SE 350 - Object-Oriented Software Development

Software Design Patterns: Chain of Responsibility

Instructor: Stefan Mitsch

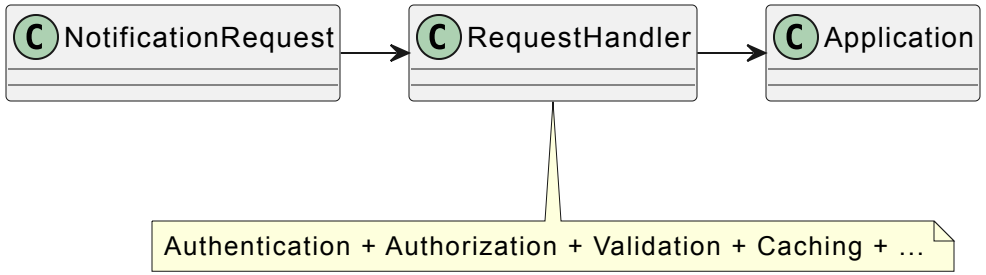
Learning Objectives

- Understand the problem solved by Chain of Responsibility
- Understand the consequences of using the pattern

Design Challenge

❓ What if the `Receiver` of a `Command` could be one of multiple objects and we do not know a priori which one?

❓ What if we want to execute `Receiver` objects sequentially to do multiple checks on a command?



⚠ Inflexible request handling code

⚠ Cannot easily reconfigure and skip steps

Chain of Responsibility Introduction

Chain of Responsibility avoids coupling between sender and receiver by allowing multiple request handlers

- Behavioral design pattern
- Allow multiple handlers to address a request sequentially

Problem illustration:

❓ How to avoid inflexible request handling code?

```
public class RequestHandler {
  public Response handle(NotificationRequest r) {
    if (authenticate(r)) {
      if (authorize(r)) {
        if (validate(r)) {
          // ...
        } else return new InvalidRequestResponse();
      } else return new UnauthorizedResponse();
    } else return new AuthenticationFailedResponse();
  }
}
```

Chain of Responsibility Overview

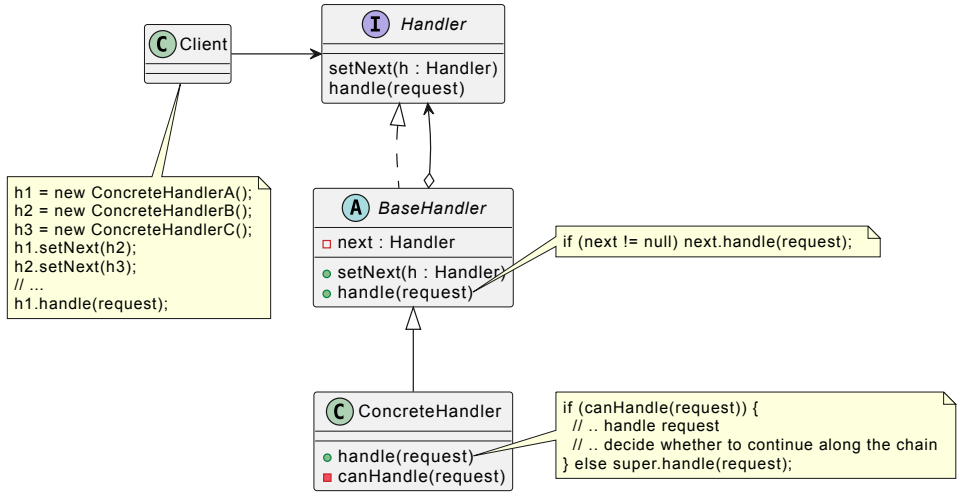
Problem

Allow multiple handlers for a request

Intent

- Support sequential manipulation of the same command/request
- Use when order of handlers can change at runtime
- Use when type of handler and execution order is unknown at design time

