36. Chain of Responsibility

Chain of Responsibility is a design pattern that allows you to apply multiple processing steps to an action. Any step of the process may have a condition added to it, which will allow it to exit the process immediately and not execute any further steps.

Perhaps, the best-known example of Chain of Responsibility design pattern in .NET is ASP.NET Core middleware[2]. This is where you can apply multiple stages of processing to incoming HTTP requests. The request will not get to its destination until it goes through all the steps, which you, as an application developer, can add or remove at will. For example, you can add a step that will verify whether or not the user is authorized. Then, if so, the request will proceed to the next stage of processing. If not, the response will be returned immediately with an appropriate response code.

Chain of Responsibility can be summarized as follows:

- There is either an abstract class or an interface that represent a **Handler** object.
- The **Handler** has a method to set its **Successor**, which would be another instance of the **Handler** object.
- The **Handler** would also have the definition of the actual method that would represent the processing step.
- Each concrete implementation of a **Handler** would have some logic inside this method, which, after performing its own processing, would call the same method on the **Successor** (if one is set).

[2] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/

- The method may also have a condition which will allow it to short-circuit the process and return immediately.

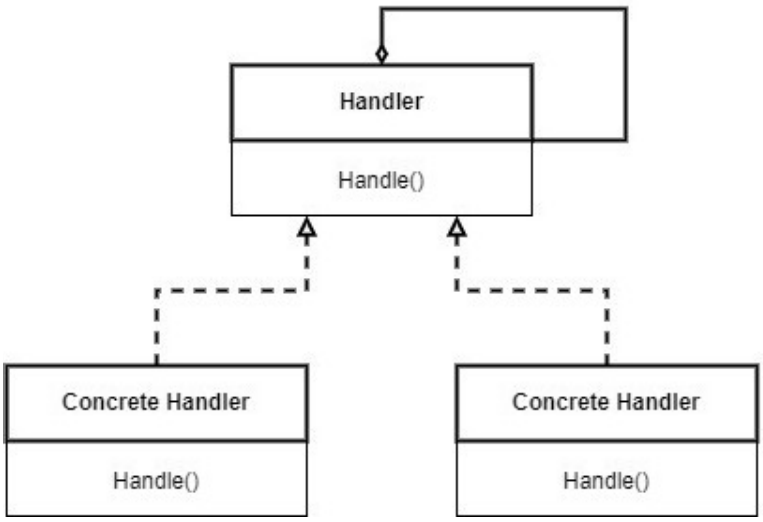

Figure 36.1 - Chain of Responsibility UML diagram

We will now go through an example implementation of Chain of Responsibility. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Chain-of-Responsibility

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Chain of Responsibility implementation example

To demonstrate Chain of Responsibility, we will use a simplified request processing example, similar to what ASP.NET Core middleware uses. But, as this is just a conceptual representation of request processing and not an actual thing, we won't have to build a fully-fledged web application. We will create a simple console app.

Once created, we will add classes to it that will represent our `Request` and `Response` objects. Our `Request` object will look like this:

```
namespace Chain_of_Responsibility_Demo;

internal class Request
{
    public string? Username { get; set; }
    public string? Password { get; set; }
    public string? Role { get; set; }
}
```

And here is what our `Response` object will look like:

```
namespace Chain_of_Responsibility_Demo;

internal class Response(
    bool success, string message)
{
    public bool Success { get; }
        = success;
    public string? Message { get; }
        = message;
}
```

Next, we will add a representation of a **Handler**. In our case, it will be the `RequestHandler.cs` file with the following content:

```
namespace Chain_of_Responsibility_Demo;

internal abstract class RequestHandler
{
    protected RequestHandler? successor;

    public abstract Response HandleRequest(
        Request request);

    public void SetNext(
        RequestHandler successor)
    {
        this.successor = successor;
    }
}
```

In here, we have a protected `successor` field, which represents the next instance of a **Handler** in the chain. Because the process of setting the successor will be identical for any concrete **Handler**, we have a non-virtual `SetNext` method. The main processing step is represented by the `HandleRequest` method, which is abstract because it will be specific to each specific **Handler** implementation.

