
# LECTURE 08

## **Chain of Responsibility** and **Strategy**

Software Development Principle 94~110

# Chain of Responsibility (职责链条)

## Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

## Motivation

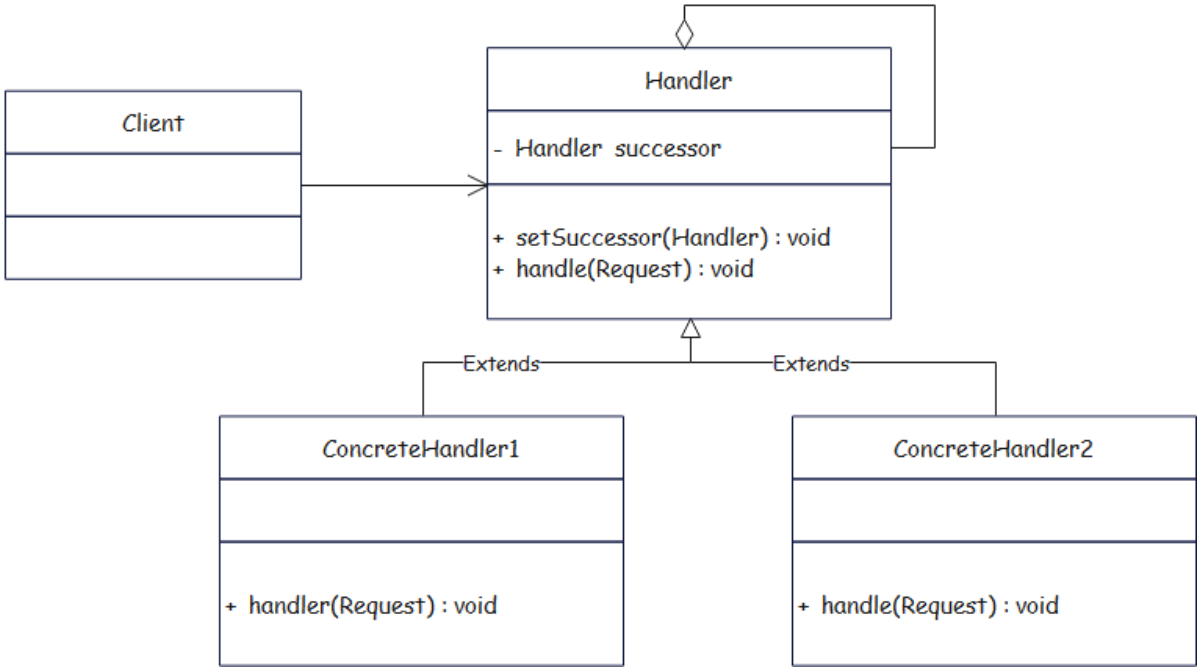
The problem here is that the object that ultimately provides the help isn't known explicitly to the object (e.g., the button) that initiates the help request. What we need is a way to decouple the button that initiates the help request from the objects that might provide help information.

## Applicability

Use Chain of Responsibility when:

- more than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

* Class Diagram



## Participants

### Handle

- defines an interface for handling requests.
- (optional) implements the successor link.

### ConcreteHandle

- handles requests it is responsible for.
- can access its successor.
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

### Client

- initiates the request to a ConcreteHandler object on the chain.

## Collaborations

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

## Consequences

Chain of Responsibility has the following benefits and liabilities:

a) Reduced coupling.

b) Added flexibility in assigning responsibilities to objects.

c) Receipt isn't guaranteed.

## Implementation

Here are implementation issues to consider in Chain of Responsibility:

