Behavioral Design Pattern

Hung Tran

Fpt software

December 1, 2021

Outline

Behavioral Pattern Overview

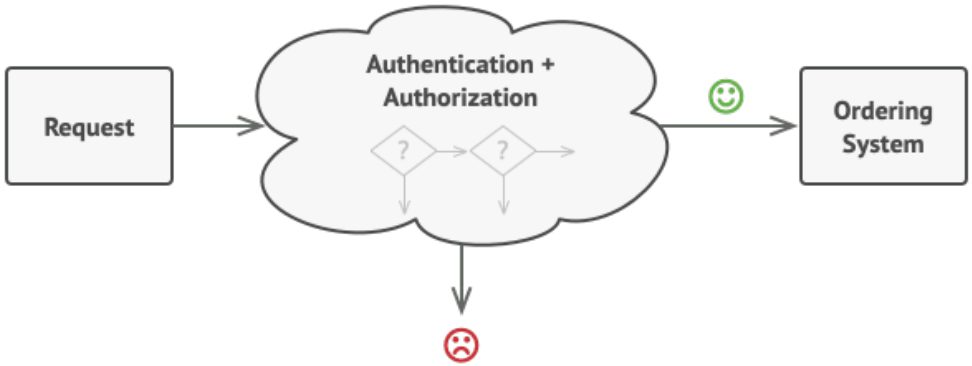
Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

- **Chain of responsibility**: lets you pass requests along a chain of handlers.
- **Command**: turns a request into a stand-alone object that contains all information about the request.
- **Iterator**: lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- **Mediator**: lets you reduce chaotic dependencies between objects.
- **Memento**: lets you save and restore the previous state of an object without revealing the details of its implementation.

Behavioral Pattern Overview

- **Observer**: lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- **State**: lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- **Strategy**: lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- **Template Method**: defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- **Visitor**: lets you separate algorithms from the objects on which they operate.

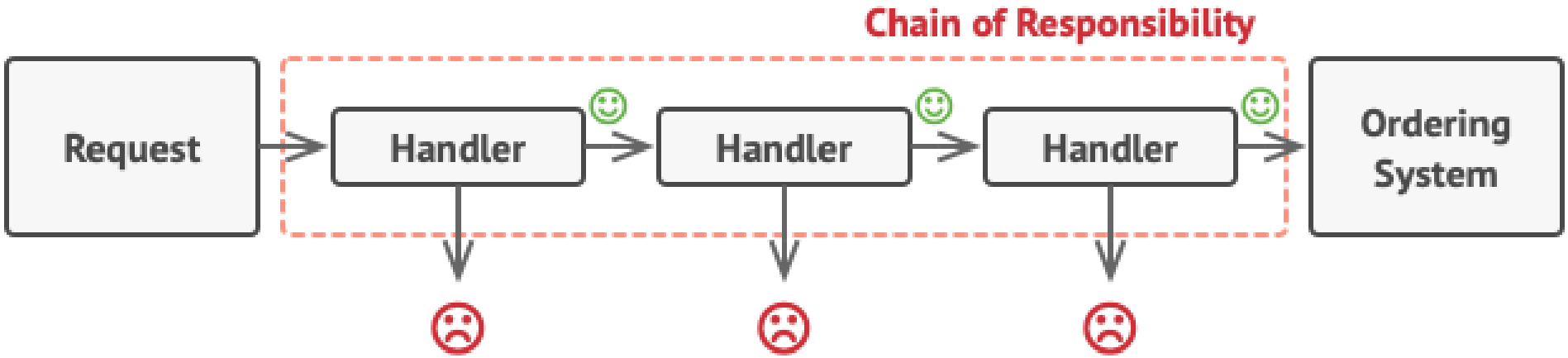
Problem Statement: online ordering system



- Restrict access to the system so only authenticated users can create orders.
- Users who have administrative permissions must have full access to all orders.
- These checks must be performed sequentially.

- Validation step to sanitize the data in a request.
- Check that filters repeated failed requests coming from the same IP address.
- Speed up the system by returning cached results on repeated requests containing the same data. Hence, you added another check which lets the request pass through to the system only if there's no suitable cached response.