And now we will start adding specific implementations of it. First, we will add `AuthenticationHandler.cs` file with the following content:

```
namespace Chain_of_Responsibility_Demo;

internal class AuthenticationHandler :
    RequestHandler
{
    public override Response HandleRequest(
        Request request)
    {
        if (request.Username != "John" ||
            request.Password != "password")
            return new Response(
                false,
                "Invalid username or password.");

        if (successor is not null)
            return successor.HandleRequest(
                request);

        return new Response(
            true,
            "Authentication successful.");
    }
}
```

In here, purely for demo purposes, we are checking whether the request contains a specific username and password. Of course, in a real-life scenario, you wouldn't use hardcoded values. But we are using them here for the sake of simplicity.

If either the username or the password doesn't match, we return the call immediately. Otherwise, we either call the `HandleRequest` method on the **Successor**, if one is set or just return a response that indicates success.

Next, we will add the `AuthorizationHandler.cs` file with the following content:

```
namespace Chain_of_Responsibility_Demo;

internal class AuthorizationHandler :
    RequestHandler
{
    public override Response HandleRequest(
        Request request)
    {
        if (request.Role != "Admin")
            return new Response(
                false,
                "User not authorized.");

        if (successor is not null)
            return successor.HandleRequest(
                request);

        return new Response(
            true,
            "Authorization successful.");
    }
}
```

This is an authorization handler. The idea behind it is that, even if a correct username and password were provided, the user may still not be authorized to access the resource. In this case, we are only allowing access to a user that has the role of `Admin`. Otherwise, the process is the same as before. Call the successor if one is set. Return a response with a success code otherwise.

Now, to bring it all together, we will replace the content of the `Program.cs` file with the following:

```
using Chain_of_Responsibility_Demo;

var handler1 =
    new AuthenticationHandler();
var handler2 =
    new AuthorizationHandler();
handler1.SetNext(handler2);

var request1 = new Request
{
    Username = "Invalid",
    Password = "Invalid",
};

var request2 = new Request
{
    Username = "John",
    Password = "password",
    Role = "User"
};

var request3 = new Request
{
    Username = "John",
    Password = "password",
    Role = "Admin"
};

var response1 = handler1
    .HandleRequest(request1);
var response2 = handler1
    .HandleRequest(request2);
var response3 = handler1
    .HandleRequest(request3);
Console.WriteLine(
```

```
36      $"Response 1: Success - {
37          response1.Success}, Message - {
38          response1.Message}");
39  Console.WriteLine(
40      $"Response 2: Success - {
41          response2.Success}, Message - {
42          response2.Message}");
43  Console.WriteLine(
44      $"Response 3: Success - {
45          response3.Success}, Message - {
46          response3.Message}");
47
48  Console.ReadKey();
```

So, we are setting our Chain of Responsibility up in such a way that `AuthorizationHandler` is the successor of `AuthenticationHandler`. Then, we send three requests to it to see what responses we would get. In the first request, we have provided the wrong username and password. In the second request, the username and password are correct, but the role is wrong from the perspective of `AuthorizationHandler`. In the third request, the username, the password, and the role are correct.

And this is the result that we get:

```
Response 1: Success - False, Message - Invalid username or password.
Response 2: Success - False, Message - User not authorized.
Response 3: Success - True, Message - Authorization successful.
```

Figure 36.2 - Chain of Responsibility in action

As we expected, the first message indicates that `AuthenticationHandler` has marked the response as a failure. The second message indicates that `AuthorizationHandler` has marked the response as a failure. The third message indicates that `AuthorizationHandler` has marked the response as a success.

This concludes the overview of the Chain of Responsibility design pattern. Let's summarize its main benefits.

Benefits of using Chain of Responsibility

The benefits of using Chain of Responsibility can be summarized as follows:

- Processing stages can be easily added and removed at will.
- The single responsibility principle is well implemented, as each processing step is isolated in its own implementation.
- Open-closed principle is well implemented, as you can add new functionality without modifying any existing classes.

Of course, Chain of Responsibility has its own caveats. And this is what we will have a look at next.

Caveats of using Chain of Responsibility

There aren't really many caveats about using a Chain of Responsibility. But what you really need to watch out for is that you may end up with your requests remaining unhandled if you make your chain too complex and misplace any of its links.

It's not always possible to make each concrete **Handler** implementation agnostic of the step it's used in. For example, checking whether a user is authorized to access a resource would be meaningless if we haven't already checked whether or not the user is authenticated. And this is exactly the reason why accidentally misplacing the steps may cause your chain to fail.

So, when using this design pattern, you need to always think about such a scenario. If it's possible, make each **Handler** implementation agnostic of the sequence of the steps it's used in. If it can't be done - make sure that sufficient documentation is provided.