C# Project with SOLID Principles

Project Report

SWE 4302 - Object Oriented Concepts II Lab

Name: Minhajul Abedin Bhuiyan
ID: 210042148
Program: BSc in Software Engineering
Department: Computer Science and Engineering
Submission Date: January 19, 2024

Islamic University of Technology
Board Bazar, Gazipur, Dhaka, 1704, Bangladesh

SOLID Principle

The SOLID principles are a set of five design principles in object-oriented programming intended to make software designs more understandable, flexible, and maintainable. The acronym SOLID stands for:

1. Single Responsibility Principle (SRP):

A class should have only one reason to change, meaning it should have only one responsibility.

The Single Responsibility Principle (SRP) advocates that a class should focus on doing one thing and doing it well. This ensures that if there is a need for modification in the future, it is triggered by only one aspect of the system.

Example in C#:

In the SRP violated example, the `Employee` class handles both salary calculation and report generation. In the Refactored example, these responsibilities are separated into two classes, `Employee` and `ReportGenerator`.

```
// SRP Violated example
class Employee
{
    public void CalculateSalary() { /*...*/ }
    public void GenerateReport() { /*...*/ }
}

// Refactored example
class Employee
{
    public void CalculateSalary() { /*...*/ }
}

class ReportGenerator
{
    public void GenerateReport() { /*...*/ }
}
```

2. Open/Closed Principle (OCP):

Software entities should be open for extension but closed for modification.

The Open/Closed Principle encourages designing classes and modules in a way that allows them to be easily extended without altering their existing code. This promotes code stability and reduces the risk of introducing bugs when making changes.

Example in C#:

In the bad example, the `AreaCalculator` class would need modification when adding a new shape. The good example uses an interface (`IShape`) to allow for easy extension without modifying existing code.

```
// Bad example
class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

// This violates OCP when adding a new shape
class AreaCalculator
{
    public double CalculateArea(Rectangle rectangle) { /*...*/ }
}

// Good example
interface IShape
{
    double CalculateArea();
}

class Rectangle : IShape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public double CalculateArea() { /*...*/ }
}
```

3. Liskov Substitution Principle (LSP):

Subtypes should be substitutable for their base types without altering the correctness of the program.

The Liskov Substitution Principle ensures that objects of a base class can be replaced with objects of a derived class without affecting the correctness of the program. It maintains the expected behavior when substituting objects.

Example in C#:

In the bad example, the `Penguin` class overrides the `Fly` method, violating the expectation of the base class. In the good example, the `Penguin` class implements the `IFlyable` interface without altering the behavior.

```
// Violated example
class Bird
{
    public void Fly() { /*...*/ }
}

class Penguin : Bird
```

```
{
    // Penguins can't fly, violates LSP
    public new void Fly() { /*...*/ }
}

// Refactor example
interface IFlyable
{
    void Fly();
}

class Bird : IFlyable
{
    public void Fly() { /*...*/ }
}
class Penguin : IFlyable
{
    public void Fly()
    {
        // Penguins can't fly, handle accordingly
    }
}
```

4. Interface Segregation Principle (ISP):

A class should not be forced to implement interfaces it does not use.

The Interface Segregation Principle suggests that a class should not be burdened with implementing interfaces that contain methods it doesn't need. This prevents classes from being forced to provide unnecessary implementations.

Example in C#:

In the violated example, the `Manager` class is forced to implement both `Work` and `Eat` methods. In the corrected example, the `IWorker` and `IEater` interfaces are separate, allowing classes to implement only what they need.

```
// Violated example
interface IWorker
{
    void Work();
    void Eat();
}

class Manager : IWorker
{
    public void Work() { /*...*/ }
    public void Eat() { /*...*/ }
```

```
}

// Refactored example
interface IWorker
{
    void Work();
}

interface IEater
{
    void Eat();
}

class Manager : IWorker, IEater
{
    public void Work() { /*...*/ }
    public void Eat() { /*...*/ }
}
```

5. Dependency Inversion Principle (DIP):

High-level modules should not depend on low-level modules. Both should depend on abstractions, and abstractions should not depend on details.

The Dependency Inversion Principle encourages designing systems where high-level modules (e.g., business logic) are not directly dependent on low-level modules (e.g., database access). Instead, both should depend on abstractions, allowing for flexibility and ease of maintenance.

