Design Pattern: Chain of Responsibility

Note that the code found in this tutorial is written to illustrate the advantages of using the pattern and the disadvantages of not using it. Any syntactical errors or lack of best practices are not to be modeled.

Overview

Objective

1. To understand how the Chain of Responsibility (a.k.a. Handler) design pattern solves common problems found in software design.
2. To understand *how* and *where* Liferay Portal uses this pattern.

Agenda

1. Understand a common extension problem found in software design.
2. Introduce the Chain of Responsibility design pattern.
3. Examine how the Chain of Responsibility pattern solves the problem.
4. Discuss where Liferay Portal uses the Chain of Responsibility pattern.

The Problem

The Problem

Imagine you are working in a small team on an email web application for your company. The business wants a standard set of filtering done on incoming emails for all employees. The first two filtering criteria are:

1. Emails from known clients should have priority set to high.
2. Emails not passing the spam detector should be deleted.

The company already has a SPAM detector software. It just needs to be called to process email messages. You are tasked to implement this feature. Below is the initial code.

The Problem

```
import com.acme.email.SpamDetector;
public class EmailProcessor {
    private SpamDetector detector = new Detector();
    public void process(Email email) {
        // if spam, move to TRASH_BIN
        if ( detector.isSpam( email.getBody() ) {
            email.moveTo( Email.TRASH_BIN );
        }
        // if from client, mark high priority
        else if ( isFromClient(email) ) {
            email.setPriority( Email.HIGH_PRIORITY );
        }
    }
    private boolean isFromClient(Email email) {
        // check email domain to find a match
        // return true if match found
    }
}
```

The Problem

The team compiles and deploys your code to production. Everything seems to be running well. Two weeks later, the business comes back with a list of 20 additional filters they want applied to all emails to all employees. "No problem!" you thought. You would just update the `EmailProcessor` class. You and your partner Joe decided to split the 20 filtering requirements in half. He'll work on 10, and you the other 10. Since he's going on vacation in a week, you let him work on it first. You will start when he finishes.

```
public void process(Email email) {
    if ( detector.isSpam( email.getBody() ) {
        email.moveTo( Email.TRASH_BIN );
    } else if ( isFromClient(email) ) {
        email.setPriority( Email.HIGH_PRIORITY );
    }
    // additional filters
    if ( email.getBody().indexOf(Email.CRITERIA_ONE) >= 0 ) {
        // do something
    }
    // 19 more filters done in ifs and if/elses
}
```

The Problem

When the code is done, the `process` method inside `EmailProcessor` has 22 `if` or `if-else` statements within. "Oh well!" you thought. The team compiles, runs regression tests, and deploys the new changes to production. Everything seems to be working fine, until the business comes back with a discovery that the spam detector is actually filtering out some crucial business inquiries. The spam detector needs to be disabled ASAP! Again, you'll need to comment out the code. The team needs to, again, build, regression test, deploy!

```
public void process(Email email) {
    //  if ( detector.isSpam( email.getBody() ) {
    //      email.moveTo( Email.TRASH_BIN );
    //  } else
    if ( isFromClient(email) ) {
        email.setPriority( Email.HIGH_PRIORITY );
    }
    // additional filters
    if ( email.getBody().indexOf(Email.CRITERIA_ONE) >= 0 ) {
        // do something
    }
    // 19 more filters done in ifs and if/elses
}
```

The Problem

The business thought that the email web application is doing so well. They now want it packaged and sold as a new product line. However, the filtering system needs to be configurable. All the filters can be enabled or disabled via configuration. How are you to do this? Since everything is already in `EmailProcessor`, you proceed to wrap all the `if` and `if-else` statements with another `if` statement each.

```
if ( EmailConfig.ENABLE_SPAM_FILTER ) {
    if ( detector.isSpam( email.getBody() ) {
        email.moveTo( Email.TRASH_BIN );
    }
}
if ( EmailConfig.ENABLE_CLIENT_PRI ) {
    if ( isFromClient(email) ) {
        email.setPriority( Email.HIGH_PRIORITY );
    }
}
if ( EmailConfig.ENABLE_CRITERIA_ONE ) {
    if ( email.getBody().indexOf(Email.CRITERIA_ONE) >= 0 ) {
    // do something
    }
}
```

