Chain of Responsability

Use the **Chain of Responsibility** Pattern when you want to give more than one object a chance to handle a request.

Toni Sellarès
Universitat de Girona

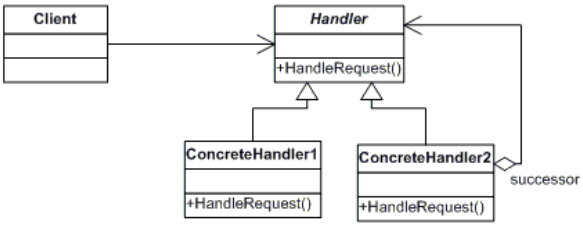
Chain of Responsibility: Motivation

- The Chain of Responsibility is intended to promote loose coupling between the sender of a request and its receiver by giving more than one object an opportunity to handle the request.
- The receiving objects are chained and pass the request along the chain until one of the objects handles it.
- The set of potential request handler objects and the order in which these objets form the chain can be decide dynamically at runtime by the client depending on the current state of the application.

Chain of Responsibility: Definition

Use the **Chain of Responsibility** Pattern when you want to give more than one object a chance to handle a request.

Participants



Handler:

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

ConcreteHandler:

- 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

Chain of Responsibility: Structural Code

```
/**
 * Test driver for the pattern.
 */

public class Test {
        public static void main( String arg[] ) {
                Handler handler1 = new ConcreteHandler1();
                Handler handler2 = new ConcreteHandler2();
                Handler handler3 = new ConcreteHandler3();
                Handler handler4 = new ConcreteHandler4();

                handler1.setSuccessor( handler2 );
                handler2.setSuccessor( handler3 );
                handler3.setSuccessor( handler4 );

                handler1.handleRequest();
                }
}
```

```
/**
 * Defines an interface for handling requests. Implements the successor link.
 */

public class Handler {
        private Handler successor;

        public void setSuccessor( Handler successor ) {
                this.successor = successor;
        }

        public Handler getSuccessor() {
                return successor;
        }

        public void handleRequest()
        {
                successor.handleRequest();
        }
}
```

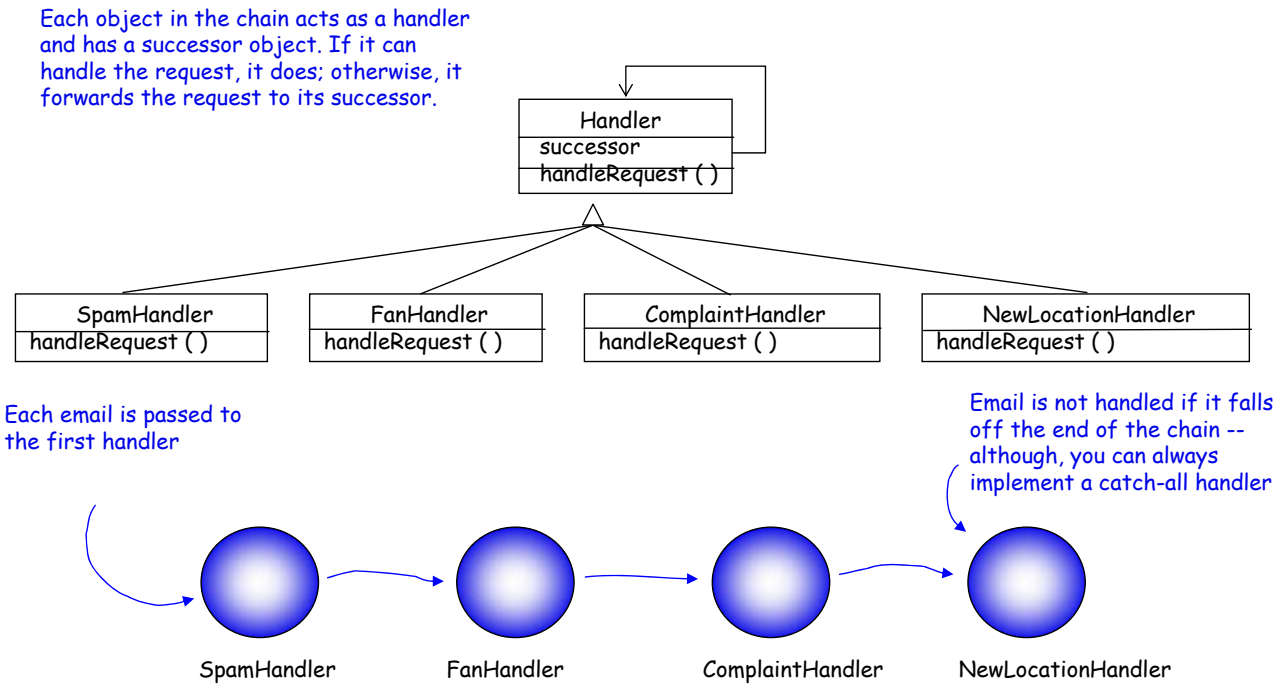
```
/**
 * Handles requests it is responsible for. Can access its successor. If the
 * ConcreteHandler can handle the request, it does so; otherwise it forwards the
 * request to its successor.
 */

public class ConcreteHandler1 extends Handler {
        public void handleRequest() {
               if( /* We should handle this == */ true ) {
                        // Handle the request.
                }
                else
                getSuccessor().handleRequest();
        }
}
```

Chain of Responsibility: Example

Scenario

- An enterprise has been getting more email than they can handle.
- They get:
  - Fan mail from customers,
  - Complaints from customers,
  - Requests for new machines,
  - Spam.
- Your task:
  - The enterprise has already written the AI detectors that can tell whether an email is fan, complaint, request or spam,
  - You need to create a design that can use the detectors to handle incoming email.



Chain of Responsibility: Benefits, Uses and Drawbacks

- Benefits:
  - Decouples the sender of the request and its receivers,
  - Simplifies your object because it doesn’t have to know the chain’s structure and keep direct reference to its members,
  - Allows you to add or remove responsibilities dynamically by changing the members or the order of the chain.
- Uses:
  - Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- Drawbacks:
  - Execution of the request isn’t guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage),
  - Can be hard to observe the runtime characteristics and debug.