Chain of Responsibility Design Pattern

Last Updated : 03 Jan, 2025

The Chain of Responsibility design pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass it along the chain to the next handler.

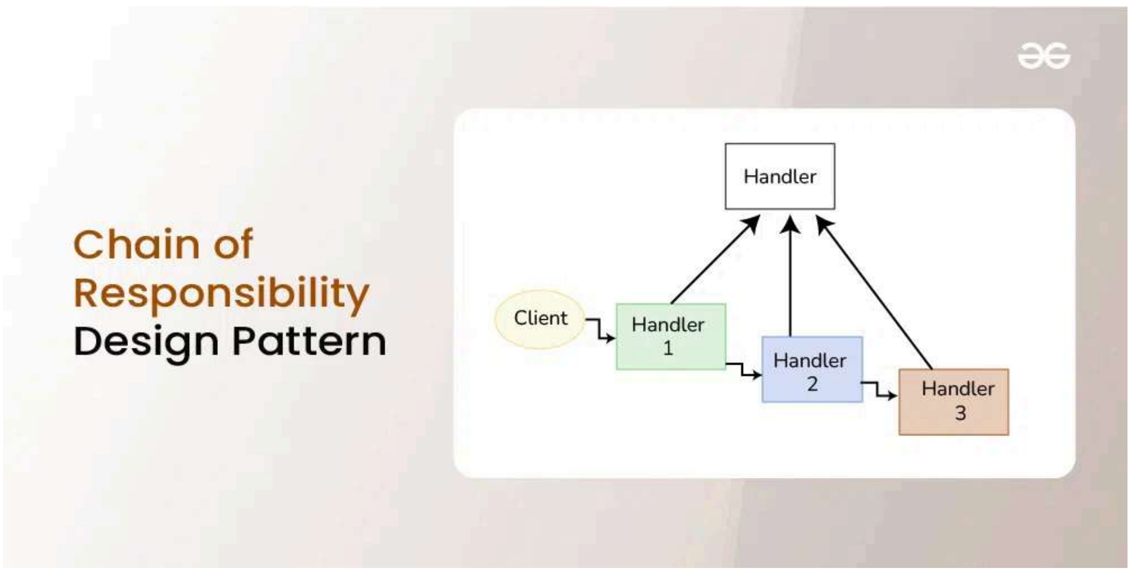


Table of Content

What is the Chain of Responsibility Design Pattern?

Chain of Responsibility Pattern or Chain of Responsibility Method is a Behavioral Design Pattern, which allows an object to send a request to other objects without knowing who is going to handle it.

- This pattern is frequently used in the chain of multiple objects, where each object either handles the request or passes it on to the next object in the chain if it is unable to handle that request.
- This pattern encourages loose coupling between sender and receiver, providing freedom in handling the request.

Characteristics of the Chain of Responsibility Design Pattern

Below are the main characteristics of chain of responsibility design pattern:

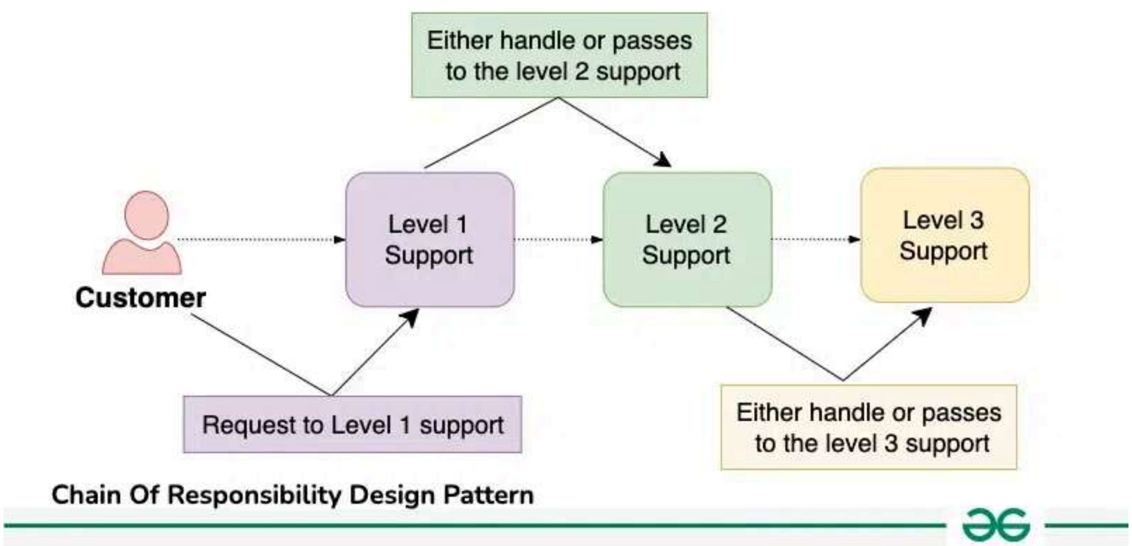
- **Loose Coupling**: This means the sender of a request doesn't need to know which specific object will handle it. Similarly, the handler doesn't need to understand how the requests are sent. This keeps the components separate and flexible.
- **Dynamic Chain**: While the program is running, changing the chain is simple. This makes your code incredibly flexible because you may add or delete handlers without changing the main body of the code.
- **Single Responsibility Principle**: Each handler in the chain has one job: either to handle the request or to pass it to the next handler. This keeps the code organized and focused, making it easier to manage.
- **Sequential Order**: Requests move through the chain one at a time. Each handler gets a chance to process the request in a specific order, ensuring consistency.
- **Fallback Mechanism**: If a request isn't handled by any of the handlers, the chain can include a fallback option. This means there's a default way to deal with requests that don't fit anywhere else.