Question Set I

1. Identify all the pain points in the software code above.

2. How would you have solved these pain points?

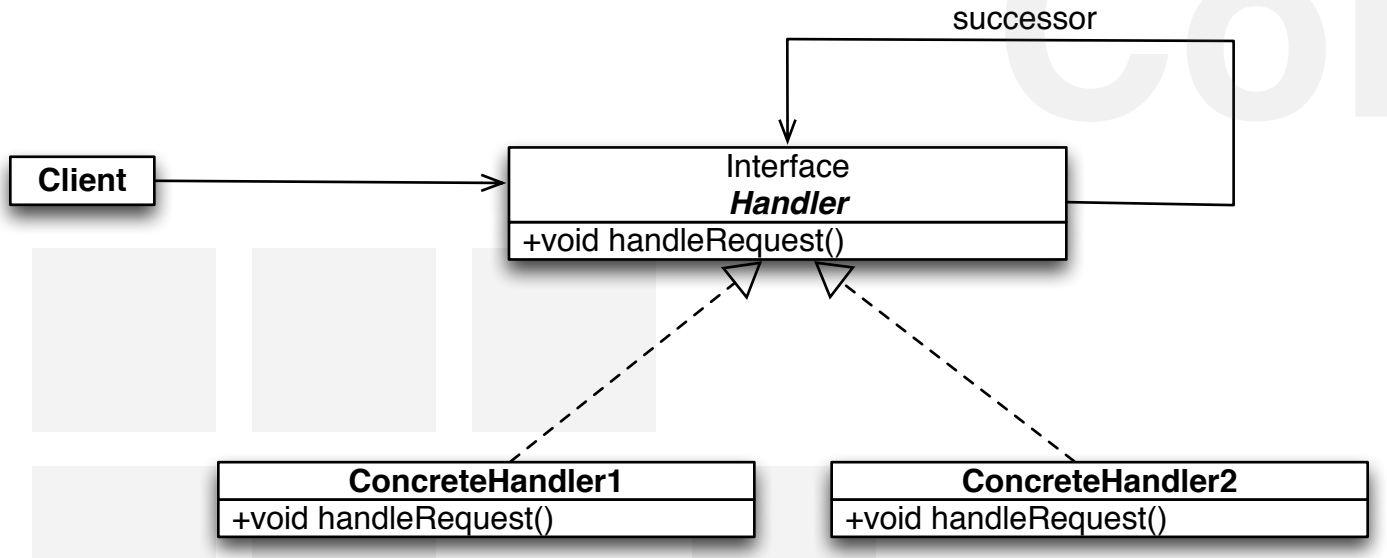
Introduction

The Chain of Responsibility design pattern is a solution to the problem in the previous section. The idea behind it is to split each filtering requirement into its own class, and then chain them together. Here are the parts.

1. Each class doing a part is known as a `Handler`.
2. Each handler has a reference to the next handler in the chain.

The diagram in the next slide illustrates their relationship.

Diagram



As you can see, the `Handler` interface exposes a method to handle a request. All concrete (real) `Handler` classes will place their core logic into that method. Each `Handler` class also has a reference to one other handler (successor) next in line to do the job. So when a Handler is done with its tasks, it may call the next `Handler` in line to `handleRequest`.

Gang of Four

Now open up your *Gang of Four* book and look for the Chain of Responsibility section under Behavioral Patterns.

Review and try to understand (as a class) the intent, motivation, applicability, structure, participants, and consequences of this pattern.