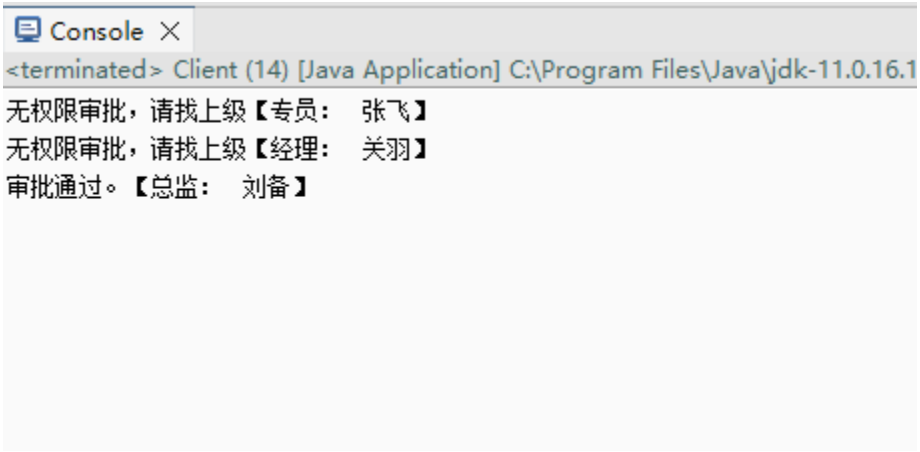
- Implementing the successor chain.
- Connecting successors.
- Representing requests.

```java
package approval;

public class Staff {
    private String name;
    public Staff(String name) {
        this.name = name;
    }
    public boolean approve(int amount) {
        if (amount <= 1000) {
            System.out.println("审批通过。【专员:  " + name + "】");
            return true;
        }
        else {
            System.out.println("无权限审批，请找上级【专员:  " + name + "】");
            return false;
        }
    }

}
```

```java
package approval;

public class Manager {
    private String name;
    public Manager(String name) {
        this.name = name;
    }
    public boolean approve(int amount) {
        if (amount <= 5000) {
            System.out.println("审批通过。【经理:  " + name + "】");
            return true;
        }
        else {
            System.out.println("无权限审批，请找上级【经理:  " + name + "】");
            return false;
        }
    }
}
```

```java
public class CFO {
    private String name;
    public CFO(String name) {
        this.name = name;
    }
    public boolean approve(int amount) {
        if (amount <= 10000) {
            System.out.println("审批通过。【总监: " + name + "】");
            return true;
        }
        else {
            System.out.println("驳回申请。【总监: " + name + "】");
            return false;
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        int amount = 10000;
        Staff staff = new Staff("张飞");
        if (!staff.approve(amount)) {
            Manager manager = new Manager("关羽");
            if(!manager.approve(amount)) {
                CFO cfo = new CFO("刘备");
                cfo.approve(amount);
            }
        }
    }
}
```


Console
<terminated> Client (14) [Java Application] C:\Program Files\Java\jdk-11.0.16.1
无权限审批，请找上级【专员：　张飞】
无权限审批，请找上级【经理：　关羽】
审批通过。【总监：　刘备】

```java
public abstract class Approver {
    protected String name;
    protected Approver nextApprover;
    public Approver(String name) {
        this.name = name;
    }
    protected Approver setNextApprover(Approver nextApprover) {
        this.nextApprover = nextApprover;
        return this.nextApprover;
    }
    public abstract void approve(int amount);

}
```

```java
public class Staff extends Approver{
    public Staff(String name) {
        super(name);
    }
    @Override
    public void approve(int amount) {
        if (amount <= 1000) {
            System.out.println("审批通过。【专员: " + name + "】");
        }
        else {
            System.out.println("无权限审批，请找上级【专员: " + name + "】");
            this.nextApprover.approve(amount);
        }
    }

}
```

```java
public class Manager extends Approver {

    public Manager(String name) {
        super(name);
    }
    @Override
    public void approve(int amount) {
        if (amount <= 5000) {
            System.out.println("审批通过。【经理:  " + name + "】");
        }
        else {
            System.out.println("无权限审批，请找上级【经理:  " + name + "】");
            this.nextApprover.approve(amount);
        }
    }
}
```

```java
public class CFO extends Approver{
    public CFO(String name) {
        super(name);
    }
    @Override
    public void approve(int amount) {
        if (amount <= 10000) {
            System.out.println("审批通过。【总监:  " + name + "】");
        }
        else {
            System.out.println("驳回审批。【总监:  " + name + "】");
        }
    }
}
```

```java
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Approver flightJohn = new Staff("张飞");
        //使用链式编程配置责任链
        flightJohn
        .setNextApprover(new Manager("关羽"))
        .setNextApprover(new CFO("刘备"));
        flightJohn.approve(100);//报销打车
        flightJohn.approve(1500);//报销应酬
        flightJohn.approve(6000);//报销出差
        flightJohn.approve(16000);//报销公费买4090
    }

}
```


Console
<terminated> Client (15) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\ja
审批通过。【专员： 张飞】
无权限审批，请找上级【专员： 张飞】
审批通过。【经理： 关羽】
无权限审批，请找上级【专员： 张飞】
无权限审批，请找上级【经理： 关羽】
审批通过。【总监： 刘备】
无权限审批，请找上级【专员： 张飞】
无权限审批，请找上级【经理： 关羽】
驳回审批。【总监： 刘备】