Example in C#:

In the violated example, the `Switch` class has a direct dependency on the concrete `LightBulb` class. In the refactored example, an interface (`IDevice`) is introduced to abstract the dependency.

```
// Violated example
class LightBulb
{
    public void TurnOn() { /*...*/ }
    public void TurnOff() { /*...*/ }
}

class Switch
{
    private LightBulb bulb;

    public Switch()
    {
        bulb = new LightBulb(); // Direct dependency on a concrete class
    }
```

```
    public void Operate()
    {
        // Operate the light bulb
        bulb.TurnOn();
        bulb.TurnOff();
    }
}

// Refactored example
interface IDevice
{
    void TurnOn();
    void TurnOff();
}

class LightBulb : IDevice
{
    public void TurnOn() { /*...*/ }
    public void TurnOff() { /*...*/ }
}

class Switch
{
    private IDevice device;

    public Switch(IDevice device)
    {
        this.device = device; // Dependency is now on an abstraction
    }

    public void Operate()
    {
        // Operate the device
        device.TurnOn();
        device.TurnOff();
    }
}
```

Code Smell

Code smells are certain patterns or structures in code that might indicate potential issues or areas for improvement. They are not bugs or errors in themselves, but they can hint at design problems or areas where the code might be harder to maintain, understand, or extend. Here are some common code smells:

1. Long Method: A method that is excessively long, making it harder to understand and maintain.

2. Large Class: Classes with too many responsibilities, leading to decreased maintainability.

3. Code Duplication (Duplicated Code): Repeating identical or very similar code in multiple places, potentially causing maintenance challenges.

4. Comments (Excessive Comments): Overuse of comments may indicate unclear or complex code that needs better self-explanatory design.

5. Feature Envy: Methods in one class using more methods or properties of another class than its own, suggesting a potential design issue.

6. Dead Code: Code that is never executed, adding unnecessary clutter to the codebase.

7. Inconsistent Naming: Lack of consistent naming conventions for variables, methods, or classes, reducing code readability.

8. Tight Coupling: Classes that are tightly coupled to each other, making the code harder to modify or extend.

9. Shotgun Surgery: Making the same change in multiple places, indicating a lack of encapsulation and proper design.

10. Missing Abstraction: Absence of higher-level concepts or abstractions, leading to code dealing only with low-level details.

Design Smell:

1. Rigidity: Changes in one-part lead to widespread, costly modifications.

2. Fragility: Modifying one area risks breaking unrelated parts.

3. Immobility: Difficulty in reusing code leads to duplication and hinders good practices.

4. Viscosity: Not-good solutions become convenient, promoting bad practices.

5. Needless Complexity: Over-design introduces unnecessary complexity, making code harder to maintain.

6. Needless Repetition: Repeated code reduces maintainability and increases the risk of bugs.

7. Opacity: Hard-to-understand code hampers maintainability and can lead to errors.

Creational Design Patterns

Singleton Pattern: The Singleton pattern restricts class instantiation to a single instance. It is useful for scenarios where only one object is needed to coordinate actions across the system, such as in logging. The pattern ensures a single instance exists and provides global access to it, often through a static method.

Factory Method Pattern: The Factory Method pattern provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects created. It solves the problem of creating product objects without specifying their concrete classes. The pattern involves a factory method in an interface, implemented by concrete classes, allowing loose coupling and flexibility in object creation.

Project: Hospital Management System

The "Hospital Management System" project is a simple yet effective software solution designed to streamline hospital operations. It focuses on managing patient and doctor information, appointment scheduling, and medical records. The goal is to provide a user-friendly tool for easy registration, updating, and retrieval of essential data, contributing to smoother administrative processes in healthcare.

The "Hospital Management System" project adheres to the SOLID principles, promoting maintainability, flexibility, and scalability in the software design. Let's explore each SOLID principle in detail and see how they are implemented:

Features

1. Patient Management:

- Registration of new patients.
- Updating and maintaining patient information.
- Viewing detailed patient profiles.

2. Doctor Management:

- Registering new doctors.
- Updating and managing doctor information.
- Viewing detailed doctor profiles.

3. Appointment Scheduling:

- Scheduling new appointments between patients and doctors.
- Viewing and managing existing appointments.
- Rescheduling or canceling appointments.

