Name: Chain of Responsibility

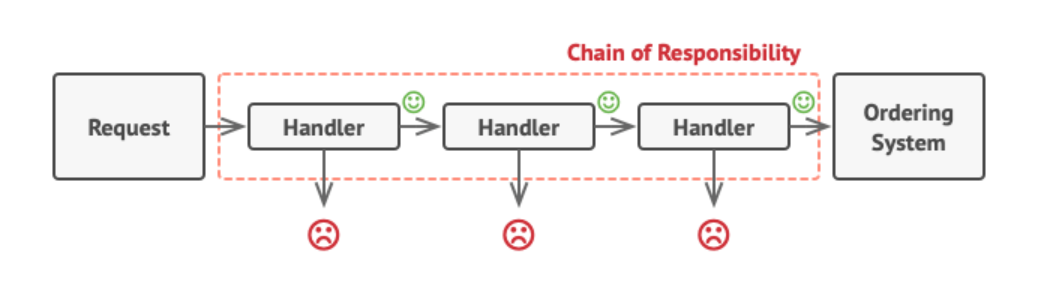
Classification: Chain of Responsibility belongs to Behavioral Pattern

Intent: Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Motivation:

* Problem: when clients send many requests to server, we need to perform check for each request which is large and slow.
* Solution if not using CoR: use cache to reduce the number of checking time. For example, when user make requests many times in short interval, we can save the old result in cache so the server doesn’t need to check again

Solution:



* Create many handler in a chain, 1 handler fail will stop the flow so that server doesn’t need to check with other handlers

Example: Example: Checking an integer is positive, negative or equal to zero.

Source code:

**public** **class** Chain

{

Processor chain;

**public** Chain(){

    buildChain();

}

**private** **void** buildChain(){

    chain = **new** NegativeProcessor(**new** ZeroProcessor(**new** PositiveProcessor(**null**)));

}

**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**

        {

**super**.process(request);

        }

    }

}

**class** PositiveProcessor **extends** Processor

{

**public** PositiveProcessor(Processor nextProcessor){

**super**(nextProcessor);

    }

**public** **void** process(Number request)

    {

**if** (request.getNumber() > 0)

        {

            System.out.println("PositiveProcessor : " + request.getNumber());

        }

**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));

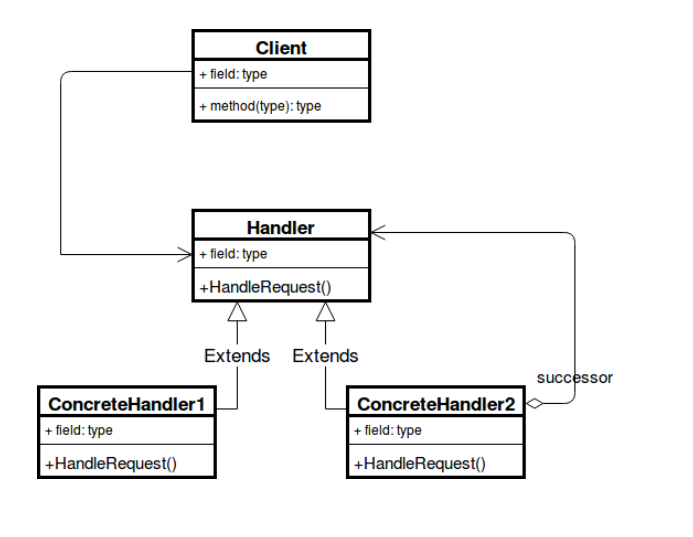
        chain.process(**new** Number(0));

        chain.process(**new** Number(91));

    }

}

Class diagram:

Pros:

* You can control the order of request handling.
* *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform operations.
* *Open/Closed Principle*. You can introduce new handlers into the app without breaking the existing client code.

Cons:

* Some requests may end up unhandled.

Applicability: Ordering system with 2 roles. Normal user can view some products. Admin can view all product. Each has its own checking function. When adding many functionalities, the checking step would be very slow, and the code is large.

* Using this pattern, we can make the checking step optimized.