# Strategy (策略模式)

## Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

## Also known as

Policy

## Motivation

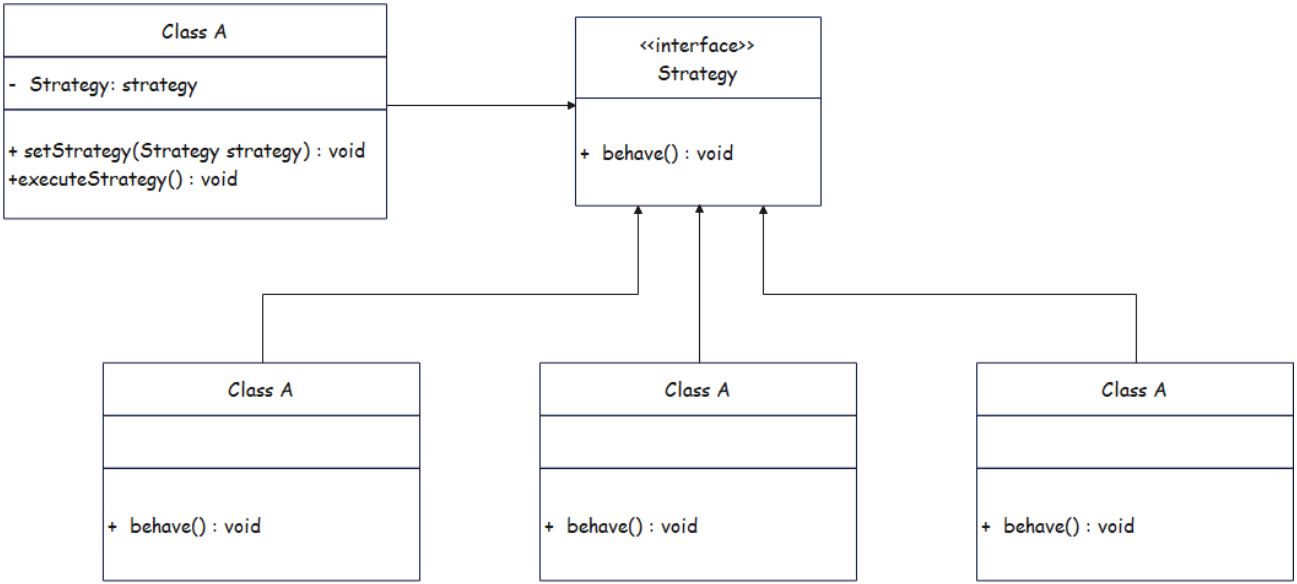
Some problems can be avoided by defining classes that encapsulate different linebreaking algorithms.

## Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior.
- you need different variants of an algorithm.
- an algorithm uses data that clients shouldn't know about.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

* Class Diagram



## Participants

### Strategy

- declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

### ConcreteStrategy

- implements the algorithm using the Strategy interface.

### Context

- is configured with a ConcreteStrategy object.
- maintains a reference to a Strategy object.
- may define an interface that lets Strategy access its data.

## Collaborations

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.

## Consequences

The Strategy pattern has the following benefits and drawbacks:

Families of related algorithms.

An alternative to subclassing.

Strategies eliminate conditional statements.

A choice of implementations.

Clients must be aware of different Strategies.

Communication overhead between Strategy and Context.

Increased number of objects.

## Implementation

*Consider the following implementation issues:*

- *Defining the Strategy and Context interfaces.*
- Strategies as template parameters.
- Making Strategy objects optional.

```java
public class Calculator {
    public int add(int a, int b) {
        return a+b;
    }
    public int sub(int a, int b) {
        return a-b;
    }
}
```

```java
public interface Strategy {
    public int calculate(int a, int b);
}
```

```java
public class Addition implements Strategy{
    @Override
    public int calculate(int a, int b) {
        return a+b;
    }
}
```

```java
public class Subtraction implements Strategy{
    @Override
    public int calculate(int a, int b) {
        return a-b;
    }

}
```

```java
public class Calculator {
    private Strategy strategy;
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }
    public int getResult(int a, int b) {
        return this.strategy.calculate(a, b);
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        calculator.setStrategy(new Addition());
        System.out.println(calculator.getResult(561654, 1552));

        calculator.setStrategy(new Subtraction());
        System.out.println(calculator.getResult(1234, 233));
    }
}
```

