Chain of Responsibility Design Pattern

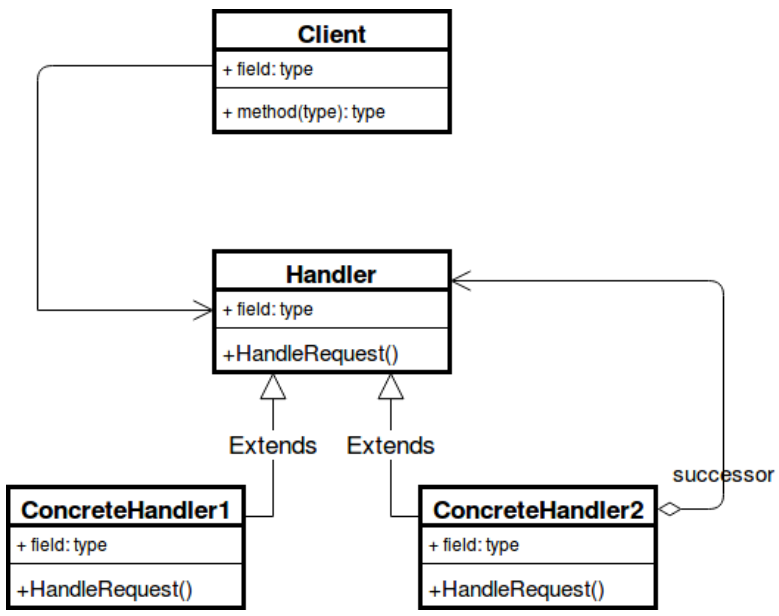
Difficulty Level : Easy • Last Updated : 24 Feb, 2022

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Where and When Chain of Responsibility pattern is applicable :

- When you want to decouple a request's sender and receiver
- Multiple objects, determined at runtime, are candidates to handle a request
- When you don't want to specify handlers explicitly in your code
- When you want to issue a request to one of several objects without specifying the receiver explicitly.

This pattern is recommended when multiple objects can handle a request and the handler doesn't have to be a specific object. Also, the handler is determined at runtime. Please note that a request not handled at all by any handler is a valid use case.



- **Handler :** This can be an interface which will primarily receive the request and dispatches the request to a chain of handlers. It has reference to the only first handler in the chain and does not know anything about the rest of the handlers.
- **Concrete handlers :** These are actual handlers of the request chained in some sequential order.
- **Client :** Originator of request and this will access the handler to handle it.

How to send a request in the application using the Chain of Responsibility

The Client in need of a request to be handled sends it to the chain of handlers which are classes that extend the Handler class. Each of the handlers in the chain takes its turn trying to handle the request it receives from the client.
If ConcreteHandler1 can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler2, the next one in the chain.

Let's see an Example of Chain of Responsibility Design Pattern:

Java

```
public class Chain
{
```

```
private void buildChain(){
    chain = new NegativeProcessor(new ZeroProcessor(new PositiveProcesso
}

public void process(Number request) {
    chain.process(request);
}

}

abstract class Processor
{
    private Processor nextProcessor;

    public Processor(Processor nextProcessor){
        this.nextProcessor = nextProcessor;
    };

    public void process(Number request){
        if(nextProcessor != null)
            nextProcessor.process(request);
    };
}

class Number
{
    private int number;

    public Number(int number)
    {
        this.number = number;
    }

    public int getNumber()
    {
        return number;
    }

}

class NegativeProcessor extends Processor
{
    public NegativeProcessor(Processor nextProcessor){
        super(nextProcessor);

    }

    public void process(Number request)
    {
        if (request.getNumber() < 0)
        {
            System.out.println("NegativeProcessor : " + request.getNumber());
        }
        else
        {
            super.process(request);
        }
    }
}

class ZeroProcessor extends Processor
{
    public ZeroProcessor(Processor nextProcessor){
        super(nextProcessor);
    }

    public void process(Number request)
    {
        if (request.getNumber() == 0)
        {
            System.out.println("ZeroProcessor : " + request.getNumber());
        }
        else
        {
```

```
{

    public PositiveProcessor(Processor nextProcessor){
        super(nextProcessor);
    }

    public void process(Number request)
    {
        if (request.getNumber() > 0)
        {
            System.out.println("PositiveProcessor : " + request.getNumbe
        }
        else
        {
            super.process(request);
        }
    }
}

class TestChain
{
    public static void main(String[] args) {
        Chain chain = new Chain();

        //Calling chain of responsibility
        chain.process(new Number(90));
        chain.process(new Number(-50));
        chain.process(new Number(0));
        chain.process(new Number(91));
    }
}
```

Output:

```
PositiveProcessor : 90
NegativeProcessor : -50
ZeroProcessor : 0
PositiveProcessor : 91
```

Advantages of Chain of Responsibility Design Pattern

- To reduce the coupling degree. Decoupling it will request the sender and receiver.
- Simplified object. The object does not need to know the chain structure.
- Enhance flexibility of object assigned duties. By changing the members within the chain or change their order, allow dynamic adding or deleting responsibility.
- Increase the request processing new class of very convenient.

DisAdvantages of Chain of Responsibility Design Pattern

- The request must be received not guarantee.
- The performance of the system will be affected, but also in the code debugging is not easy may cause cycle call.
- It may not be easy to observe the characteristics of operation, due to debug.

Further Read: Chain of Responsibility Design Pattern in Python

This article is contributed by **Saket Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Next
Command Pattern

RECOMMENDED ARTICLES

Page : 1 2 3

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Easy

Easy Normal Medium Hard Expert

Improved By : Akanksha_Rai, Rahul 2, gabaa406, sureshsanjeevi

Article Tags : Design Pattern

Start Your Coding Journey Now!

Login

Sign in to GeeksforGeeks with Google

Hasan masum
masumbuetcse18@gmail.com

Continue as Hasan

To create your account, Google will share your name, email address, and profile picture with GeeksforGeeks. See GeeksforGeeks's privacy policy and terms of service.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link a

Load Comments

A-143, 9th Floor, Sovereign Corporate Tower,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org