Name: Muhammad Rehan

Reg No: 4404-FOC/BSSE/F22-A

Subject: SCD

Submitted To: SIR Shakir Rashid

Assignment no= 2

Design a Multi-Channel Notification System

Objective

Design and implement a Notification System that can send messages through various channels (Email, SMS, Push, and In-App). The system must be:

- **Extensible (Open-Closed Principle):** Easily allow new notification types without altering existing code.
- **Focused (Single Responsibility Principle):** Each class has a single responsibility.
- **Managed (Singleton Pattern):** Ensure that there is only one instance of the Notification Manager.
- **Flexible (Factory Method Pattern):** Delegate the creation of notification objects to a factory.

Scenario Overview

Imagine you are tasked with creating an application that notifies users via their preferred communication channels. Depending on user preferences, the application can send:

- An **Email**
- An **SMS**
- A **Push Notification**
- An **In-App Notification** (as a new addition)

The system must be designed so that adding a new type of notification (for example, a chat notification) does not require modifying the existing classes.

Requirements

1. **Open-Closed Principle:**
   - The system should be open for extension but closed for modification.
   - New notification types must be added by creating new classes rather than changing existing ones.
2. **Single Responsibility Principle:**
   - Each notification class is responsible only for the logic of sending its specific type of message.
   - The Notification Factory is solely responsible for instantiating notification objects.

- The Notification Manager is solely responsible for coordinating the sending process.

3. **Singleton Pattern:**
   - The Notification Manager must be implemented as a Singleton to ensure a single point of control for sending notifications.
4. **Factory Method Pattern:**
   - The Notification Factory will decide which type of notification object to create based on the input (e.g., a string identifier).

Java Code Implementation

Below is an example implementation in Java:

```
// Notification.java
interface Notification {
    void send(String message);
}

// EmailNotification.java
class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

// SMSNotification.java
class SMSNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

// PushNotification.java
class PushNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending Push Notification: " + message);
    }
}

// InAppNotification.java (New Notification Type)
class InAppNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending In-App Notification: " + message);
    }
}
```

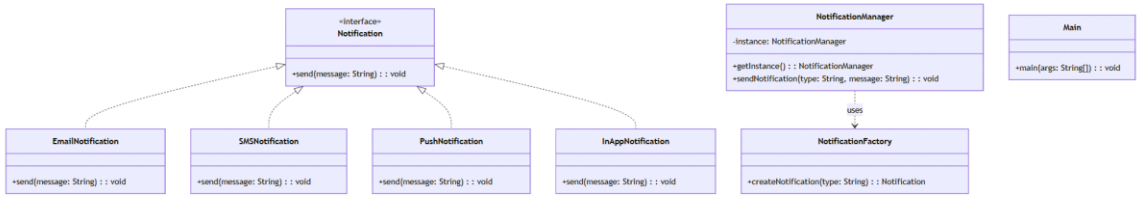
```
// NotificationFactory.java
class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type == null || type.isEmpty()) {
            return null;
        }
        switch (type.toUpperCase()) {
            case "EMAIL":
                return new EmailNotification();
            case "SMS":
                return new SMSNotification();
            case "PUSH":
                return new PushNotification();
            case "INAPP":
                return new InAppNotification();
            default:
                return null;
        }
    }
}

// NotificationManager.java (Singleton)
class NotificationManager {
    private static NotificationManager instance;

    // Private constructor to prevent instantiation
    private NotificationManager() {}

    public static NotificationManager getInstance() {
        if (instance == null) {
            instance = new NotificationManager();
        }
        return instance;
    }

    // Use factory to create a notification and send it
    public void sendNotification(String type, String message) {
        Notification notification =
NotificationFactory.createNotification(type);
        if (notification != null) {
            notification.send(message);
        } else {
            System.out.println("Invalid Notification Type: " + type);
        }
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        NotificationManager manager = NotificationManager.getInstance();

        // Sending different types of notifications
        manager.sendNotification("EMAIL", "Hello via Email!");
        manager.sendNotification("SMS", "Hello via SMS!");
```

```
        manager.sendNotification("PUSH", "Hello via Push Notification!");
        manager.sendNotification("INAPP", "Hello via In-App Notification!");
    }
}
```

Code Explanation

- **Notification Interface & Concrete Classes:**
  Each concrete notification (Email, SMS, Push, In-App) implements the `Notification` interface and defines its own `send()` method.
- **NotificationFactory:**
  The factory method `createNotification(String type)` returns an instance of the correct notification class based on the provided type. This allows the system to be extended with new notification types easily.
- **NotificationManager (Singleton):**
  The `NotificationManager` ensures that only one instance exists across the application. It handles the sending of notifications by delegating object creation to the factory and calling the appropriate send method.
- **Main Class:**
  Demonstrates the usage of the NotificationManager to send notifications through different channels.

UML Class Diagram



Summary

- **Open-Closed Principle:** The design allows new notification types to be added without modifying existing classes.
- **Single Responsibility Principle:** Each class has a clear, single purpose.
- **Singleton Pattern:** Ensures that only one instance of NotificationManager exists.
- **Factory Method Pattern:** The NotificationFactory creates instances of the correct notification class based on input, encapsulating the creation logic.