Real-World Analogy of the Chain Of Responsibility Design Pattern

> *Imagine a customer service department with multiple levels of support staff, each responsible for handling different types of customer inquiries*

based on their complexity. The chain of responsibility can be illustrated as follows:

- **Level 1 Support**: This represents the first point of contact for customer inquiries. Level 1 support staff handle basic inquiries and provide general assistance. If they cannot resolve the issue, they escalate it to Level 2 support.
- **Level 2 Support**: This level consists of more experienced support staff who can handle more complex issues that Level 1 support cannot resolve. If Level 2 support cannot resolve the issue, they escalate it to Level 3 support.
- **Level 3 Support**: This is the highest level of support, consisting of senior or specialized staff who can handle critical or highly technical issues. If Level 3 support cannot resolve the issue, they may involve other departments or experts within the organization.



Components of the Chain of Responsibility Design Pattern

The Chain of Responsibility Pattern consists of the following key components:

- **Handler Interface or Abstract Class**: This is the base class that defines the interface for handling requests and, in many cases, for chaining to the next handler in the sequence.
- **Concrete Handlers:** These are the classes that implement how the requests are going to be handled. They can handle the request or pass it to the next handler in the chain if it is unable to handle that request.

- **Client**: The request is sent by the client, who then forwards it to the chain's first handler. Which handler will finally handle the request is unknown to the client.

How to Implement Chain of Responsibiility Design Pattern?

Below are the main steps for how to implement chain of responsibility design pattern:

- **Step 1: Define the Handler Interface**: Create an interface with methods for setting the next handler and processing requests.
- **Step 2: Create Concrete Handlers**: Implement the handler interface in multiple classes, each handling specific requests and passing unhandled requests to the next handler.
- **Step 3: Set Up the Chain**: Create instances of your handlers and link them together by setting the next handler for each one.
- **Step 4: Send Requests**: Use the first handler in the chain to send requests, allowing each handler to decide whether to process it or pass it along.

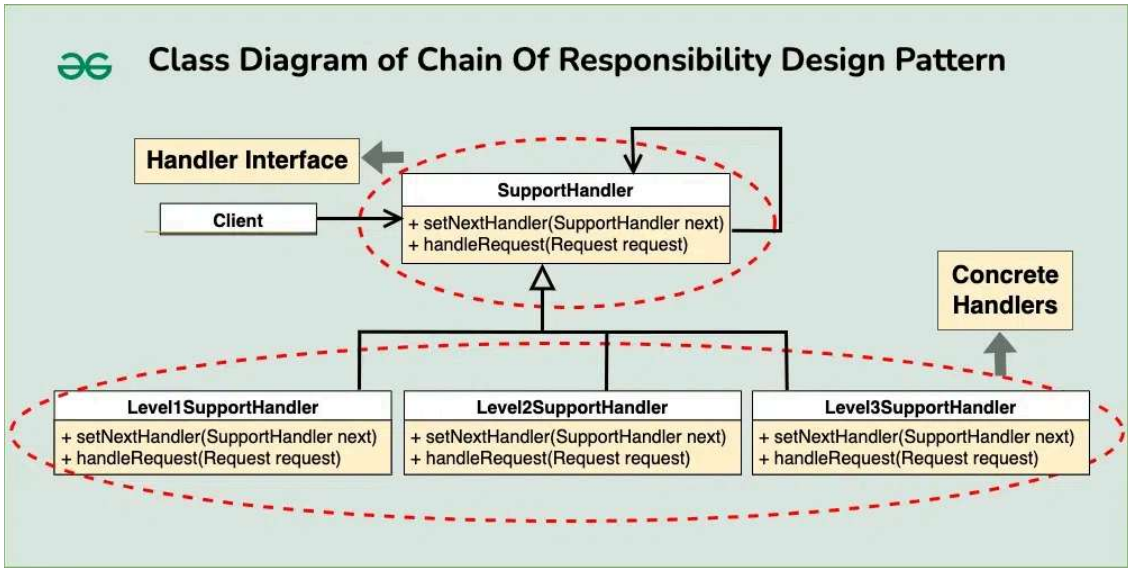
Chain of Responsibility Design Pattern Example