4. Medical Record Management:

- Storing and retrieving medical records for patients.
- Providing a detailed view of medical records.

5. Flexibility and Extensibility:

- Potential for further extension and addition of features.

6. Data Persistence:

- Storage and retrieval of patient, doctor, appointment, and medical record data.
- Use of text files for data persistence.

7. Operational Efficiency:

- Streamlining administrative processes for hospital staff.

- Enhancing overall operational efficiency in healthcare service delivery.

8. Appointment Filtering:

- Viewing appointments on a specific date.

GitHub Link:

https://github.com/MinhajulBhuiyan/Hospital-Management-System.git

Implementation of SOLID Principle

1. Single Responsibility Principle (SRP):

The SRP states that a class should have only one reason to change. In the project:

Example: PatientRegistrar Class

- Responsibility: Manages the registration of patients.

- Code Example:

```
public class PatientRegistrar : IRegistrar<Patient>
{
    private readonly IDataAccessor<Patient> dataAccessor;

    public PatientRegistrar(IDataAccessor<Patient> dataAccessor)
    {
        this.dataAccessor = dataAccessor;
    }

    public void Register(Patient patient)
    {
        // Registration logic
        // ...
        dataAccessor.SaveData(patients);
    }
}
```

2. Open/Closed Principle (OCP):

The OCP suggests that a class should be open for extension but closed for modification. In the project:

- Example: DoctorUpdater Class

  - Responsibility: Updates doctor information.

  - Code Example:

```
public class DoctorUpdater : IUpdater<Doctor>
{
    private readonly IDataAccessor<Doctor> dataAccessor;


public DoctorUpdater(IDataAccessor<Doctor> dataAccessor)
    {
        this.dataAccessor = dataAccessor;
    }

    public void UpdateInformation()
    {
        // Update logic
        // ...
        dataAccessor.SaveData(doctors);
    }
}
```

3. Liskov Substitution Principle (LSP):

The LSP emphasizes that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In the project:

- Example: PatientViewer Class

  - Responsibility: Views patient details.

  - Code Example:

```
public class MedicalRecordViewer : IViewer<MedicalRecord>
{
    private readonly IMedicalRecordService medicalRecordService;

    public MedicalRecordViewer(IMedicalRecordService medicalRecordService)
    {
        this.medicalRecordService = medicalRecordService;
    }

    public void ViewDetails(int patientId)
    {
        // Viewing logic for medical records
```

```
        var medicalRecord = medicalRecordService.RetrieveRecord(patientId);

        if (medicalRecord != null)
        {
            Console.WriteLine($"Medical Record for Patient ID {patientId}:");
            Console.WriteLine($"Details: {medicalRecord.Details}");
        }
    }
}
```

4. Interface Segregation Principle (ISP):

The ISP suggests that a class should not be forced to implement interfaces it does not use. In the project:

- Example: IAppointmentScheduler Interface

  - Responsibility: Schedules, cancels, and reschedules appointments.

  - Code Example:

```
public interface IAppointmentScheduler
{
    int ScheduleAppointment(int patientId, int doctorId, string problem, DateTime
appointmentDate);
    void CancelAppointment(int appointmentId);
    void RescheduleAppointment(int appointmentId, DateTime newDate);
}
```

5. Dependency Inversion Principle (DIP):

The DIP states that high-level modules should not depend on low-level modules but rather both should depend on abstractions. In the project:

- Example: DoctorManagement Class

  - Dependency Injection: Utilizes interfaces for doctor management.

