Assignment # 1

Design Patterns

Faizan (FA21-BSE-011)

Fawad Iqbal (FA21-BSE-012)

28-Oct-2024

Contents

Question No. 01

Singleton Design Pattern

Ensures a class has only one instance and provides a global access point to it. It is used to manage shared resources, like a configuration or logging instance in an application.

Class Diagram

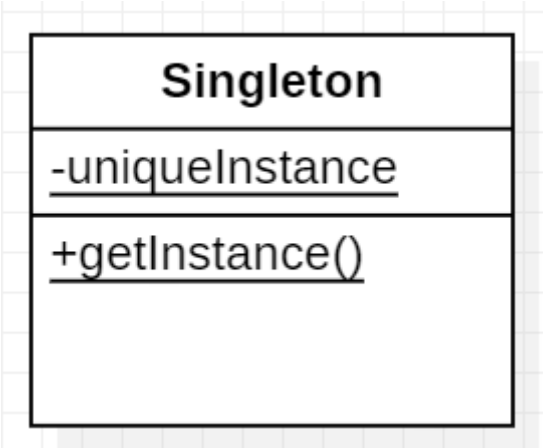

Figure 1: Class Diagram for Singleton Design Pattern

Implementation

```
public class Singleton {
    // variable to hold single instance
    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}
```

Observer Design Pattern

Established a one-to-many dependency between objects. When the state of the subject changes, all the observers (subscribers) are notified and updated automatically.

Class Diagram

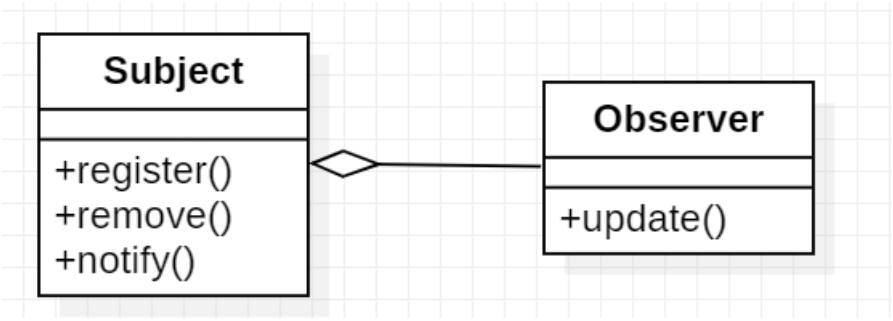

Figure 2: Class Diagram for Observer Design Pattern

Implementation

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

// Subject class
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }

    // Register and remove observers
    public void attach(Observer observer) {
```

```
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

    // Notify all observers of state change
    private void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

// Observer interface
interface Observer {
    void update();
}

// Concrete Observer classes
class ConcreteObserverA implements Observer {
    private Subject subject;

    public ConcreteObserverA(Subject subject) {
        this.subject = subject;
        subject.attach(this);
    }

    @Override
```

```
    public void update() {
        System.out.println("ConcreteObserverA: State updated to
" + subject.getState());
    }
}

class ConcreteObserverB implements Observer {
    private Subject subject;

    public ConcreteObserverB(Subject subject) {
        this.subject = subject;
        subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("ConcreteObserverB: State updated to
" + subject.getState());
    }
}

public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();

        // Create observers and attach to subject
        new ConcreteObserverA(subject);
        new ConcreteObserverB(subject);

        // Change state
        System.out.println("Setting state to 10.");
```

```
        subject.setState(10);

        System.out.println("Setting state to 20.");
        subject.setState(20);
    }
}
```

Question No. 03

1. Changing the algorithm that an object uses:
   - **Design Pattern:** Strategy Pattern
   - **Explanation:** The Strategy Pattern allows you to define a family of algorithms and make them interchangeable. The context class (the object) maintains a reference to a strategy (algorithm), and you can switch out this strategy at runtime to change the algorithm used by the object.
2. Changing the class of the object that a method returns:
   - **Design Pattern:** Factory Method Pattern
   - **Explanation:** The Factory Method Pattern defines an interface for creating an object but allows subclasses to alter the type of object that will be created. This pattern is useful when you want to allow a method to return objects of different classes, possibly based on some configuration or runtime condition.
3. Changing the kind and number of objects that react to changes in the object you are designing:
   - **Design Pattern:** Observer Pattern
   - **Explanation:** The Observer Pattern allows an object (subject) to maintain a list of dependents (observers) and notify them of any changes, without knowing who they are or how many exist. This pattern is ideal when multiple objects need to react to changes in another object dynamically.
4. Adding operations to classes without changing the class:
   - **Design Pattern:** Visitor Pattern
   - **Explanation:** The Visitor Pattern allows you to define new operations on elements of an object structure without modifying their classes. You create a visitor class that implements operations on elements of different types, which allows you to add new operations independently of the classes on which they operate.
5. Changing how methods in a class behave:
   - **Design Pattern:** Decorator Pattern
   - **Explanation:** The Decorator Pattern allows you to attach additional responsibilities or behaviours to an object dynamically. By wrapping the object with decorator classes, you can extend or modify its behaviour at runtime without modifying its actual class. This pattern is commonly used to change or enhance how methods in a class behave.

Question No. 04

To solve the problem of multiple loggers potentially causing issues like race conditions and file corruption in a large-scale application, the **Singleton Pattern** is an ideal choice.

Solution Using Singleton Pattern

The Singleton Pattern ensures that only one instance of the logger class is created throughout the application. By creating a single, shared instance of the logger, all classes can use the same logging object. This approach avoids issues like race conditions, file corruption, and inefficient log management because:

1. **Single Access Point:** All logging operations are centralized through one instance, which prevents multiple loggers from writing to the same log file at the same time.
2. **Resource Management:** A single instance uses less memory and avoids unnecessary duplication of loggers.
3. **Thread Safety:** The Singleton can be designed to be thread-safe, ensuring that log entries are written in a consistent, synchronized manner.

Why Singleton Pattern is Suitable

- **Control over Log File Access:** The Singleton pattern provides a single access point to the log file, reducing the chance of conflicts.
- **Consistency:** Every class in the system uses the same instance of the logger, ensuring consistent logging format and location.
- **Efficiency:** By reusing the same instance, the Singleton pattern reduces the overhead of creating multiple loggers.

Java Implementation

