

"Heaven's Light is Our Guide"



Rajshahi University of Engineering & Technology
Department of Computer Science & Engineering

Lab Report On Design Patterns

Course No. : CSE 3206

Course Title : Software Engineering Sessional

Index :

- 1. *Proxy Design Pattern***
- 2. *Chain of Responsibilities***

<i>Submitted by -</i> <i>Group 05</i> <i>Consists of</i> <i>2003163,2003164,2003165</i>	<i>Submitted To -</i> <i>Farjana Parvin</i> <i>Lecturer</i> <i>Department of CSE, RUET</i>
--	---

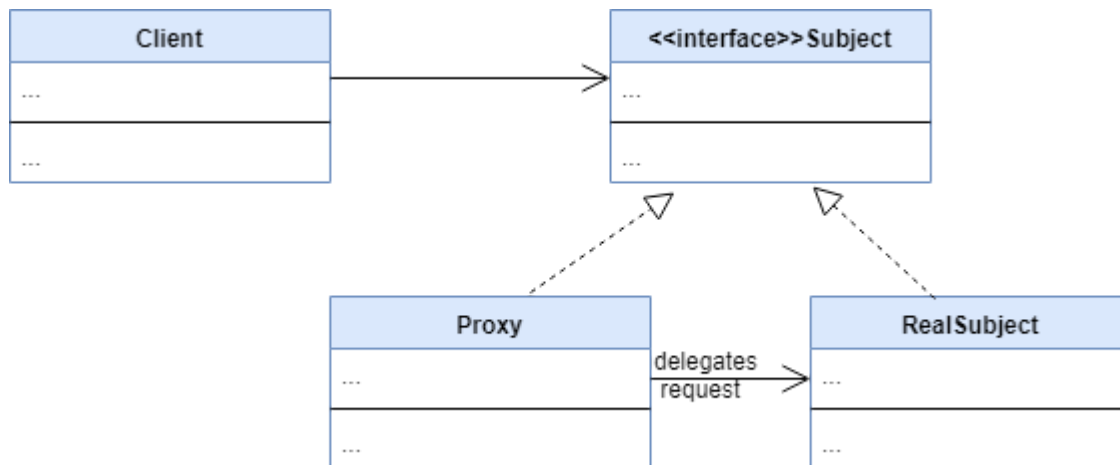
1 . Proxy Design Pattern

Title: Implementing the Proxy Design Pattern in an Internet Access System

Introduction:

The Proxy Design Pattern is used to control access to something by acting as a "middleman" between the user and the real object.

In this example, the proxy manages which websites users can access. Instead of allowing everyone to connect to any website, the proxy checks whether a website is allowed or blocked. This helps control access, improve security, and ensure users only visit approved websites.



Problem Statement

In an office network, unrestricted internet access can lead to misuse and productivity loss. A system is required to restrict access to certain websites while allowing access to others. The Proxy Design Pattern will help enforce these restrictions without modifying the underlying internet access system.

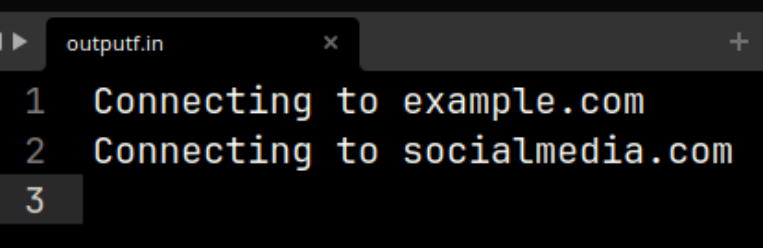
Code Without Proxy Design Pattern

```
// Real Subject: Provides unrestricted internet access
class InternetAccess {
    public void connectTo(String website) {
        System.out.println("Connecting to " + website);
    }
}

public class InternetWithoutProxy {
    public static void main(String[] args) {
        InternetAccess internet = new InternetAccess();

        // Unrestricted access
        internet.connectTo("example.com");
        internet.connectTo("socialmedia.com");
    }
}
```

Output:



```
outputf.in
1 Connecting to example.com
2 Connecting to socialmedia.com
3
```