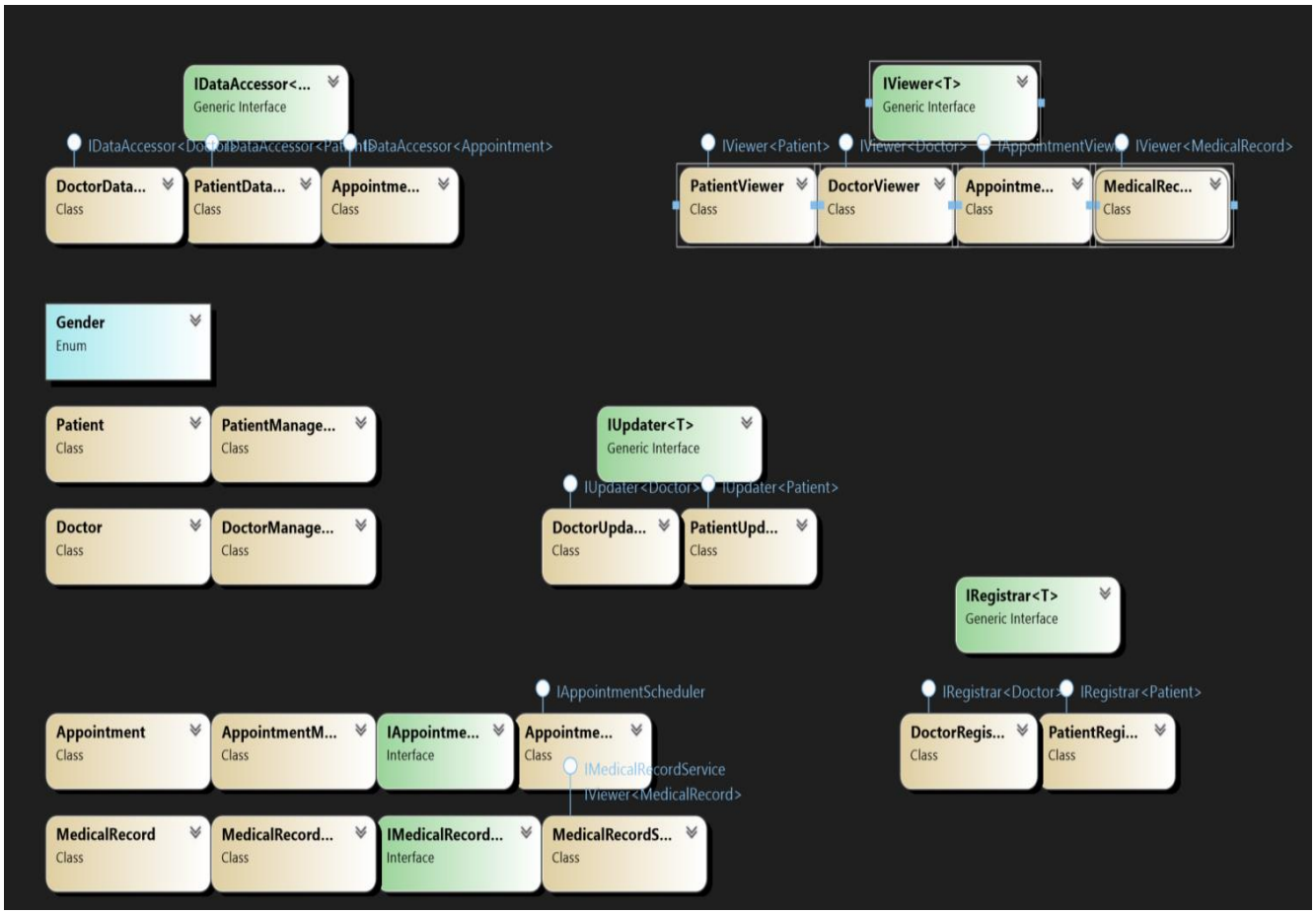
  - Code Example:

```
public class DoctorManagement
{
    private readonly IRegistrar<Doctor> doctorRegistrar;
    private readonly IUpdater<Doctor> doctorUpdater;
    private readonly IViewer<Doctor> doctorViewer;

    public DoctorManagement(IRegistrar<Doctor> doctorRegistrar, IUpdater<Doctor>
doctorUpdater, IViewer<Doctor> doctorViewer)
```

```
    {
        this.doctorRegistrar = doctorRegistrar;
        this.doctorUpdater = doctorUpdater;
        this.doctorViewer = doctorViewer;
    }

    // ... Other methods
}
```

These examples showcase how the SOLID principles are integrated into various components of the "Hospital Management System" project, contributing to a more robust and maintainable codebase.

UML Diagram:



Code Smell

In the "Hospital Management System" project, the implementation adheres to best practices, resulting in a codebase that is free from common code smells. Below are examples of how the project maintains clean, readable, and efficient code:

1. Code Smell: Duplicated Code

- Example: Reusable Code

  - Description: The project encourages code reuse to prevent duplications.