Let's understand this concept with the help of an example:

> *Imagine a customer support system where customer requests need to be handled based on their priority. There are three levels of support: Level 1, Level 2, and Level 3. Level 1 support handles basic requests, Level 2 support handles more complex requests, and Level 3 support handles critical issues that cannot be resolved by Level 1 or Level 2.*

Benefit of Using the Chain of Responsibility in this scenario:

The Chain of Responsibility pattern is beneficial in this situation because it allows us to create a chain of handlers, where each handler can either handle a request or pass it to the next handler in the chain. This way, we can easily add or remove handlers without modifying the client code, providing flexibility and scalability in handling customer requests.



Below is the code of above problem statement using Chain of Responsibility Design Pattern:

Let's break down into the component wise code:

1. Handler Interface

Defines the interface for handling requests. Includes methods for handling requests (**`handleRequest()`**) and setting the next handler in the chain (**`setNextHandler()`**).

```
public interface SupportHandler {
    void handleRequest(Request request);
    void setNextHandler(SupportHandler nextHandler);
}
```

2. Concrete Handlers

Implement the `SupportHandler` interface. Each handler is responsible for handling requests based on its assigned priority level. If a handler can handle the request, it processes it; otherwise, it passes the request to the next handler in the chain.

```java
public class Level1SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.BASIC) {
            System.out.println("Level 1 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

public class Level2SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.INTERMEDIATE) {
            System.out.println("Level 2 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

public class Level3SupportHandler implements SupportHandler {
    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.CRITICAL) {
            System.out.println("Level 3 Support handled the request.");
        } else {
            System.out.println("Request cannot be handled.");
        }
    }

```

```
    public void setNextHandler(SupportHandler nextHandler) {
        // No next handler for Level 3
    }
}
```

Complete code for the above example

Below is the complete code for the above example:

```
// Handler Interface
interface SupportHandler {
    void handleRequest(Request request);
    void setNextHandler(SupportHandler nextHandler);
}

// Concrete Handlers
class Level1SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.BASIC) {
            System.out.println("Level 1 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

class Level2SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

```

```
    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.INTERMEDIATE) {
            System.out.println("Level 2 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

class Level3SupportHandler implements SupportHandler {
    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.CRITICAL) {
            System.out.println("Level 3 Support handled the request.");
        } else {
            System.out.println("Request cannot be handled.");
        }
    }

    public void setNextHandler(SupportHandler nextHandler) {
        // No next handler for Level 3
    }
}

// Request Class
class Request {
    private Priority priority;

    public Request(Priority priority) {
        this.priority = priority;
    }

    public Priority getPriority() {
        return priority;
    }
}

// Priority Enum
enum Priority {
    BASIC, INTERMEDIATE, CRITICAL
}

// Main Class
```

```
public class Main {
    public static void main(String[] args) {
        SupportHandler level1Handler = new Level1SupportHandl
        SupportHandler level2Handler = new Level2SupportHandl
        SupportHandler level3Handler = new Level3SupportHandl

        level1Handler.setNextHandler(level2Handler);
        level2Handler.setNextHandler(level3Handler);

        Request request1 = new Request(Priority.BASIC);
        Request request2 = new Request(Priority.INTERMEDIATE)
        Request request3 = new Request(Priority.CRITICAL);

        level1Handler.handleRequest(request1);
        level1Handler.handleRequest(request2);
        level1Handler.handleRequest(request3);
    }
}
```

```
Level 1 Support handled the request.
Level 2 Support handled the request.
Level 3 Support handled the request.
```