Console

<terminated> Client (16) [Java Application] C:\Prograr

```
563206
1001
```

```java
public interface USB {
    public void read();
}
```

```java
public class Mouse implements USB {
    @Override
    public void read() {
        System.out.println("键盘指令数据……");
    }
}
```

```java
public class KeyBoard implements USB{
    @Override
    public void read() {
        System.out.println("键盘指令数据……");
    }
}
```

```java
public class Camera implements USB{
    @Override
    public void read() {
        System.out.println("视频流数据……");
    }
}
```

```java
public class Computer {
    private USB usb;
    public void setUSB(USB usb) {
        this.usb = usb;
    }
    public void compute() {
        usb.read();
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Computer com = new Computer();
        com.setUSB(new KeyBoard());
        com.compute();
        com.setUSB(new Mouse());
        com.compute();
        com.setUSB(new Camera());
        com.compute();
    }
}
```

```
Console
<terminated> Client (17) [Java Application] C:\Program Files\Jav
键盘指令数据.....
键盘指令数据.....
视频流数据.....
```

# Principle 94

**Get it right before you make it faster**

It is far easier to adapt a working program to make it run faster than to adapt a fast program to make it work. Don't worry about optimization when doing your initial coding. [On the other hand, don't use a ridiculously inefficient algorithm or set of data structures (Principles 79 and 93).]

# Principle 95

**Comment before you finalize your code**

I've often heard programmers say, "Why should I bother commenting my code now? It'll only change!" We comment code to make the software easi-er to debug, test, and maintain. By commenting your code while coding (or beforehand, see Principle 96),it will be easier for you to debug the software.

# Principle 96

**Document before you start coding**

96.This advice will seem strange to some readers, but it becomes natural after being practiced for a while. Principle 95 explained why you should document your code before finishing it. Principle 96 goes one step further: You should document your code before starting to code!

# Principle 97

**Hand-execute every component**

It might take 30 minutes to execute a software component by hand with a few simple test cases. Do it! I am suggesting this in addition to, not in lieu of, the more thorough computer-based unit testing that is already being performed.

# Principle 98

## Inspect code

Inspection of software detailed design and code was first proposed by Michael Fagan in his paper entitled"Design and Code Inspections to Reduce Errors in Program Development"[IBM Systems Journal,15,3 (July 1976), pp.182-211]. It can account for as many as 82 percent of all errors found in software. Inspection is much better than testing for finding errors. Define criteria for completing an inspection. Keep track of the types of errors found through inspection. Fagan's inspections consume approximately 15 percent of development resources with a net reduction in total development cost of 25 to 30 percent.

# Principle 99

**You can use unstructured languages**

Unstructured code violates Edsger Dijkstra's guidance to restrict control structures to IF-THEN-ELSE, DO-WHILE, DO-UNTIL, and CASE. Notice that it is possible to write structured code in languages without these structures, such as in assembly languages, by documenting the code with the structured control statements and restricting the use of coro's to implementing these structures only.

# Principle 100

**Structured code is not necessarily good code**

A variety of simple techniques can be used to reduce nesting. See the following refer-ence for examples and techniques.

# Principle 101

**Don't nest too deep**

Nesting IF-THEN-ELSE statements greatly simplifies programming logic. On the other hand, nesting them more than, say, three levels decreases their understandability considerably. The human mind is capable of remembering only a certain amount of logic before it becomes confused. A variety of simple techniques can be used to reduce nesting. If your application requires a great use of character strings or complex data structures, select a language that supports them.

# Principle 102

**Use appropriate languages**

If your application requires a great use of character strings or complex data structures, select a language that supports them. If your product must be maintained by a group of existing maintainers who know language X, then use language X. Finally, if your customer says,"Thou shalt use language Y," then use language Y or you won't be in business long.

# Principle 103

## Programming languages is not an excuse

Some projects are forced to use a less-than-ideal programming language. This might be caused by a desire to reduce maintenance costs("All our maintainers know COBOL"), to program fast ("We have the highest pro-ductivity with C"), to ensure high reliability("Ada programs are the most fail-safe"), or to achieve high execution speed ("Our applications are so time-critical, we need to use assembly language"). It is possible to write quality programs in any language. In fact, if you are a good programmer, you should be a good programmer in any language (Principle 104); a less-than-ideal language might make you work harder, though.