Solution: Chain of Responsibility Pattern

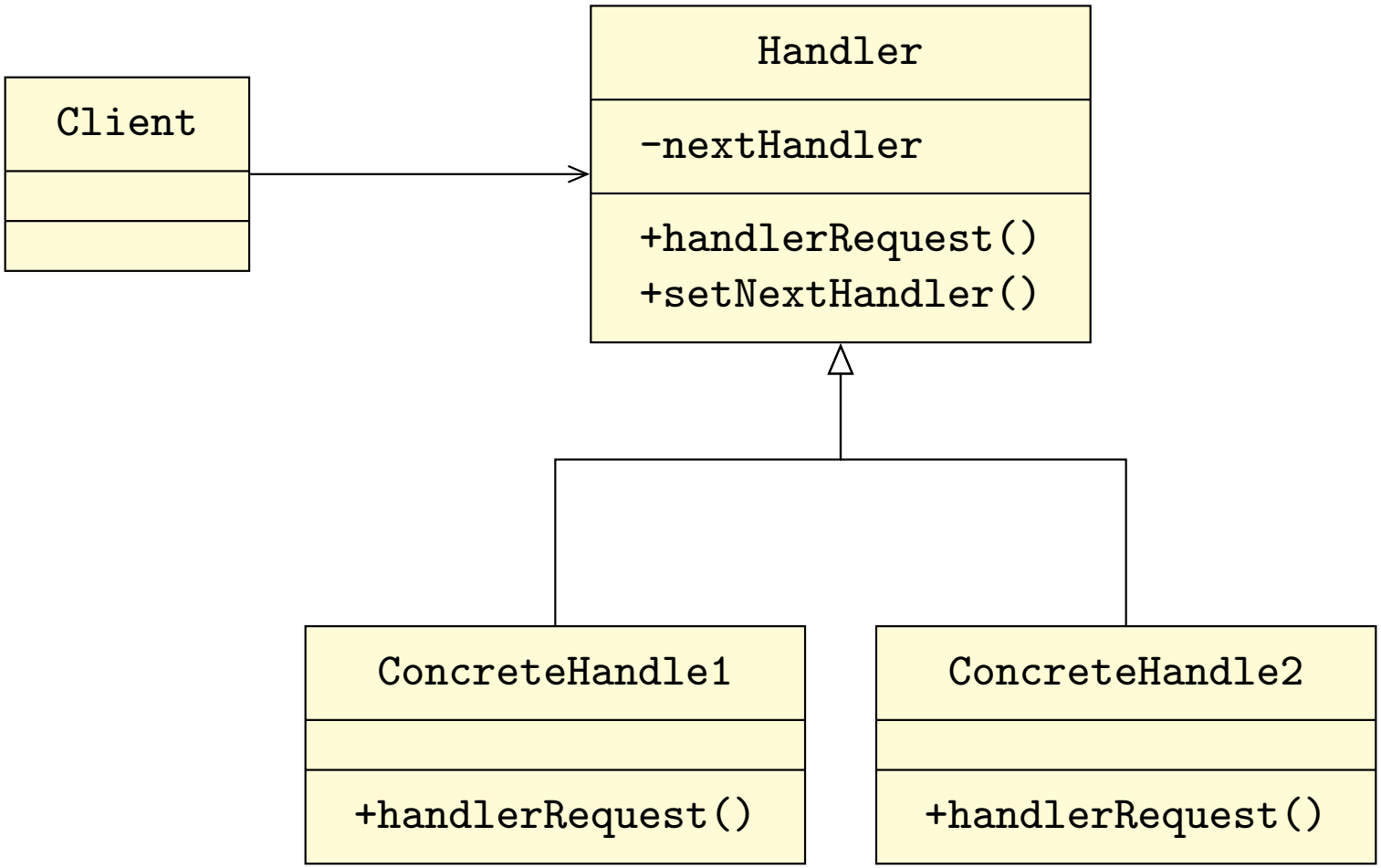


- Each check should be extracted to its own class with a single method that performs the check.
- Each linked handler has a field for storing a reference to the next handler.
- A handler can decide not to pass the request further down the chain and effectively stop any further processing

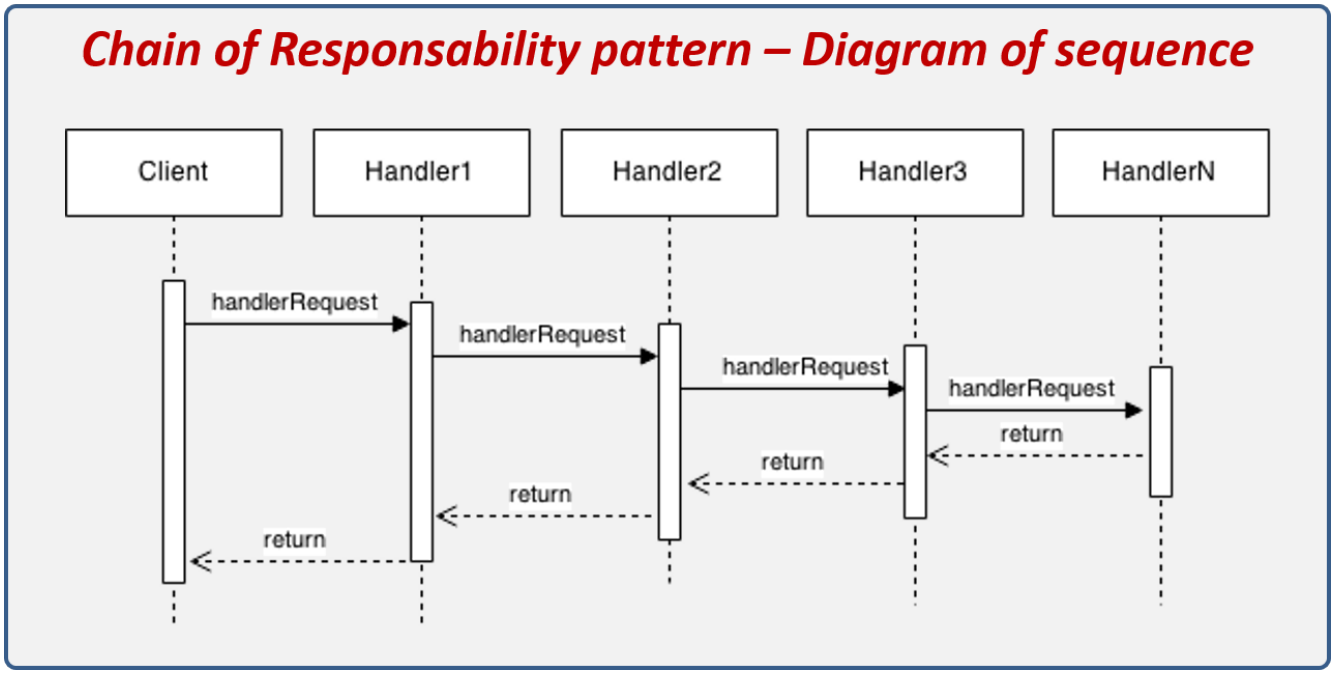
The Intent of Chain of Responsibility Design Pattern

