, an exception is thrown outside of this chain.

Note In java.util.logging.Logger, you can see a different overloaded version of log() methods that supports a similar concept.

Another built-in support can be seen in the doFilter (ServletRequest request, ServletResponse response, FilterChain chain) interface method in javax.Servlet.Filter.

Illustration

Let's consider the scenario that is discussed in the computer-world example. Let's further assume that in the following example, you can process both normal and high-priority issues that may come from either the email or fax pillar.

Class Diagram

Figure 22-2 shows the class diagram.

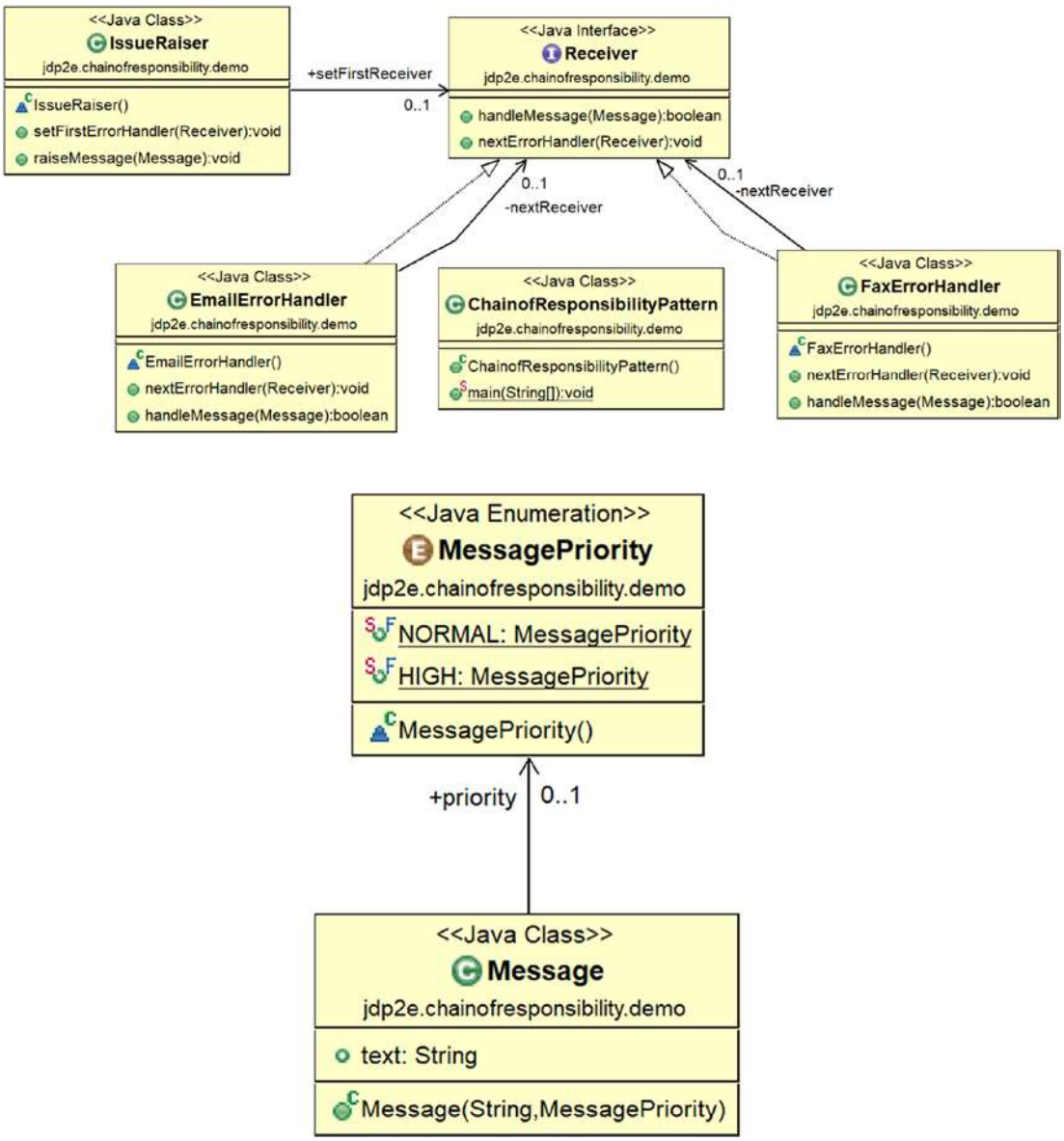

Figure 22-2. *Class diagram*

Package Explorer View

Figure 22-3 shows the high-level structure of the program.