```
import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;


public class Logger {

    // Volatile instance variable to ensure thread-safe double-checked locking

    private static volatile Logger instance;

    private PrintWriter writer;


    // Private constructor to prevent instantiation from other classes
```

```
    private Logger() {
        try {
            writer = new PrintWriter(new FileWriter("application.log", true));
        } catch (IOException e) {
            System.out.println("Error initializing logger: " + e.getMessage());
        }
    }

    // Public method to provide access to the singleton instance
    public static Logger getInstance() {
        if (instance == null) { // First check (without locking)
            synchronized (Logger.class) {
                if (instance == null) { // Second check (with locking)
                    instance = new Logger();
                }
            }
        }
        return instance;
    }

    // Method to log messages to the log file
    public void log(String message) {
        synchronized (this) {
            writer.println(message);
            writer.flush();
        }
    }
```

```
    // Method to close the writer when the application is shutting
down
    public void close() {
        writer.close();
    }
}


public class ApplicationComponent {
    public void performAction() {
        Logger logger = Logger.getInstance();
        logger.log("Action performed in ApplicationComponent.");
    }
}
```

Question No. 05

To address this requirement, the **Observer Pattern** is the most suitable design pattern. The Observer Pattern enables a "one-to-many" dependency, where one object (the subject) notifies multiple dependent objects (observers) of any changes, without needing to know details about the observers.

Why Observer Pattern is Suitable for This Scenario

- **Real-time Notifications:** When a user posts an update, the Observer Pattern allows all followers (observers) to be notified automatically.
- **Decoupling:** The Observer Pattern decouples the user who posts content (subject) from their followers (observers). The subject doesn't need to know who its followers are; it just broadcasts the updates to all registered observers.
- **Scalability:** This pattern easily supports adding new followers or removing existing ones, making it flexible and scalable for a large social media platform.

Components of the Observer Pattern in This Scenario

1. **Subject (Observable):** The user, page, or group that can be followed. When the subject posts new content, it notifies all registered observers.
2. **Observers:** Followers of the subject (users who follow the subject). Observers get updated with new content or posts.
3. **Notification Mechanism:** A method by which the subject notifies all observers when there’s a new post.

Java Implementation

1. Define the Observer Interface

```
interface Observer {
    void update(String post);
}
```

2. Define the Subject Interface

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

interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String post);
}
```

3. Implement the Subject as a User

```
class User implements Subject {
    private List<Observer> followers;
    private String name;

    public User(String name) {
        this.name = name;
        this.followers = new ArrayList<>();
    }

    @Override
    public void addObserver(Observer observer) {
        followers.add(observer);
    }

    @Override
```

```
    public void removeObserver(Observer observer) {
        followers.remove(observer);
    }

    @Override
    public void notifyObservers(String post) {
        for (Observer follower : followers) {
            follower.update(name + " posted: " + post);
        }
    }

    // Method to post content
    public void postContent(String post) {
        System.out.println(name + " posted a new update: " +
post);
        notifyObservers(post);
    }
}
```

4. Implement the Observer as a Follower

```
class Follower implements Observer {
    private String followerName;

    public Follower(String followerName) {
        this.followerName = followerName;
    }

    @Override
    public void update(String post) {
        System.out.println(followerName + " received update: " +
post);
    }
```

```
}
```

5. Demonstration of Usage

```
public class SocialMediaPlatform {
    public static void main(String[] args) {
        // Create a user (subject)
        User user1 = new User("Alice");

        // Create followers (observers)
        Follower follower1 = new Follower("Bob");
        Follower follower2 = new Follower("Charlie");

        // Followers start following Alice
        user1.addObserver(follower1);
        user1.addObserver(follower2);

        // Alice posts content
        user1.postContent("Check out my new blog post!");

        // Removing a follower and posting again
        user1.removeObserver(follower1);
        user1.postContent("New travel photos uploaded!");
    }
}
```