Code With Proxy Design Pattern

```
import java.util.ArrayList;
import java.util.List;

// Subject Interface
interface Internet {
    void connectTo(String website);
}

// Real Subject: Provides actual internet access
class RealInternet implements Internet {
    @Override
    public void connectTo(String website) {
        System.out.println("Connecting to " + website);
    }
}
```

```

// Proxy: Controls access to the RealInternet
class ProxyInternet implements Internet {
    private RealInternet realInternet = new RealInternet();
    private static List<String> bannedWebsites;

    static {
        // Define a list of restricted websites
        bannedWebsites = new ArrayList<>();
        bannedWebsites.add("socialmedia.com");
        bannedWebsites.add("gaming.com");
    }

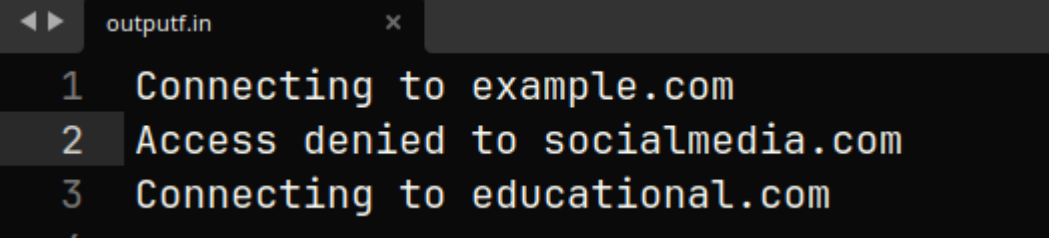
    @Override
    public void connectTo(String website) {
        if (bannedWebsites.contains(website)) {
            System.out.println("Access denied to " + website);
        } else {
            realInternet.connectTo(website);
        }
    }
}

// Client: Demonstrates the use of the Proxy
public class InternetWithProxy {
    public static void main(String[] args) {
        Internet internet = new ProxyInternet();

        internet.connectTo("example.com");
        internet.connectTo("socialmedia.com");
        internet.connectTo("educational.com");
    }
}

```

Output:



```

1 Connecting to example.com
2 Access denied to socialmedia.com
3 Connecting to educational.com

```

Conclusion

The Proxy Design Pattern is effectively used here to manage internet access. By using a proxy, the system can block restricted websites while still allowing access to approved ones. This improves security, ensures control over internet usage, and enhances productivity without changing the real internet access system.

2.Chain of Responsibilities

Introduction:

The Chain of Responsibility Pattern is a behavioral design pattern that creates a chain of receiver objects to process a request. It decouples the sender and receiver, ensuring that the request is handled dynamically based on its type and the capability of the handler. Each handler in the chain holds a reference to the next handler, allowing the request to traverse the chain until it is processed or reaches the end. This pattern promotes flexibility and simplifies system maintenance by enabling handlers to be added, removed, or modified without altering the client code.

Objective:

The objective of this lab is:

1. Recognize scenarios where the Chain of Responsibility Pattern is applicable.
2. Design and implement a chain of objects to handle different requests dynamically.
3. Explore how the pattern allows dynamic addition or modification of handlers.
4. Gain insights into the benefits of decoupling sender and receiver for improving code maintainability and scalability.

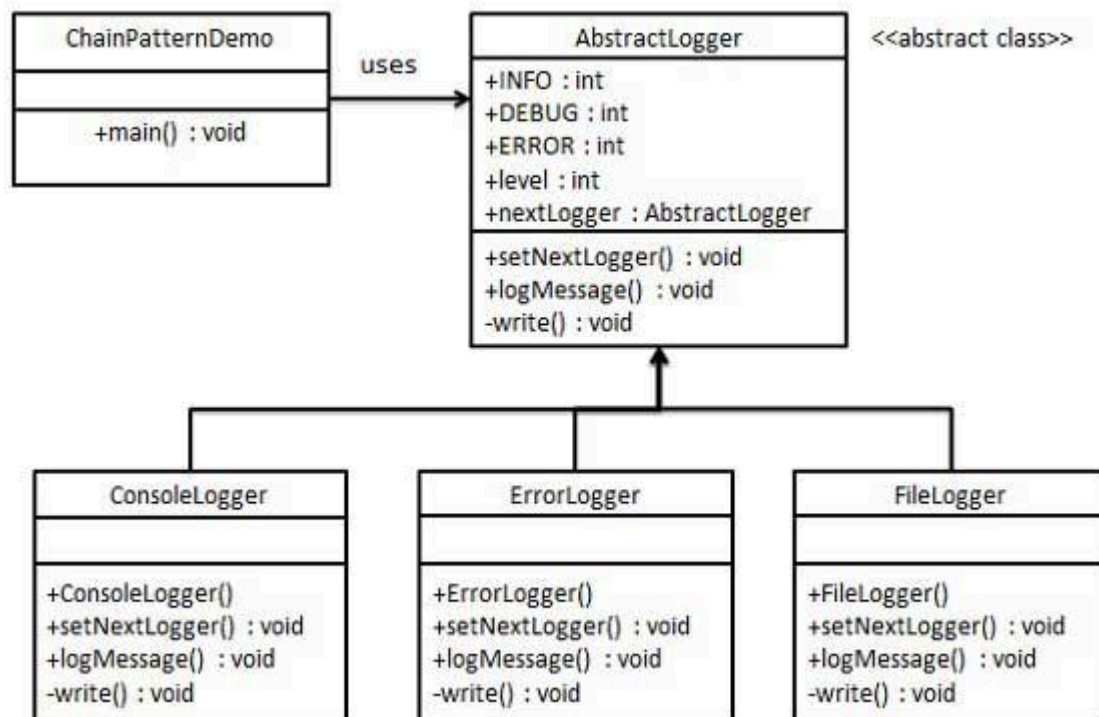
Problem Statement:

In a logging system, different levels of logs (e.g., INFO, DEBUG, ERROR) require distinct handling mechanisms, which may include console output, file storage, or database storage. Implementing all these loggers in a tightly coupled manner makes it difficult to add new logging mechanisms or modify existing ones. Develop a system where requests can be processed

dynamically by a chain of handlers, each responsible for a specific type of request, without tightly coupling the sender and the receivers.

Design and Implementation:

UML Diagram:



The UML diagram consists of:

1. **AbstractLogger** (Abstract class): Defines the interface and structure for handling requests.
2. **Concrete Loggers** (**ConsoleLogger**, **FileLogger**, **ErrorLogger**): Specific implementations for handling requests based on log levels.
3. **Client**: Configures the chain of responsibility and sends requests.

Code Implementation:

Step 1:create Abstract Logger Class:

```
1 public abstract class AbstractLogger {
2     public static int INFO = 1;
3     public static int DEBUG = 2;
4     public static int ERROR = 3;
5
6     protected int level;
7
8     //next element in chain or responsibility
9     protected AbstractLogger nextLogger;
10
11     public void setNextLogger(AbstractLogger nextLogger){
12         this.nextLogger = nextLogger;
13     }
14
15     public void logMessage(int level, String message){
16         if(this.level <= level){
17             write(message);
18         }
19         if(nextLogger != null){
20             nextLogger.logMessage(level, message);
21         }
22     }
23
24     abstract protected void write(String message);
25 }
26 }
```

Step 2: Concrete Logger Classes.

Implementations for specific logging levels.

```
1
2 public class ConsoleLogger extends AbstractLogger {
3
4     public ConsoleLogger(int level){
5         this.level = level;
6     }
7
8     @Override
9     protected void write(String message) {
10         System.out.println("Standard Console::Logger: " + message);
11     }
12 }
```

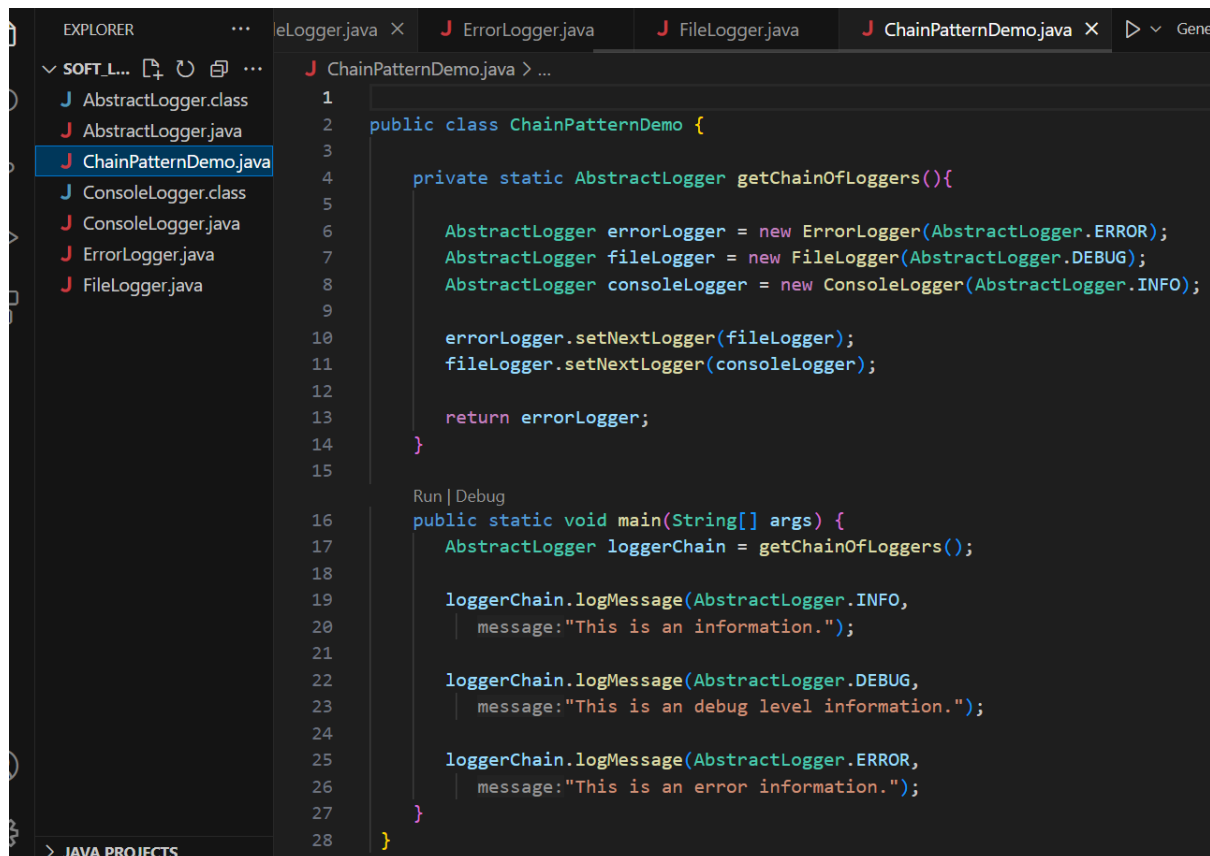
The screenshot shows an IDE with the Explorer on the left and the Editor on the right. The Explorer shows a project named 'SOFT_L...' with files: AbstractLogger.class, AbstractLogger.java, ChainPatternDemo.java, ConsoleLogger.class, ConsoleLogger.java, ErrorLogger.java (selected), and FileLogger.java. The Editor shows the code for ErrorLogger.java, which extends AbstractLogger. The code is as follows:

```
1
2 public class ErrorLogger extends AbstractLogger {
3
4     public ErrorLogger(int level){
5         this.level = level;
6     }
7
8     @Override
9     protected void write(String message) {
10         System.out.println("Error Console::Logger: " + message);
11     }
12 }
```