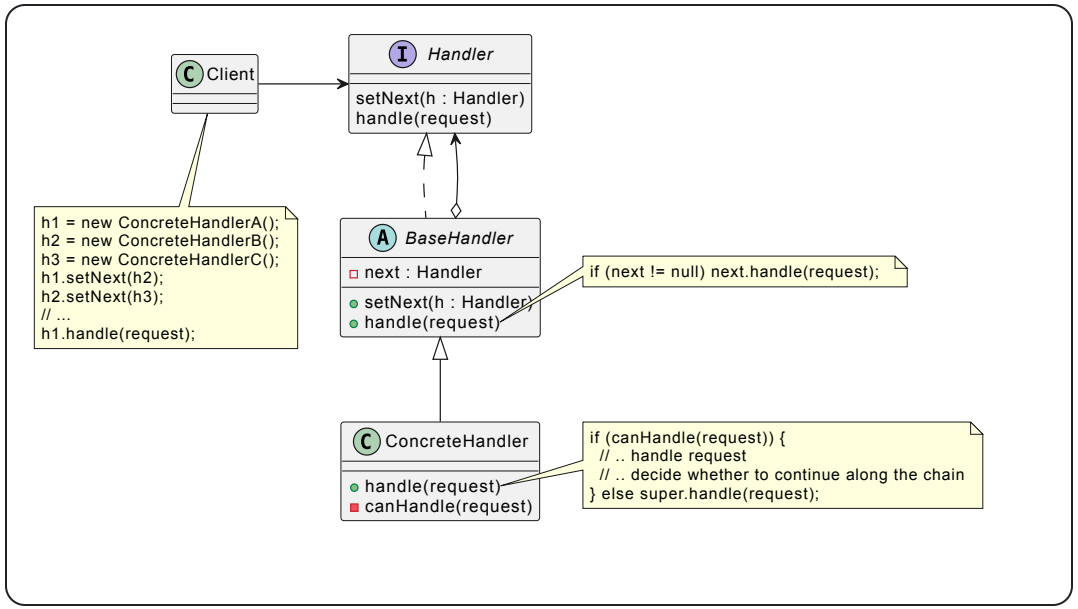
Structure



Chain of Responsibility Participants

1. The `Handler` declares an common interface for handling requests

2. The `BaseHandler` is optional; if present, it contains fields for storing the next handler and a default implementation to delegate to the next handler



3. The `ConcreteHandler` objects contain the actual code for processing requests (e.g., authentication, authorization, validation); they also know whether to continue processing along the chain

4. The `Client` composes chains and may reconfigure them at runtime; it also sends requests to the chain

Chain of Responsibility Implementation

- Declare the `Handler` interface for handling requests
- Decide how the client will pass requests to handlers; the most flexible way is using `Command`
- Avoid duplicate boilerplate code with an abstract `BaseHandler`
- Implement `ConcreteHandlers` : decide how a concrete handler handles a request, and whether it passes it on along the chain

```java
// request interface (command pattern)
public interface NotificationRequest {
  String getMessage();
  Instant getSchedule();
}

// handler interface
public interface NotificationRequestHandler {
  void handle(NotificationRequest r);
}

// base handler
public abstract class BaseNotificationRequestHandler {
  private NotificationRequestHandler next;
  public void setNext(NotificationRequestHandler next) {
    this.next = next;
  }
  public Response handle(NotificationRequest r) {
    if (next != null) return next.handle(r);
    else return new UnhandledRequestResponse(r);
  }
}

// concrete handler
public class ValidateNotificationRequest extends BasenNotificationRequestHandler {
  public Response handle(NotificationRequest r) {
    if (r.getMessage() != null && !r.getMessage().isEmpty()) super.handle(r);
    else return new EmptyMessageResponse();
  }
}
```

Chain of Responsibility Discussion

Benefits

- Control the order of request handling
- Promotes the Single Responsibility Principle: multiple handler steps each in their own class
- Promotes the Open/Closed Principle: can easily introduce new handlers

Drawbacks

- Some requests may end up unhandled