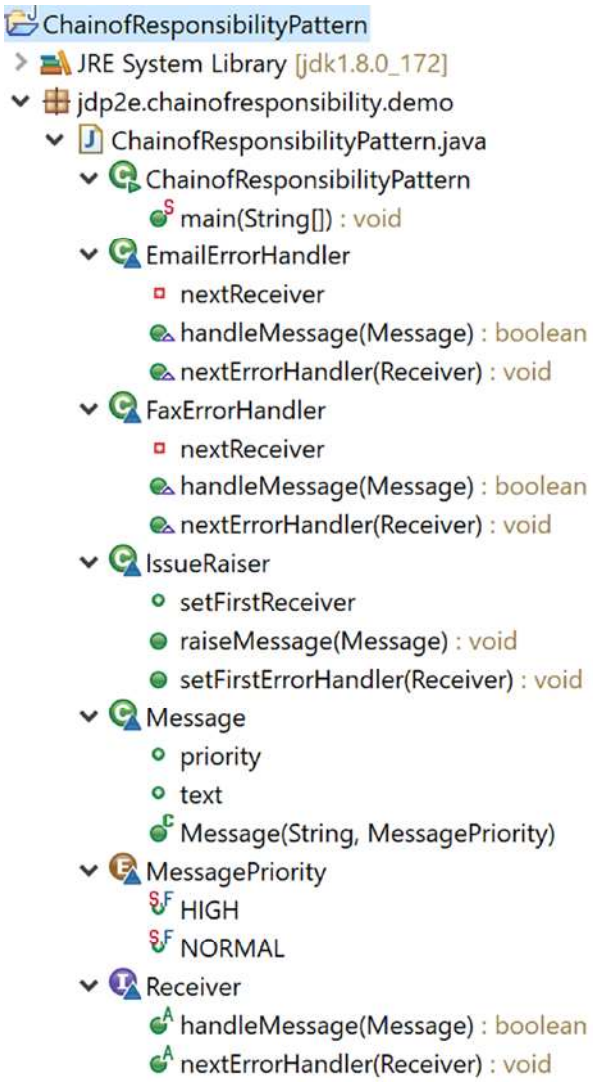

Figure 22-3. *Package Explorer view*

Implementation

Here's the implementation.

```
package jdp2e.chainofresponsibility.demo;

enum MessagePriority
{
    NORMAL,
    HIGH
}
class Message
{
    public String text;
    public MessagePriority priority;
    public Message(String msg, MessagePriority p)
    {
        text = msg;
        this.priority = p;
    }
}

interface Receiver
{
    boolean handleMessage(Message message);
    void nextErrorHandler(Receiver nextReceiver);
}
class IssueRaiser
{
    public Receiver setFirstReceiver;
    public void setFirstErrorHandler(Receiver firstErrorHandler)
    {
        this.setFirstReceiver = firstErrorHandler;
    }
    public void raiseMessage(Message message)
    {
        if (setFirstReceiver != null)
```

```
            setFirstReceiver.handleMessage(message);
    }
}
// FaxErrorHandler class
class FaxErrorHandler implements Receiver
{
    private Receiver nextReceiver;
    @Override
    public void nextErrorHandler(Receiver nextReceiver)
    {
        this.nextReceiver = nextReceiver;
    }
    @Override
    public boolean handleMessage(Message message)
    {
        if (message.text.contains("Fax"))
        {
            System.out.println(" FaxErrorHandler processed " +message.
            priority +" priority issue :"+ message.text);
            return true;
        }
        else
        {
            if (nextReceiver != null)
                nextReceiver.handleMessage(message);
        }
        return false;
    }
}
// EmailErrorHandler class
class EmailErrorHandler implements Receiver
{
    private Receiver nextReceiver;
    @Override
    public void nextErrorHandler(Receiver nextReceiver)
```

```
    {
        this.nextReceiver = nextReceiver;
    }
    @Override
    public boolean handleMessage(Message message)
    {
        if (message.text.contains("Email"))
        {
            System.out.println(" EmailErrorHandler processed "+message.
            priority+ " priority issue: "+message.text);
            return true;
        }
        else
        {
            if (nextReceiver != null)
                nextReceiver.handleMessage(message);
        }
        return false;
    }
}
//Client code
public class ChainofResponsibilityPattern {

    public static void main(String[] args) {
        System.out.println("\n ***Chain of Responsibility Pattern
        Demo***\n");
        /* Forming the chain as IssueRaiser->FaxErrorhandler->
        EmailErrorHandler*/
        Receiver faxHandler, emailHandler;
        //Objects of the chains
        IssueRaiser issueRaiser = new IssueRaiser();
        faxHandler = new FaxErrorHandler();
        emailHandler = new EmailErrorHandler();
        //Making the chain
        //Starting point:IssueRaiser will raise issues and set the first
        //handler
```

```
        issueRaiser.setFirstErrorHandler(faxHandler);
        //FaxErrorHandler will pass the error to EmailHandler if needed.
        faxHandler.nextErrorHandler(emailHandler);
        //EmailErrorHandler will be placed at the last position in the chain
        emailHandler.nextErrorHandler(null);

        Message m1 = new Message("Fax is going slow.",
        MessagePriority.NORMAL);
        Message m2 = new Message("Emails are not reaching.",
        MessagePriority.HIGH);
        Message m3 = new Message("In Email, CC field is disabled always.",
        MessagePriority.NORMAL);
        Message m4 = new Message("Fax is not reaching destinations.",
        MessagePriority.HIGH);

        issueRaiser.raiseMessage(m1);
        issueRaiser.raiseMessage(m2);
        issueRaiser.raiseMessage(m3);
        issueRaiser.raiseMessage(m4);
    }
}
```