Question Set II

1. What real world examples can you think of that are analogous to the Chain of Responsibility pattern?

2. With what you know so far about the Chain of Responsibility pattern, what problems does it solve?

3. What problems are still outstanding?

4. Are there any new problems introduced along with the pattern? if so, what?

Solution

Going back to our example from the first section, all the filtering criteria would be grouped into different `Handler` classes. For example, the spam detector call can be placed in a `Handler` class called `SpamEmailHandler`. The client email priority piece can be placed in the `ClientPriorityHandler`. The remaining tasks can be added similarly.

```
public interface EmailHandler {
    public void handleEmail(Email email);
}
```

```
public class SpamEmailHandler implements EmailHandler {
    private EmailHandler successor;

    public void handleEmail(Email email) {
        if ( detector.isSpam( email.getBody() ) ) {
            email.moveTo( Email.TRASH_BIN );
        }
        if ( successor != null )
            successer.handleEmail(email);
    }
}
```

Solution

```
public class ClientPriorityHandler implements EmailHandler {
    private EmailHandler successor;

    public void handleEmail(Email email) {
        if ( isFromClient(email) ) {
            email.setPriority( Email.HIGH_PRIORITY );
        }
        if ( successor != null )
            successer.handleEmail(email);
    }

    private boolean isFromClient(Email email) {
        // check email domain to find a match
        // return true if match found
    }
}
```

With all the `Handler` classes in place, the next question is, how would you chain them together? In the class diagram, `Handler` classes are supposed to hold a reference to another `Handler` class that they are supposed to call afterward.

Question Set III

1. What are some ways you can chain the `Handler` classes together?

2. What are the tradeoffs in each of these different ways to chain handlers?

3. What if someone implements a `Handler` but forgot about the successor? How can you enforce that?

Solution

One way to solve the chaining problem is to offload it to another class, let's call it the `EmailHandlerController`. This class would be responsible for setting the handlers in the proper chaining order. However, if all handlers and their ordering are hardcoded in the `EmailHandlerController`, that defeats the purpose.

? Why is hardcoding the handlers into the `EmailHandlerController` class a bad idea?

A better approach would be to create a configuration file, or a properties file. This configuration file contains a list of handlers that are enabled. It also indicates the ordering of the chain. On initialization, `EmailHandlerController` would read the config file and load all the handlers. The `EmailHandlerController` would also set the successors for each `Handler` .

Solution

Here is an example of the configuration file.

```
<?xml version="1.0"?>
<handlers>
    <handler>
        <class>com.acme.email.handler.SpamEmailHandler</class>
        <enabled>true<enabled>
    </handler>
    <handler>
        <class>com.acme.email.handler.ClientPriorityHandler</class>
        <enabled>true<enabled>
    </handler>
    <handler>
        <class>com.acme.email.handler.AttachmentHandler</class>
        <enabled>true<enabled>
    </handler>
    ...
</handlers>
```

Advantages

What are the advantages of this solution?

1. The filtering criteria is no longer tied to one class, namely the `EmailProcessor`.

2. Changing the order of the `Handler` classes or removing one from the chain does not require code changes.

3. Adding additional filtering criteria via creating new `Handler` classes can happen independently.

Question Set IV

1. Does Liferay Portal make use of the Chain of Responsibility (Handler) design pattern? If so, where?
2. What's a good way for a client to access the first handler in the chain?
3. What are some code variations you can think of in chaining the handlers?
4. What are some shortcomings of this pattern?

Servlet Filters

Because Liferay Portal is a Java web application running in a servlet container, it makes use of Servlet filters. Servlet filters are a customizable chain of filters that can act on `ServletRequest` objects before the `Servlet` gets them, or on `ServletResponse` objects before they are returned.

The structure of Servlet filters mimics the Chain of Responsibility design pattern, but with a slight variation. As a web developer, you have the ability to add additional Servlet filters to alter or complement HTTP requests coming into Liferay Portal. The same is applicable to HTTP responses that Liferay Portal returns.

Homework

1. Please read through the following tutorial on Servlet filters:
   `http://in.liferay.com/documents/210910/1212914/Follow+the+Chain+of+Responsibility/6faaea88-0753-4604-a259-6d66fa77e0d5`

2. Create a new servlet filter in Liferay Portal that checks whether a user is logged in. If the user is not logged in, redirect the user to the home page, so that a guest user can not visit any pages other than the home page and the login page.