Problem No. 8
Problem Name: Implement and design Chain of Responsibility design pattern(for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Chain of Responsibility (CoR) Design Pattern. The pattern is used to pass requests along a chain of handlers until one of them handles it. It promotes loose coupling between sender and receiver of a request.

Theory:

The Chain of Responsibility Pattern is a behavioral design pattern that lets you build a chain of objects to process a request. Each handler in the chain either processes the request or forwards it to the next handler.

Key Components:

1. **Handler:** Declares an interface for handling requests and optionally sets the next handler.
2. **ConcreteHandler:** Handles the request it is responsible for; otherwise forwards it to the next handler.
3. **Client:** Initiates and sends the request to the chain.

This pattern avoids coupling the sender to the receiver and allows adding/removing handlers dynamically.

Application:

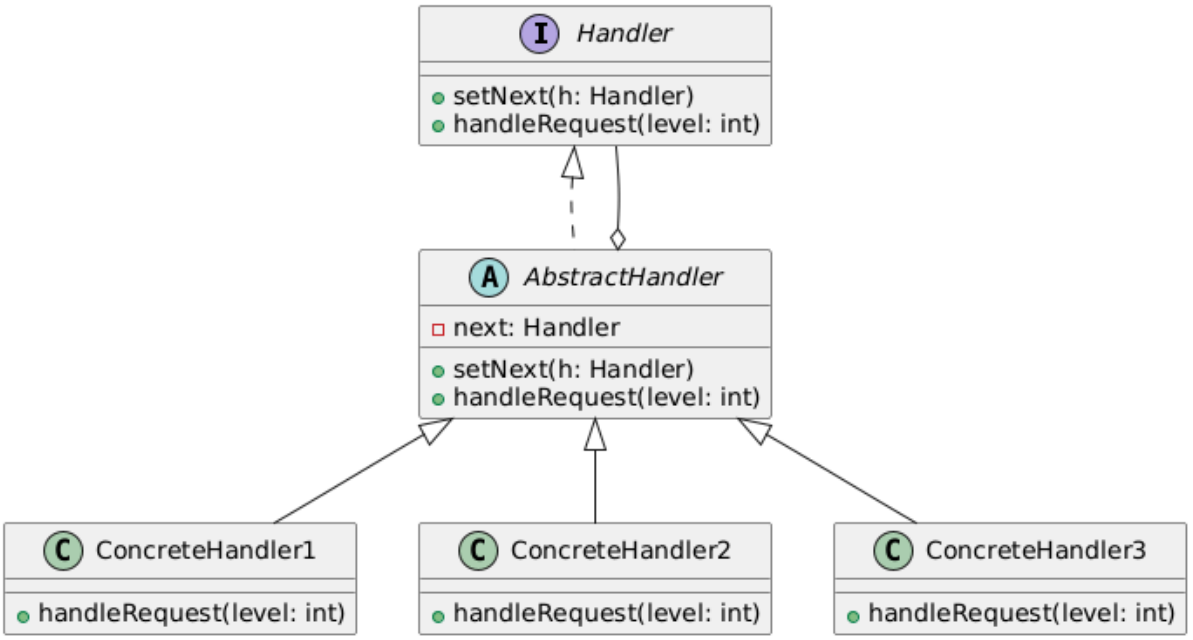
The Chain of Responsibility pattern is useful when:

1. Multiple objects can handle a request, and the handler is determined at runtime.
2. You want to decouple sender and receiver.
3. You want to specify a chain of processing objects dynamically.

Examples:

1. Exception handling systems.
2. Event handling in GUI frameworks.
3. Logging frameworks (debug → info → warning → error).
4. Customer support request escalation systems.

UML Design:



Implementation in C++:

```
#include <iostream>
using namespace std;

// Handler interface
class Handler {
public:
    virtual Handler* setNext(Handler* handler) = 0;
    virtual void handleRequest(int level) = 0;
    virtual ~Handler() = default;
};

// Abstract Handler
class AbstractHandler : public Handler {
protected:
    Handler* next = nullptr;
public:
    Handler* setNext(Handler* handler) override {
        next = handler;
        return handler;
    }
    void handleRequest(int level) override {
        if (next)
            next->handleRequest(level);
    }
};

// Concrete Handler 1
class ConcreteHandler1 : public AbstractHandler {
public:
```

```
    void handleRequest(int level) override {
        if (level == 1) {
            cout << "Handler1 processed request of level 1" << endl;
        } else if (next) {
            next->handleRequest(level);
        }
    }
};

// Concrete Handler 2
class ConcreteHandler2 : public AbstractHandler {
public:
    void handleRequest(int level) override {
        if (level == 2) {
            cout << "Handler2 processed request of level 2" << endl;
        } else if (next) {
            next->handleRequest(level);
        }
    }
};

// Concrete Handler 3
class ConcreteHandler3 : public AbstractHandler {
public:
    void handleRequest(int level) override {
        if (level == 3) {
            cout << "Handler3 processed request of level 3" << endl;
        } else if (next) {
            next->handleRequest(level);
        } else {
            cout << "No handler available for request of level " << level << endl;
        }
    }
};

// Demo
int main() {
    ConcreteHandler1 h1;
    ConcreteHandler2 h2;
    ConcreteHandler3 h3;

    // Build chain h1 -> h2 -> h3
    h1.setNext(&h2)->setNext(&h3);

    cout << "Sending request level 1:" << endl;
    h1.handleRequest(1);

    cout << "Sending request level 2:" << endl;
    h1.handleRequest(2);

    cout << "Sending request level 3:" << endl;
    h1.handleRequest(3);

    cout << "Sending request level 4:" << endl;
```

```
    h1.handleRequest(4);
    return 0;
}
```

Result Discussion

Sample Output:

```
Sending request level 1:
Handler1 processed request of level 1

Sending request level 2:
Handler2 processed request of level 2

Sending request level 3:
Handler3 processed request of level 3

Sending request level 4:
No handler available for request of level 4
```

Discussion:

1. Requests are passed along the chain until a suitable handler processes them.
2. Handler1 processed level 1, Handler2 processed level 2, Handler3 processed level 3.
3. If no handler exists for the request (level 4), a message is shown.

Conclusion:

The Chain of Responsibility Design Pattern provides a way to decouple senders and receivers of requests by allowing multiple handlers to process them dynamically. In this lab:

- We designed a handler chain with three concrete handlers.
- We demonstrated request processing at different levels.
- We showed how unhandled requests are propagated until the end of the chain.

This pattern increases flexibility and scalability by letting you add or remove handlers without changing client code.