Output

Here's the output.

```
***Chain of Responsibility Pattern Demo***

 FaxErrorHandler processed NORMAL priority issue :Fax is going slow.
 EmailErrorHandler processed HIGH priority issue: Emails are not reaching.
 EmailErrorHandler processed NORMAL priority issue: In Email, CC field is
 disabled always.
 FaxErrorHandler processed HIGH priority issue :Fax is not reaching
 destinations.
```

Q&A Session

1. **In the example, what is the purpose of message priorities?**

   Good catch. Actually, you could ignore them because, for simplicity in the handlers, you are just searching for the words *email* or *fax*. These priorities are added to beautify the code. But instead of using separate handlers for email and fax, you could make a different kind of chain that handles the messages based on the priorities. In such a case, these priorities can be used more effectively.

2. **What are the advantages of using a chain-of-responsibility design pattern?**

   - You can have more than one object to handle a request. (Notice that if a handler cannot handle the whole request, it may forward the responsibility to the next handler in the chain).
   - The nodes of the chain can be added or removed dynamically. Also, you can shuffle the order. For example, if you notice that the majority of issues are with email processing, then you may place EmailErrorHandler as the first handler in the chain to save the average processing time of the application.
   - A handler does not need to know how the next handler in the chain will handle the request. It focuses only on its own handling mechanism.
   - In this pattern, you are promoting loose coupling because it decouples the senders (of requests) from the receivers.

3. **What are the challenges associated with using the chain-of-responsibility design pattern?**

   - There is no guarantee that the request will be handled (fully or partially) because you may reach the end of the chain; but it is possible that you have not found any explicit receiver to handle the request.
   - Debugging may become tricky with this kind of design.

4. **How can you handle the scenario where you have reached at the end of chain, but the request is not handled at all?**

   One simple solution is to use `try/catch` (or `try/finally` or `try/catch/finally`) blocks. You may put the handlers in these constructs. You may notice that a `try` block can be associated with multiple `catch` blocks also.

   In the end, if no one can handle the request, you may raise an exception with the appropriate messages and catch the exception in your intended `catch` block to draw your attention (or handle it in some different way).

   The GoF talked about Smalltalk's automatic forwarding mechanism, `doesNotUnderstand`, in a similar context. If a message cannot find a proper handler, it is caught in `doesNotUnderstand` implementations that can be overridden to forward the message in the object's successor, log it in a file, and store it in a queue for later processing, or you can simply perform any other intended operations. But you must make a note that by default, this method raises an exception that needs to be handled in a proper way.

5. **In short, if a handler cannot handle the request fully, it will pass it to the next handler. Is this correct?**

   Yes.

6. **It appears that there are similarities between the observer pattern and the chain-of-responsibility pattern. Is this correct?**

   In an observer pattern, all registered users get notifications in parallel; but in a chain-of-responsibility pattern, objects in the chain are notified, one by one, in a sequential manner. This process continues until an object handles the notification fully (or you reach the end of the chain). I show the comparisons in diagrams in the "Q&A Session" in Chapter 14.