Week 6 Lab: SOLID Principles

LCSCI5202: Object Oriented Design Week 5

Dr. Waleed Iqbal

Lab Statement

- You are provided with a non-SOLID C# program that handles multiple notification types (Email, SMS, WhatsApp) within a single class.
- To refactor a poorly designed NotificationService class and apply each of the SOLID principles (SRP, OCP, LSP, ISP, DIP) to make the code cleaner, modular, and easier to maintain.

Code to be Refactored

```
public class NotificationService
{
    public void SendNotification(string message, string type) {
            if (type == "Email")
                    Console.WriteLine("Sending Email: " + message);
            else
                    if (type == "SMS")
                            Console.WriteLine("Sending SMS: " + message);
                    else
                            if (type == "WhatsApp")
                                    Console.WriteLine("Sending WhatsApp: " + message);
                            else
                                    Console.WriteLine("Unknown notification type");
            }
}
```

Code to be Refactored

```
class Program
{
static void Main()
{
    NotificationService service = new NotificationService();
    service.SendNotification("Welcome to SOLID Lab!", "Email");
    service.SendNotification("System update available!", "WhatsApp");
}
}
```

Task 1

- *The current NotificationService class sends all types of notifications.*
- *Create separate classes for each type: EmailService, SMSService, WhatsAppService.*
- *Each class should have its own Send(string message) method.*
- *Update NotificationService so it only coordinates which service to call.*
- *Keep NotificationService responsible only for managing message delivery, not for sending logic.*

Task 2

- Create an interface *INotificationChannel* with *void Send(string message)*.
- Make *EmailService, SMSService*, and *WhatsAppService* implement this interface.
- Modify *NotificationService* to work with *INotificationChannel* instead of concrete classes.
- Add a new notification type (e.g., *SlackService*) without editing *NotificationService*.
- Verify that new notification types work through the same interface.

Task 3

- All notification classes now implement *INotificationChannel*.
- Create a new class *TestChannel* that also implements *INotificationChannel*.
- Replace one of the existing services (e.g., *EmailService*) with *TestChannel* in *Main()*.
- The system should still run correctly, proving substitutability.
- Ensure no implementation breaks expected behavior when substituted.

Task 4

- Create smaller, focused interfaces for specific responsibilities.
- Define two interfaces:
  - *INotificationChannel* should contains *Send(string message)*
  - *ISchedulable* should contains *Schedule(string message, DateTime time)*
- Only classes that support scheduling (e.g., *EmailService*) implement *ISchedulable*.
- Classes that don’t support scheduling (e.g., *SMSService*) implement only *INotificationChannel*.
- Verify that each class only implements methods relevant to its function.

Task 5

- Modify NotificationService to depend on the interface INotificationChannel.
- Use constructor injection to pass the dependency:

```
public NotificationService(INotificationChannel channel) {
        _channel = channel;
}
```

- Remove all new keywords that directly create channel objects inside *NotificationService*.
- In Main(), create the channel you want (e.g., new *EmailService()*) and pass it into *NotificationService*.
- Test swapping between *EmailService, SMSService*, and *SlackService* without changing *NotificationService*.
- Verify that your design now supports flexible and testable dependency injection.