# Chain of responsibility

DESIGN PATTERN PRESENTATION

Gavril Luca, Group 30432

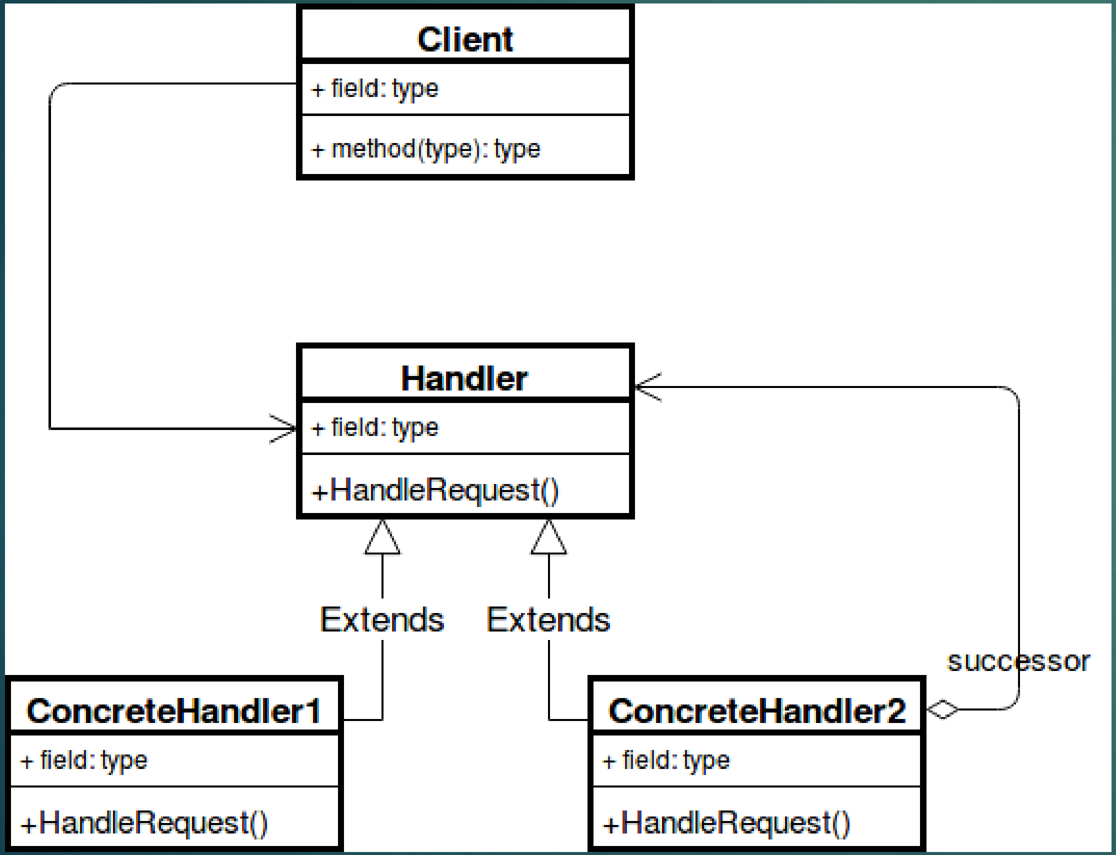
## Overview

- The chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples the sender and receiver of a request based on request type. This pattern comes under behavioral patterns.
- 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.

## When the 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.

## Sample UML Class Diagram:



**Handler :** This can be an interface/abstract class which will primarily receive the request and dispatch the request to the chain of handlers. It has a reference to only the 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 the request, accessing the handler for his request.

The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a "null next".

## A few important principles:

- Each handler in the chain will have its implementation for processing a request
- Every handler in the chain should have reference to the next handler
- Handlers should not form a recursive cycle
- Only one handler in the chain resolves a given request

## Disadvantages:

- Mostly, it can get broken easily:
  - if a handler fails to call the next handler, the request gets dropped
  - if a handler calls the wrong handler, it can lead to a cycle
- It can lead to duplicate code across handlers, increasing maintenance

## Implementation example:

**The following example represents the implementation of a number processing system, using this pattern.**

- **Client/Sender class:**

```
class Number {
    private int number;
    public Number(int number)
    {
        this.number = number;
    }
    public int getNumber()
    {
        return number;
    }
}
```

- **Handler Interface:**

```
interface Chain {
    public abstract void setNext(Chain
nextInChain);
    public abstract void process(Number
request);
}
```

- **Concrete handler classes:**

```
class NegativeProcessor implements
Chain{
   private Chain nextInChain;

   public void setNext(Chain c) {
      nextInChain = c;
   }

   public void process(Number
request) {
      if (request.getNumber() < 0)
      {

System.out.println("NegativeProcess
or : " + request.getNumber());
      }
      else
      {

nextInChain.process(request);
      }
   }
}
```

```
class ZeroProcessor implements
Chain {
   private Chain nextInChain;

   public void setNext(Chain c)
   {
      nextInChain = c;
   }

   public void process(Number
request)
   {
      if (request.getNumber() == 0)
      {

System.out.println("ZeroProcessor : "
+ request.getNumber());
      }
      else
      {

nextInChain.process(request);
      }
```

```
class PositiveProcessor implements
Chain {
   private Chain nextInChain;

   public void setNext(Chain c)
   {
      nextInChain = c;
   }

   public void process(Number
request)
   {
      if (request.getNumber() > 0)
      {

System.out.println("PositiveProcessor :
" + request.getNumber());
      }
      else
      {
         nextInChain.process(request);
      }
   }
```

- **Main class and output:**

```
class TestChain {
   public static void main(String[] args)
{
      //configure Chain of Responsibility
      Chain c1 = new
NegativeProcessor();
      Chain c2 = new ZeroProcessor();
      Chain c3 = new
PositiveProcessor();
      c1.setNext(c2);
      c2.setNext(c3);

      //calling chain of responsibility
      c1.process(new Number(90));
      c1.process(new Number(-50));
      c1.process(new Number(0));
      c1.process(new Number(91));
   }
}
```

**Output:**

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

# References:

- https://www.geeksforgeeks.org/chain-responsibility-design-pattern/
- https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.html
- https://sourcemaking.com/design_patterns/chain_of_responsibility
- https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern