ASSIGNMENT FINAL REPORT

Qualification	**Pearson BTEC Level 5 Higher National Diploma in Computing**		
Unit number and title	**Unit 20: Applied Programming and Design Principles**		
Submission date	**12/12/2025**	**Date Received 1st Submission**	
Re-submission Date		**Date Received 2nd Submission**	
Student Name	**NGUYEN VIET PHUC**	**Student ID**	**BD00671**
Class	**SE07202**	**Assessor name**	**Mr VO DUC HOANG**

Plagiarism

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

Student Declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I declare that the work submitted for assessment has been carried out without assistance other than that which is acceptable according to the rules of the specification. I certify I have clearly referenced any sources and any artificial intelligence (AI) tools used in the work. I understand that making a false declaration is a form of malpractice.

Student's signature	**VIET PHUC**

Grading grid

P1	P2	P3	P4	M1	M2	D1

ASSIGNMENT GROUP WORK

Qualification	**Pearson BTEC Level 5 Higher National Diploma in Computing**		
Unit number and title	**Unit 20: Applied Programming and Design Principles**		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Group number:	**Student names & codes**	**Final scores**	**Signatures**
	Ho Duc Duong – BD00535		DUONG
	Phu Tuong Long – BD00772		LONG
	Nguyen Viet Phuc – BD00671		PhucViet
Class		**Assessor name**	Mr Vo Duc Hoang

Plagiarism

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

Student Declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I declare that the work submitted for assessment has been carried out without assistance other than that which is acceptable according to the rules of the specification. I certify I have clearly referenced any sources and any artificial intelligence (AI) tools used in the work. I understand that making a false declaration is a form of malpractice.

P5	P6	P7	M3	M4	D2

OBSERVATION RECORD

Student			
Description of activity undertaken			
Assessment & grading criteria			
How the activity meets the requirements of the criteria			
Student signature:		**Date:**	
Assessor signature:		**Date:**	
Assessor name:			

❒ Summative Feedback:	❒ Resubmission Feedback:

Grade:	Assessor Signature:	Date:

Internal Verifier's Comments:

Signature & Date:

TABLE OF CONTENT

LIST OF TABLE

LIST OF FIGURE

INTRODUCTION

The development of software is increasingly complex, requiring developers to adhere to strict design principles to ensure the system is stable, maintainable and scalable in the long term. In this context, Object-Oriented Programming (OOP) is not only a programming model but also a philosophical foundation for building modern software architecture. The purpose of this report is to analyze in depth the core principles of OOP, especially the SOLID principles, and their role in designing an enterprise-level system.

The report will focus on the following contents:

- A detailed analysis of the basic concepts of OOP (Encapsulation, Inheritance, Polymorphism, Abstraction) and how they are expressed in system design.
- Present and Apply the five SOLID principles (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to demonstrate the construction of a Clean Architecture and Clean Code.
- System Architecture Design: Apply the above principles to design a Student Information Management System (SIMS) using a Three-Tier Architecture and appropriate Design Patterns (such as Repository, Factory Method, Singleton).
- Testing Process: Outline a comprehensive testing strategy, including Unit Test, Integration Test and System Test, to ensure the correctness and reliability of the system.
- The result of this report not only provides theoretical insight but also a practical demonstration of applying industry best design practices to a real software project.

We would like to express our sincere gratitude to Mr. Vo Duc Hoang, our supervisor, for his dedicated support, expertise and detailed guidance throughout the process of completing this major assignment. His support was the key factor in helping us complete the report with the highest quality.

1. Investigate the characteristics of the object- orientated paradigm, including class relationships and SOLID principles. (P1)

1.1. Investigate Object-Oriented Paradigm

As the name suggests, Object Oriented Programming is a programming language that uses objects. The goal of OOP is to implement real-world entities like inheritance, implicit typing, polymorphism, etc. in programming. The main goal of OOP is to bind data and assign functions to them so that no other part of the code can access that data except that function **(GeeksforGeeks , 2020).**

- Encapsulation is the practice of gathering data and each method that operates on the data into a single unit, for example, in a class that is implementing the desired implementation. It will hide the internal details from outside access, protecting the integrity of the data.
- Abstraction simplifies complex systems by exposing only the relevant properties and behaviors. For example, an abstract class User can define methods like Login() and Logout() that are implemented differently by subclasses.
- Inheritance is allowed for new classes to inherit from existing classes, and from there to reuse and extend functionality.For example, Student and Teacher inherit from Person.
- Polymorphism allows one interface to represent completely different data types or behaviors, thus increasing code flexibility.

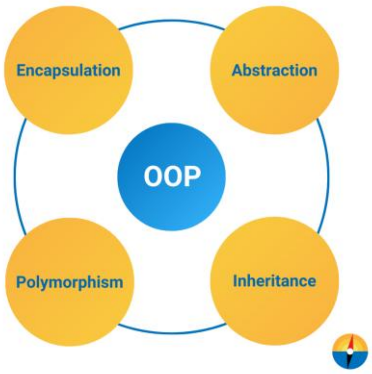


Figure 1.1 OOP

1.2. Class Relationships

In object-oriented design, classes can interact and relate to each other in a variety of ways, which we will see in real-world relationships. The most common class relationships include:

- **Association**: A general relationship between two classes, in which one class is used for the functionality of the other.
- **Aggregation**: A relationship in which a containing class contains references to each other, but can exist independently.
- **Composition**: A more powerful form of aggregation, in which the existence of contained objects depends on the containing objects.

- **Inheritance**: A hierarchical relationship that allows a child class to inherit properties and methods from a parent class.

Understanding these relationships allows developers to design flexible systems that follow real-world logical structures and improve code organization.

1.3. SOLID Principles

The SOLID principles are a set of design guidelines within object-oriented programming that aim to make software easier to understand, more flexible, and more maintainable. Each principle addresses a specific challenge in software design development:

- S – Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should perform only one function or represent only one concept.
- – Open/Closed Principle (OCP): Each software entity should be open to be extended but closed to be modified. This encourages adding new functionality without changing existing code.
- L – Liskov Substitution Principle (LSP): Subclasses should be able to replace their base classes without affecting the correctness of each program.
- I – Interface Segregation Principle (ISP): Individual clients should not be forced to depend on interfaces they will not use. This promotes the creation of smaller, more specific interfaces.
- D – Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstract layers.

Figure 1.2 SOLID

Applying these principles helps prevent tight coupling between components and promotes extensibility, ease of testing, and long-term maintainability of software systems.

2.Explain how clean coding techniques can impact on the use of data structures and operations when writing algorithms. (P2)

“Clean code is a term used to refer to code that is easy to read, understand, and maintain. It was made popular by Robert Cecil Martin, also known as Uncle Bob, who wrote "Clean Code: A Handbook of Agile Software Craftsmanship" in 2008. In this book, he presented a set of principles and best practices for writing clean code, such as using meaningful names, short functions, clear comments, and consistent formatting.” **(Codacy , 2023)**

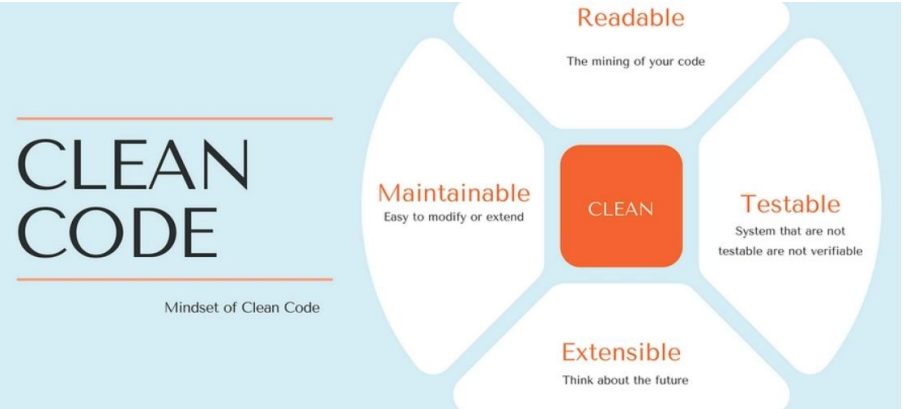


Figure 2.1 Clean code

☐ Meaningful Naming and Readability

One of the basic principles of clean code is to give clear and meaningful names to variables, functions and classes . This helps readers understand the purpose of each part of the code and avoids mistakes when working in groups.

- **Unclean writing**

```
List<string[]> s = new List<string[]>();
s.Add(new string[] { "SV001", "Nguyen Van A", "CNTT" });
Console.WriteLine(s[0][1]); // What is index[1]? confusing
```

Figure 2.2*Unclean c# code*

- **How to write cleanly**

```
public class Student
{
    0 references
    public string Id { get; set; }
    0 references
    public string FullName { get; set; }
    1 reference
    public string Major { get; set; }

}
static void Main(string[] args)
{

    List<Student> studentRegistry = new List<Student>();
    studentRegistry.Add(new Student { Id = "SV001", FullName = "Nguyen Van A", Major = "CNTT" });
    Console.WriteLine(studentRegistry[0].FullName);

}
```

Figure 2.3 *Clean c# code*

- **Explain** : In the clean version , the Student class clearly describes the data structure. Names like studentRegistry , FullName clearly indicate the purpose , making the code easier to read and reducing logical errors when manipulating data.

☐ Modularity and the Single Responsibility Principle

Clean code encourages breaking down a program into discrete modules — each module should do only one thing. This principle is the SRP principle in the SOLID framework.

- **Unclean writing**

```
0 references
void ProcessAndDisplayStudents(List<Student> data, string major)
{
    var temp = new List<Student>();
    foreach (var s in data)
        if (s.Major == major) temp.Add(s);

    temp.Sort((a, b) => a.FullName.CompareTo(b.FullName));

    foreach (var s in temp)
        Console.WriteLine($"{s.Id} - {s.FullName}");
}
```

Figure 2.4 Unclean c# code modularity

- **How to write cleanly**

```
List<Student> FilterStudentsByMajor(List<Student> students, string major) =>
students.Where(s => s.Major == major).ToList();

1 reference
void SortStudentsByName(List<Student> students) =>
    students.Sort((a, b) => string.Compare(a.FullName, b.FullName));

1 reference
void PrintStudentList(List<Student> students, string title)
{
    Console.WriteLine($"--- {title} ---");
    foreach (var s in students)
        Console.WriteLine($"{s.Id} - {s.FullName}");
}

0 references
void DisplayMajorReport(List<Student> allStudents, string major)
{
    var filtered = FilterStudentsByMajor(allStudents, major);
    SortStudentsByName(filtered);
    PrintStudentList(filtered, $"List student margior {major}");
}

}
```

Figure 2.5 Clean modular c# code

- **Explain** : Each function now does just one thing — filter, sort, or print the list. This design increases reusability, makes it easier to test, and makes it easier to extend when you want to add new functionality, like sorting by GPA.

☐ **Comments and documentation**

Clean code discourages writing redundant comments, which should only be used to explain why a decision was made.

```
// Use HashSet to ensure no duplicate student codes (O(1) lookup)
HashSet<string> studentIds = new HashSet<string>();
```

Figure 2.6 Comment code

Such proper notation helps the reader understand the purpose of the chosen data structure without having to read through the entire logic.

☐ **Consistency and Naming Conventions**

- Maintaining consistency in naming, indentation, and function organization makes the algorithm easier to read.
- For example, always use PascalCase for function names (SortStudentsByName) and camelCase for variables (StudentRegistry).
- This helps code look consistent and is easier to maintain when working in teams.

☐ **Overall impact**

- Applying clean code techniques to data structures and algorithms helps:
- Increase readability: newcomers can quickly understand the data processing flow.
- Increase maintainability: when changing data structures, no need to edit in many places.
- Increase scalability: new features can be added without breaking the old system.
- Increase teamwork efficiency: clear code makes it easy for other members to collaborate.
- In the SIMS system, these techniques help the student data processing program to be fast, stable and easy to expand when the number of students or courses increases.

3. Design a large data set processing application, utilizing SOLID principles, clean coding techniques and a design pattern. (P3)

3.1.Requirements Analysis and Architecture Selection

3.1.1.Functional Requirements

Based on the descriptions of each project aspect, the SIMS systems will have to meet the requirements of the following core functions:

- Student Registration: The systems will have to allow for efficient registration of new students, and from here, essential information including personal information and academic records will be recorded and stored.
- Course Management: With the tasks being provided with each function for administrators, from here, each course provided by the university will be managed and students assigned to each course based on their study program.

- User Authentication and Authorization: The system must ensure secure user authentication for very different objects (students, lecturers, administrators) and from here role-based access control will be deployed to limit system functions depending on user roles.

3.1.2.Non-Functional Requirements

In addition to each function, the systems must also comply with the following important non-functional requirements to ensure quality and sustainability:

- Scalability: Having the ability to handle the number of students and each course will increase over time.
- Performance: With each response to user requests within a time frame, it will be acceptable, even during peak usage times.
- Security: Implementing each measure to be able to be strongly secured to protect the sensitive information of each student and from here will ensure the integrity of the data.
- Usability: With extremely friendly interfaces, it will be easy to use for users with extremely different technical levels.
- Accessibility: Compliance with each accessibility standard so that it will be ensured that the system can be used by people with disabilities.
- Reliability: Operation with each stability with downtime will be minimal for maintenance or unexpected incidents.

3.1.3.Choice of architecture (ASP.NET Core) and model

Technology Specified: For each requirement, these applications will be implemented using ASP.NET Core.

- **Proposed Architecture:** For each requirement, we will propose to apply the 3-Layer Architecture. This is an architectural pattern that can be tested, and from here the application is divided into three separate logical layers:
- **Presentation Layer:** With responsibility for the user interface (UI) and also from here will interact with each user. In the context of ASP.NET Core, this can be with Razor pages, MVC (Model-View-Controller) or API Endpoints.
- **Business Logic Layer (BLL):** This layer will contain each core business logic as well as each rule and from there, for the workflow of each application (for example, about checking each course registration condition, from there combined with processing authentication logic).
- **Data Access Layer (DAL):** This layer will be responsible for all communication activities with data sources. In these projects, it will handle reading and writing data from CSV files.

❖ **Reasons for choosing:**

- Separation of Concerns: Each of these architectures implements the Single Responsibility Principle (SRP) at the architectural level. Each layer has a clear responsibility. The DAL layer only takes care of data, the BLL only takes care of business, and the Presentation only takes care of display.
- Meeting the Maintenance & Scalability requirement: The clear separation and from here will allow the development team to easily maintain and from here can extend each part of the system. For example, if in the future, universities want to change the data source from CSV to SQL database (and from here will meet the Scalability requirement), we only need to modify the DAL layer without affecting the BLL or Presentation layer.
- Increased Testability: The fact that they can separate business logic (BLL) from UI and data (DAL) makes it easier to write Unit Tests for business rules, contributing to system reliability.

3.2.System Design

With this presentation, it is designed with each overall aspect of the Student Information Management System (SIMS) applications.

With each design phase of each system of the transformations by the requirements gathered in each phase of each analysis aspect into one of the representations for the structures and logic of the system.

To ensure the ability with each maintenance, as well as from here will create the extensibility and also from here will be complied with the SOLID principles, as well as fully proposed architecture follows the three-layer model including Presentation Layer, Business Logic Layer and Data Access Layer.

With such system plans will be represented by three essential UML diagrams: Use Case Diagram, Class Diagram and Package Diagram, each diagram plays a specific role in illustrating the overall architecture and design of the application.

3.2.1.Use Case Diagram

"Use cases are built to refine a set of requirements based on a role or task. Instead of the traditional list of requirements that may not directly address the use of the solution, use cases group common requirements based on the type of role or goal. Use cases define what the users or roles are doing in the solution, a business process defines how they perform those functions.".(IBM, 2025)

In the SIMS application context each main actor will be comprised of Student, Lecturer and Administrator. Each of these actors interacts with the system through specific use cases that represent each core functionality of the platform.

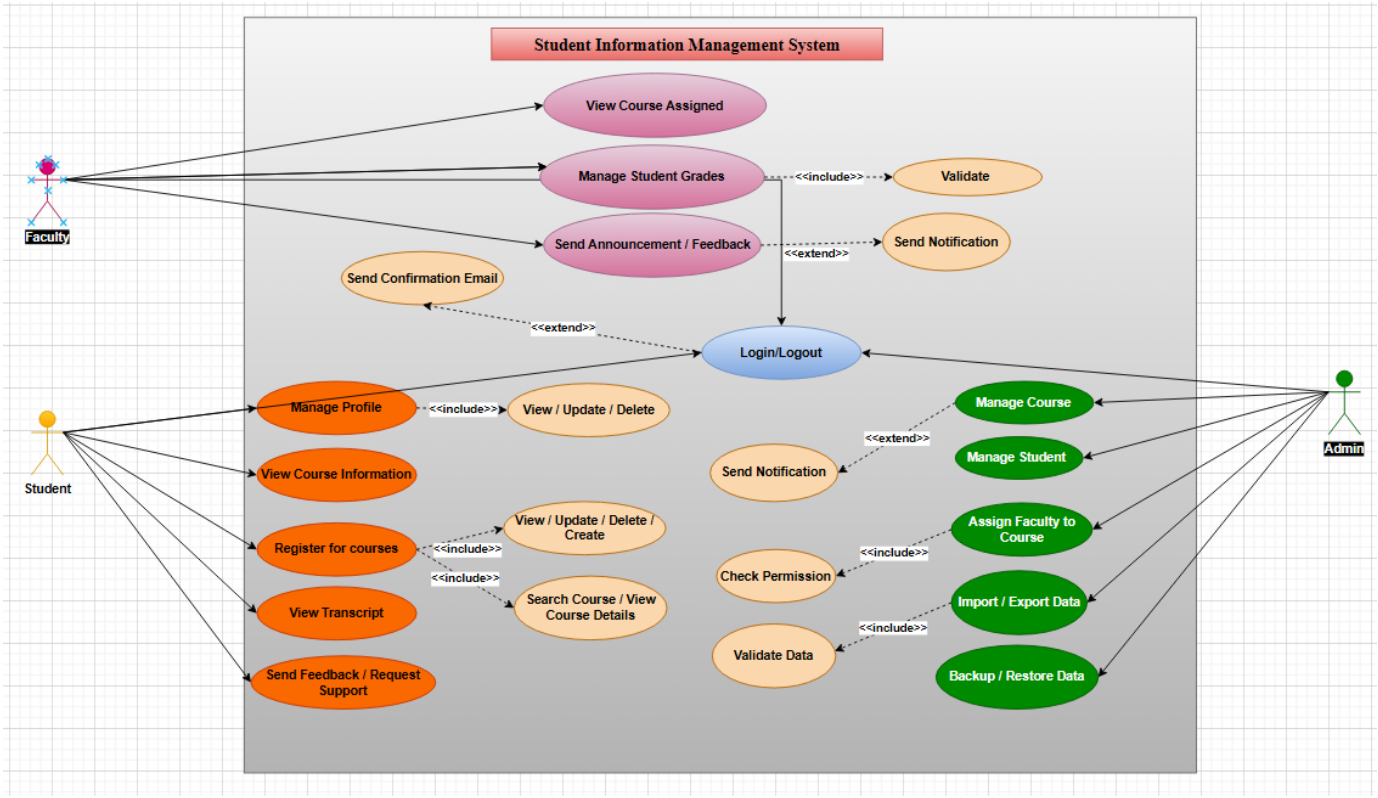


Figure 3.1Use Case Diagram of Student Information Management System

❖ Describe the use case of the system

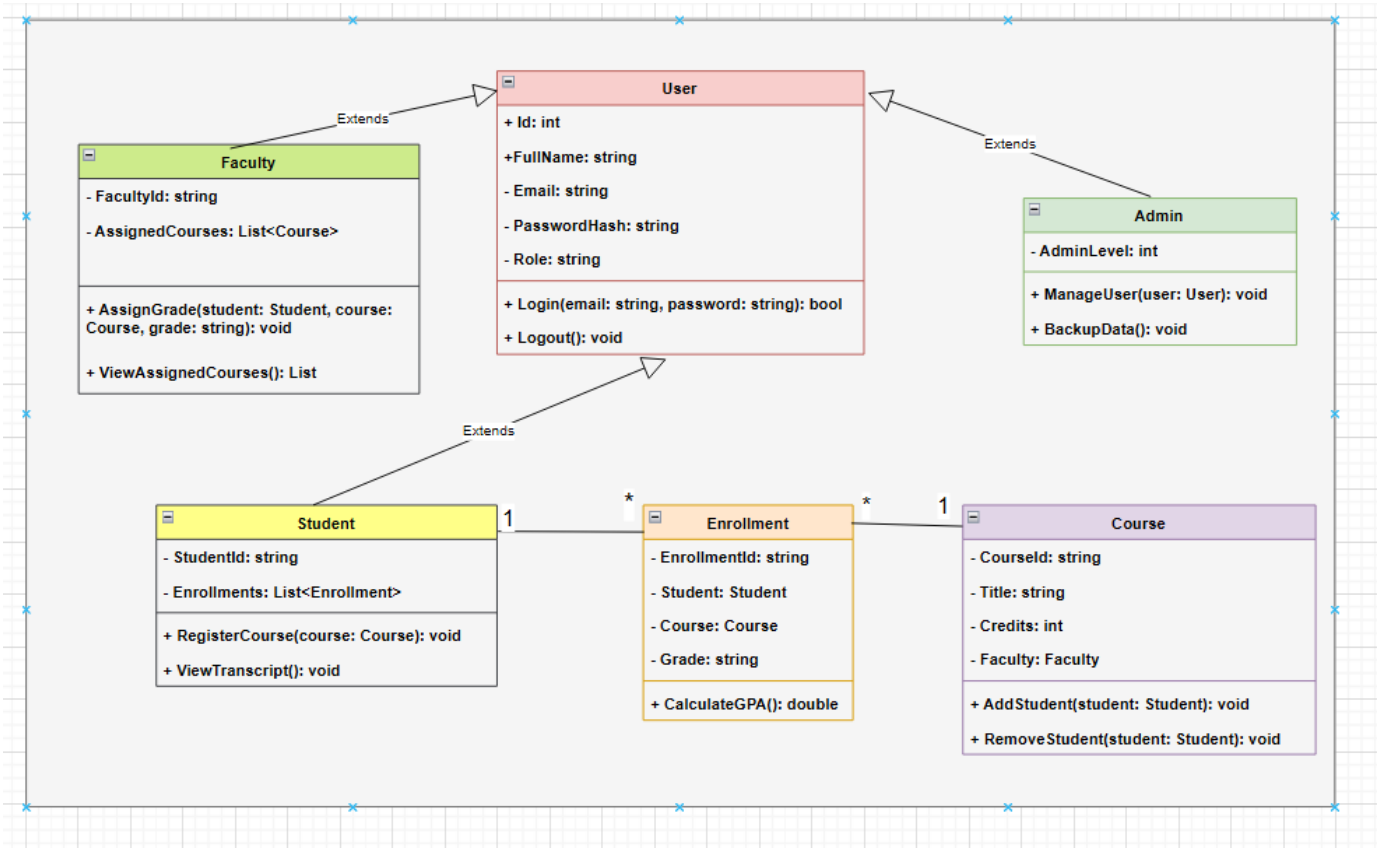
Use Case Name	Actors Involved	Description / Objective	Preconditions	Postconditions / Outcomes
Login	Student, Faculty, Admin	Allows users to log into the system with valid credentials and access features based on their roles.	The user must have an existing account.	User is authenticated and directed to their dashboard.
Register Student	Student, Admin	Enables new students to register by providing necessary personal and academic details.	None	A new student record is created in the database.
Manage Courses	Faculty, Admin	Enables the creation, editing, and deletion of course information.	User must have Faculty or Admin privileges.	Course list updated successfully.
Enroll in Course	Student	Allows students to enroll in available courses according to their program.	Student must be logged in.	Enrollment record added to the system.

View Grades	Student, Faculty	Faculty members can upload grades; students can view their own grades.	Student is enrolled in a course.	Grades are updated or displayed correctly.
Manage Users	Admin	Admin can add, remove, or modify user accounts and assign roles.	Admin must be logged in.	User information updated in the system.
Logout	All actors	Ends the user session securely.	User must be logged in.	Session terminated, system returns to login screen.

Table 3.1 Use Case Descriptions of SIMS

3.2.2.Class Diagram

"A class diagram is a visual way of depicting the data to be held within a system, the way the various classes of data are connected and their relationship to each other. It is drawn using a standard notation; Unified Modelling Language (UML). Class diagrams can be useful to determine the data requirements of the new/current business system.".(Service, I. 2020)



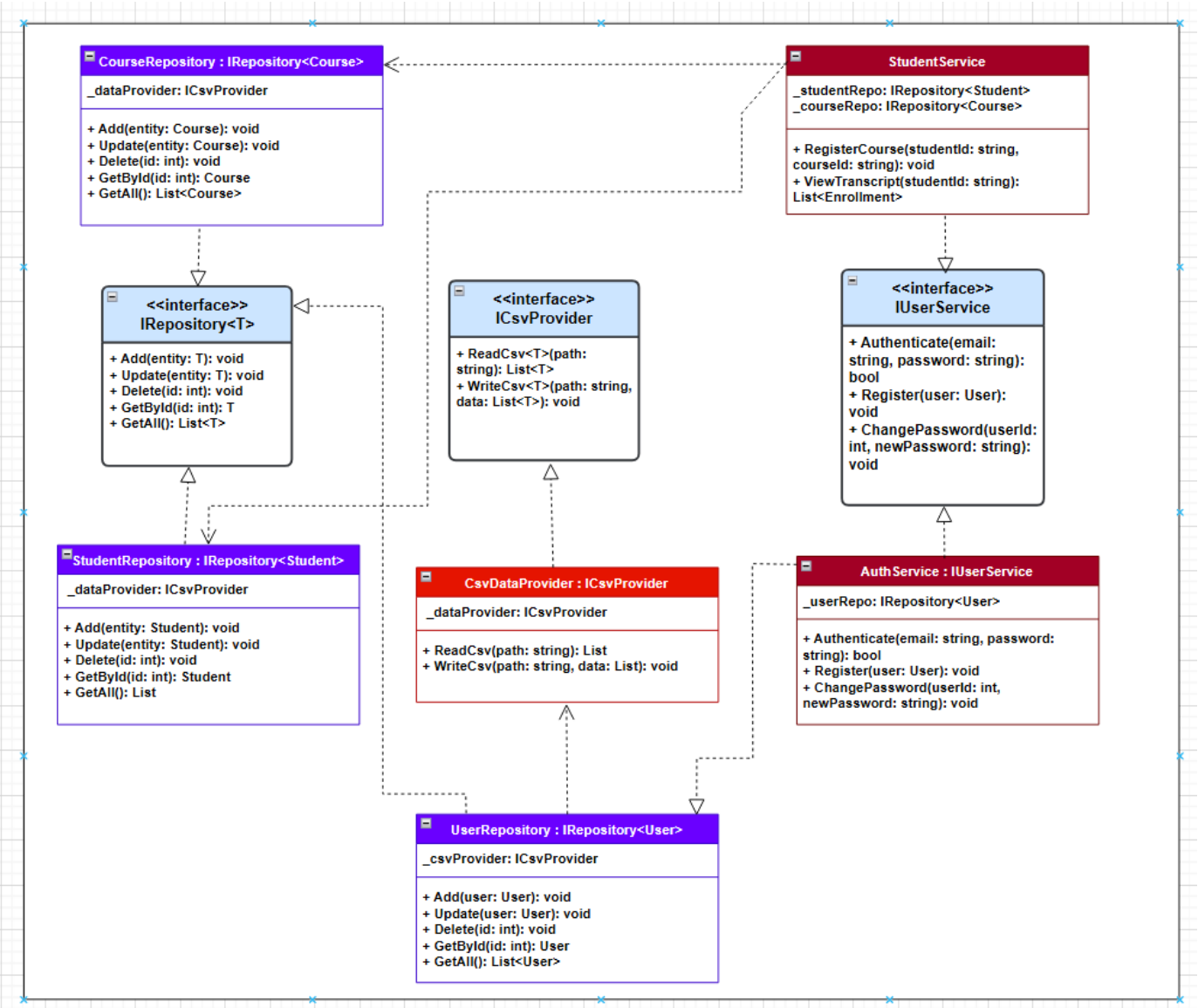


Table 3.2 Class Diagram of SIMS following SOLID Principles

With class diagrams showing the internal structure of each SIMS system, as well as each illustration of the main components, their attributes, operations and relationships.

Class diagrams represent the backbone of the system's object-oriented design, adhering to the SOLID principles of flexibility and maintainability.

With each job being designed organizationally into separate logical layers:

- Entity Layer: It will contain by data models like User, Student, Faculty, Admin, Course and Registration. With each of these models representing the real-world entity of the system.
- Repository Layer: With the job will consist of interfaces (IRepository<T>) and from there will be concrete classes (UserRepository, CourseRepository, StudentRepository) all of which will be handling the storage and from there will also be retrieving the data.