Applications of Chain of Responsibility Design Pattern

Below are the applications of chain of responsibility design pattern:

- In graphical user interfaces (GUIs), events like mouse clicks or key presses can be handled by a chain of listeners. Each listener checks if it can handle the event, passing it along the chain if it can't. This way, multiple components can respond to the same event without being tightly linked.
- In logging systems, you might have different levels of loggers (like INFO, WARN, ERROR). Each logger can handle specific log messages. If one logger can't process a message (for example, if it's below its level), it passes it to the next logger in the chain.
- In security systems, access requests can be processed by a series of handlers that check permissions. For instance, one handler might check user

roles, while another checks specific permissions. If one handler denies access, it can pass the request to the next handler for further evaluation.

Pros of the Chain of Responsibility Design Pattern

Below are the pros of chain of responsibility design pattern:

- The pattern makes enables sending a request to a series of possible recipients without having to worry about which object will handle it in the end. This lessens the reliance between items.
- New handlers can be easily added or existing ones can be modified without affecting the client code. This promotes flexibility and extensibility within the system.
- The sequence and order of handling requests can be changed dynamically during runtime, which allows adjustment of the processing logic as per the requirements.
- It simplifies the interaction between the sender and receiver objects, as the sender does not need to know about the processing logic.

Cons of the Chain of Responsibility Design Pattern

Below are the cons of chain of responsibility design pattern:

- The chain should be implemented correctly otherwise there is a chance that some requests might not get handled at all, which leads to unexpected behavior in the application.
- The request will go through several handlers in the chain if it is lengthy and complicated, which could cause performance overhead. The processing logic of each handler has an effect on the system's overall performance.
- The fact that the chain has several handlers can make debugging more difficult. Tracking the progression of a request and determining which handler is in charge of handling it can be difficult.
- It may become more difficult to manage and maintain the chain of responsibility if the chain is dynamically modified at runtime.

Conclusion

In conclusion, the Chain of Responsibility pattern is a powerful tool for creating a flexible and extensible chain of handlers to process requests. It promotes loose coupling, making it a valuable addition to your design pattern toolbox when building applications. However, like any design pattern, it should be used judiciously, considering the specific requirements of your application.

Next Article

Command Design Pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented. Creational...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

---

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

---

3. Types of Behvioural Patterns

Chain of Responsibility Design Pattern

The Chain of Responsibility design pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass...

10 min read

Command Design Pattern

The Command Design Pattern is a behavioral design pattern that turns a request into a stand-alone object called a command. With the help of this pattern, you can capture each component of a request, including th...

10 min read

Interpreter Design Pattern

The Interpreter Design Pattern is a behavioral design pattern used to define a language's grammar and provide an interpreter to process statements in that language. It is useful for parsing and executing...

10 min read

Mediator Design Pattern

The Mediator Design Pattern simplifies communication between multiple objects in a system by centralizing their interactions through a mediator. Instead of objects interacting directly, they communicate via a mediato...

7 min read

Memento Design Pattern

The Memento Design Pattern is a behavioral pattern that helps save and restore an object's state without exposing its internal details. It is like a "snapshot" that allows you to roll back changes if something goes...

6 min read

Observer Design Pattern

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When one object (the subject) changes state, all its dependents (observers) are notified...

8 min read

State Design Pattern

The State design pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. It achieves this by encapsulating the object's behavior within different state...

11 min read

Strategy Design Pattern

The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to switch algorithms dynamically without altering the code structure. Tabl...

11 min read

Template Method Design Pattern

The Template Method Design Pattern is a behavioral design pattern that provides a blueprint for organizing code, making it flexible and easy to extend. With this pattern, you define the core steps of an algorithm in a...

8 min read

Visitor design pattern

An object-oriented programming method called the Visitor design pattern makes it possible to add new operations to preexisting classes without changing them. It improves the modularity and maintainability of...

7 min read

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages

Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:

K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

Company
About Us
Legal
Privacy Policy
In Media
Contact Us
Advertise with us
GFG Corporate Solution
Placement Training Program
GeeksforGeeks Community

Languages
Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial
Tutorials Archive

DSA
Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep Jain
All Cheat Sheets

Data Science & ML
Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP
Deep Learning

Web Technologies
HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
Bootstrap
Web Design

Python Tutorial
Python Programming Examples
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question
Django

Computer Science
Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development

DevOps
Git
Linux
AWS
Docker
Kubernetes
Azure
GCP