The screenshot shows the same IDE with the Explorer on the left and the Editor on the right. The Explorer shows the same project with files: AbstractLogger.class, AbstractLogger.java, ChainPatternDemo.java, ConsoleLogger.class, ConsoleLogger.java, ErrorLogger.java, and FileLogger.java (selected). The Editor shows the code for FileLogger.java, which extends AbstractLogger. The code is as follows:

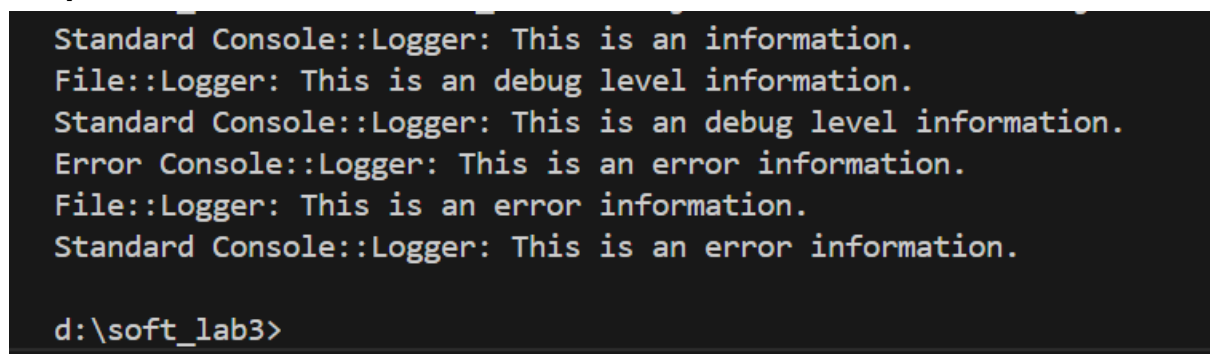
```
1 public class FileLogger extends AbstractLogger {
2
3     public FileLogger(int level){
4         this.level = level;
5     }
6
7     @Override
8     protected void write(String message) {
9         System.out.println("File::Logger: " + message);
10     }
11 }
12
```

Step 3: Configure the Chain



```
1 public class ChainPatternDemo {
2
3     private static AbstractLogger getChainOfLoggers(){
4
5         AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
6         AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
7         AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
8
9         errorLogger.setNextLogger(fileLogger);
10        fileLogger.setNextLogger(consoleLogger);
11
12        return errorLogger;
13    }
14
15    Run | Debug
16    public static void main(String[] args) {
17        AbstractLogger loggerChain = getChainOfLoggers();
18
19        loggerChain.logMessage(AbstractLogger.INFO,
20            message:"This is an information.");
21
22        loggerChain.logMessage(AbstractLogger.DEBUG,
23            message:"This is an debug level information.");
24
25        loggerChain.logMessage(AbstractLogger.ERROR,
26            message:"This is an error information.");
27    }
28 }
```

Output:



```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.

d:\soft_lab3>
```

Conclusion: The Chain of Responsibility Pattern provides an effective solution for handling requests in a flexible and maintainable manner. Through this lab, the implementation of a logging system using this pattern demonstrated how decoupling the sender and receiver allows dynamic processing of requests while maintaining system scalability. This approach is valuable in designing systems requiring hierarchical request handling or dynamic behavior changes.