  - Code Example:

```
// Example of PatientViewer and DoctorViewer reusing the ViewDetails method from
IViewer interface
public class PatientViewer : IViewer<Patient>
{
    // ...
    public void ViewDetails(int patientId)
    {
        // Common viewing logic
        // ...
    }
}

public class DoctorViewer : IViewer<Doctor>
{
    // ...
    public void ViewDetails(int doctorId)
    {
        // Common viewing logic
        // ...
    }
}
```

2. Code Smell: Long Methods

- Example: Modularized Code

  - Description: The project follows a modular approach to break down long methods into smaller, more manageable components.

  - Code Example:

```
// Example of modularized code in DoctorUpdater class
public void UpdateInformation()
{
```

```
    // ... (common logic)

    Console.WriteLine("\nSelect what you want to update:");
    Console.WriteLine("1. Name");
    Console.WriteLine("2. Age");
    Console.WriteLine("3. Gender");
    Console.WriteLine("4. Specialization");
    Console.WriteLine("5. Details");

    Console.Write("\nEnter your choice: ");
    string updateChoice = Console.ReadLine();

    switch (updateChoice)
    {
        case "1":
            UpdateName(doctorToUpdate);
            break;

        // ... (other cases)

        default:
            Console.WriteLine("Invalid choice. No changes will be made.");
            return;
    }

    // ... (common logic)
}
```

3. Code Smell: Feature Envy

- Example 3: Proper Encapsulation

- Description: The project maintains proper encapsulation, avoiding feature envy where one class excessively uses methods or properties of another class.

- Code Example:

```
// Example from DoctorManagement class
public void ViewDoctorDetails()
{
            doctorViewer.ViewDetails(GetDoctorId());
}
```

By addressing these common code smells through principles like code reuse, modularization, and proper encapsulation, the "Hospital Management System" project achieves a clean and maintainable codebase.

Design Smells:

1. Rigidity:

- Example: Patient Management Module

- Description: Changes in patient-related functionalities do not lead to widespread modifications.

- Code Structure: The patient management module is designed with modularity to handle patient-specific operations without affecting other modules.

```
public class PatientManagement
{
    // ... (patient-related operations)
}
```

2. Fragility:

- Example: Doctor Management Module

-Description: Modifications in doctor-related functionalities are contained within the doctor management module, minimizing the risk of breaking unrelated parts.

- Code Structure: The doctor management module encapsulates doctor-specific operations.

```
public class DoctorManagement
{
    // ... (doctor-related operations)
}
```

3. Immobility:

- Example: Reusable Components

- Description: Code components are designed for reusability, reducing duplication and promoting good practices.

- Code Structure: Components like the `IViewer` interface enable viewing details for various entities.

```
public interface IViewer<T>
{
    void ViewDetails(int entityId);
}
```

4. Viscosity:

- Example: Encapsulation of Solutions

- Description: Good solutions are encapsulated, making them convenient and promoting best practices.

- Code Structure: Proper encapsulation of update methods in the `DoctorUpdater` class.

```
public class DoctorUpdater : IUpdater<Doctor>
{
    // ... (update methods)
}
```

5. Needless Complexity:

- Example: Balanced Design

- Description: The project avoids unnecessary over-design, maintaining a balance between simplicity and functionality.

- Code Structure: Designs focus on essential features without introducing undue complexity.

6. Needless Repetition:

- Example: Code Reusability

- Description: Repeated code is minimized, ensuring maintainability and reducing the risk of bugs.

- Code Structure: Reusable components like the `IRegistrar` interface support code reusability.

```
public interface IRegistrar<T>
{
    void Register(T entity);
}
```

7. Opacity:

- Example: Self-Documenting Code

- Description: Code readability is prioritized, reducing opacity and facilitating maintenance.

- Code Structure: Descriptive class and method names, self-explanatory comments, and adherence to coding standards.

```
public class MedicalRecordService : IMedicalRecordService, IViewer<MedicalRecord>
{
    // ... (self-documenting code)
}
```

By adopting principles like modularity, encapsulation, reusability, and readability, the "Hospital Management System" project mitigates design smells, resulting in a codebase that is robust, maintainable, and resistant to common design issues.

Conclusion: "Hospital Management System" stands as a testament to the effective implementation of SOLID principles, ensuring a clean, efficient, and user-friendly solution for hospital operations. By addressing code and design smells, we've crafted a high-quality, maintainable software that significantly contributes to improved healthcare services.