# Principle 104

## Language knowledge is not so important

Good programmers are good regardless of the language used. Poor programmers are poor regardless of the language used. Nobody is a"great C programmer" and a"poor Ada programmer." If they really are poor at Ada, they probably were not great at C! In addition, a really good programmer should be able to learn any new language easily. This is because a really good programmer understands and appreciates the concepts of quality programming, not just the syntactic and semantic idiosyncrasies of the primary driver of language selection for a project should be appropriateness(Principle 102), not the surge of programmers who whine, "But all we know is C."If some quit because the project selected a different language, the project is probably better off!

# Principle 105

## Format your programs

The understandability of a program is greatly enhanced by using standard indentation protocols. Which protocol you choose to follow matters little, but, once you select it, use it consistently.

# Principle 106

**Don't code too soon**

Coding software is analogous to constructing a building. Both require much preliminary work. Constructing a building without a solid and stable concrete foundation will not work. Coding without a solid and stable foundation of requirements and design will not work.

# Testing principles

Think about how much more difficult it is to modify a building after the foundation is poured!

1. Performing tests on individual software components (that is, unit testing) to conclude that they are sufficiently close to behaving as specified in the component's design specification.
2. Performing tests on sets of unit-tested components (integration testing)to conclude that they behave as a team in a manner close enough to how they were specified in the design.
3.Performing tests on the entirely integrated set of software components(software systems-level testing) to conclude that they behave as a system in a manner sufficiently close to that specified in the software requirements specification.
4. Generating test plans for software systems-level testing.
5.Generating test plans for software integration testing.
6.Generating test plans for unit testing.
7.Building test harnesses and test environments.

# Principle 107

**Trace tests to requirements**

It is important to understand which tests verify which requirements. There are two reasons: (1) When generating tests, you'll find it useful to know if all requirements are being tested. (2) When performing tests, you'l find it useful to know which requirements are being checked. Furthermore, if your requirements have been prioritized (Principle 50), you can easily derive the relative priorities of tests; that is, the priority of a test is the maximum of the priorities of all its corresponding requirements

# Principle 108

**Don't test your own software**

Often software developers create their software product, then scratch their heads and say,"Now, how are we going to test this thing?" Test planning is a major task and must occur in parallel with product development so that test planning and initial (that is, pretesting) development activities are completed in synchrony.

# Principle 109

**Structured code is not necessarily good code**

Software developers should never be the primary testers of their own soft-ware. It is certainly appropriate to do initial debugging and unit testing. Independent testers are necessary:

1. To check a unit for adequacy before starting integration testing.

2.For all integration testing. 3.For all software system testing.

# Principle 110

**Don't write your own test plans**

Not only should you not test your own software (Principle 109), but you should also not be responsible for generating the test data, test scenarios, or test plans for your software.If you are, you may make the same mistakes in test generation that you made in software creation. For example, if you made a false assumption about the range of legal inputs when engineering the software, you would likely make the same assumption when generating test plans.

# Principle 111

Testing exposes presence of flaws

No matter how thorough, testing simply exposes the presence of flaws in a program; it cannot be used to verify the absence of flaws. It can increase your confidence that a program is correct, but it cannot prove correctness. To gain true correctness, one must use completely different processes, that is, correctness proofs.

# Principle 112

**Though copious errors guarantee worthlessness, zero error says nothing about the value of software**

This is Gerald Weinberg's"Absence of Errors Fallacy."It really puts testing into perspective. It also puts all software engineering and management into perspective. The first part of the principle is obviously true; software with many errors is useless. The second part provides food for thought. It says that, no matter how hard you work to remove errors, you are wasting your time unless you are building the right system.

# Principle 113

**A successful test finds an error**

I have often heard a tester gleefully declare,"Great news! My test was suc-cessful. The program ran correctly." This is the wrong attitude to have when running a test. [It also supports the position that programmers should never test their own software (Principle 109).] A more constructive attitude it that one is testing to find errors. Thus, a successful test is one that detects an error.

# Principle 114

**Half the errors found in 15 percent of modules**

Conservative estimates indicate that, in large systems, approximately half of all software errors are found in 15 percent of the modules, and 80 per-cent of all software errors are found in 50 percent of the modules.More dramatic results from Gary Okimoto and Gerald Weinberg indicate that 80 percent of all errors were found in just 2 percent of the modules.

# Principle 115

**Use black-box and white-box testing**

Black-box testing uses the specification of a component's external behavior as its only input. It is mandatory to determine if the software does what it is supposed to do and doesn't do what it is not supposed to do. White-box testing uses the code itself to generate test cases. By combining black-box and white-box, you maximize the effectiveness of testing. Neither one by itself does a thorough test.