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.

Structure of Chain of Responsibility Pattern: Object adapter



Sequence Diagram



Basic implementation: handler class

handler.h

```
1  #ifndef _HANDLER_H_
2  #define _HANDLER_H_
3  class Handler {
4  private:
5    Handler* successor;
6  public:
7    virtual ~Handler() = default;
8    virtual void setNextHandler(Handler* s
       );
9    virtual void handlerRequest();
10 };
11 #endif // _HANDLER_H_
```

handler.cpp

```
1  #include "handler.h"
2
3  void Handler::setNextHandler(Handler* s)
       {
4    successor = s;
5  }
6
7  void Handler::handlerRequest() {
8    if (successor != 0) {
9      successor->handlerRequest();
10   }
11 }
```

Basic implementation: concreteHandler1 class

concreteHandler1.h

```
1  #ifndef _CONCRETE_HANDLER_1_H_
2  #define _CONCRETE_HANDLER_1_H_
3  #include "handler.h"
4  class ConcreteHandler1 : public Handler
       {
5  public:
6    ~ConcreteHandler1() = default;
7    bool canHandle();
8    virtual void handlerRequest();
9  };
10 #endif // _CONCRETE_HANDLER_1_H_
```

concreteHandler1.cpp

```
1  #include "concreteHandler1.h"
2  #include <iostream>
3
4  bool ConcreteHandler1::canHandle() {
5    return false;
6  }
7
8  void ConcreteHandler1::handlerRequest()
       {
9    if (canHandle()) {
10     std::cout << "Handled by concrete
         handler 1" << std::endl;
11   }
12   else {
13     std::cout << "Can not handled by
         handler 1" << std::endl;
14     Handler::handlerRequest();
15   }
16 }
```

Basic implementation: concreteHandler2 class

concreteHandler2.h

```
1  #ifndef _CONCRETE_HANDLER_2_H_
2  #define _CONCRETE_HANDLER_2_H_
3  #include "handler.h"
4  class ConcreteHandler2 : public Handler
       {
5  public:
6    ~ConcreteHandler2() = default;
7    bool canHandle();
8    virtual void handlerRequest();
9  };
10 #endif // _CONCRETE_HANDLER_2_H_
```

concreteHandler2.cpp

```
1  #include "concreteHandler2.h"
2  #include <iostream>
3
4  bool ConcreteHandler2::canHandle() {
5    return false;
6  }
7
8  void ConcreteHandler2::handlerRequest()
       {
9    if (canHandle()) {
10     std::cout << "Handled by concrete
         handler 2" << std::endl;
11   }
12   else {
13     std::cout << "Can not handled by
         handler 2" << std::endl;
14     Handler::handlerRequest();
15   }
16 }
```

Basic implementation: main

main.cpp

```
1  #include "concreteHandler1.h"
2  #include "concreteHandler2.h"
3
4  int main() {
5    ConcreteHandler1 handler1;
6    ConcreteHandler2 handler2;
7
8    handler1.setNextHandler(&handler2);
9    handler1.handlerRequest();
10
11   return 0;
12 }
```

Applicability

- When we can conceptualize our program as a chain made up of links, where each link can either handle a request or pass it up the chain.
- When we want to decouple a request's sender and receiver.
- Multiple handlers determined at runtime.
- More than one object may handle a request, and the handler is not known in advance.
- The handler should be ascertained automatically.
- We may want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of handlers that can handle a request should be specified dynamically.
- A scenario within you need to pass a request to one handler among a list of handlers at run-time based on certain conditions.

How to Implement

- Declare the handler interface and describe the signature of a method for handling requests.
- To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.
- One by one create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:.
- The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.
- The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.

Pros and Cons

- 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.
- Some requests may end up unhandled.

Relations with Other Patterns

- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- Chain of Responsibility is often used in conjunction with Composite. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.
- Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.
- Chain of Responsibility and Decorator have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

Thank You!