- Service Layer: With the job will consist of business logic classes like AuthService and StudentService, implementing their respective interfaces (IUserService).
- Provider Layer: Includes ICsvProvider and CsvDataProvider, responsible for reading and writing data to CSV files.

Design Analysis:

- Single Responsibility Principle (SRP) ensures that each class has only one purpose with a clearly defined purpose (e.g. StudentService only handles student logic).
- Open/Closed Principle (OCP) allows for extending functionality (e.g. adding new services) without modifying existing code.
- Dependency Inversion Principle (DIP) is achieved through interfaces such as IRepository<T> and IUserService, and is ensured by each high-level module depending on the abstract classes rather than the implementation classes.

Relationships are represented through UML notation:

- Dotted lines with hollow triangles → Interface implementations.
- Dotted lines with open arrows → Dependencies.
- Solid Line → The association between entities.
- This diagram illustrates one of the clear modular structures that promotes testability, extensibility, and maintainability — essential for enterprise-grade systems.

3.2.3.Package Diagram

"A package diagram in UML (Unified Modeling Language) is a structural diagram that groups related elements such as classes or components into logical units called packages. It shows how these packages are organized and depend on each other, offering a clear view of a system's structure. By grouping related parts, package diagrams simplify complex designs, promote modularity, and ensure a clean separation of responsibilities within large software systems.".(Perera, N. 2025)

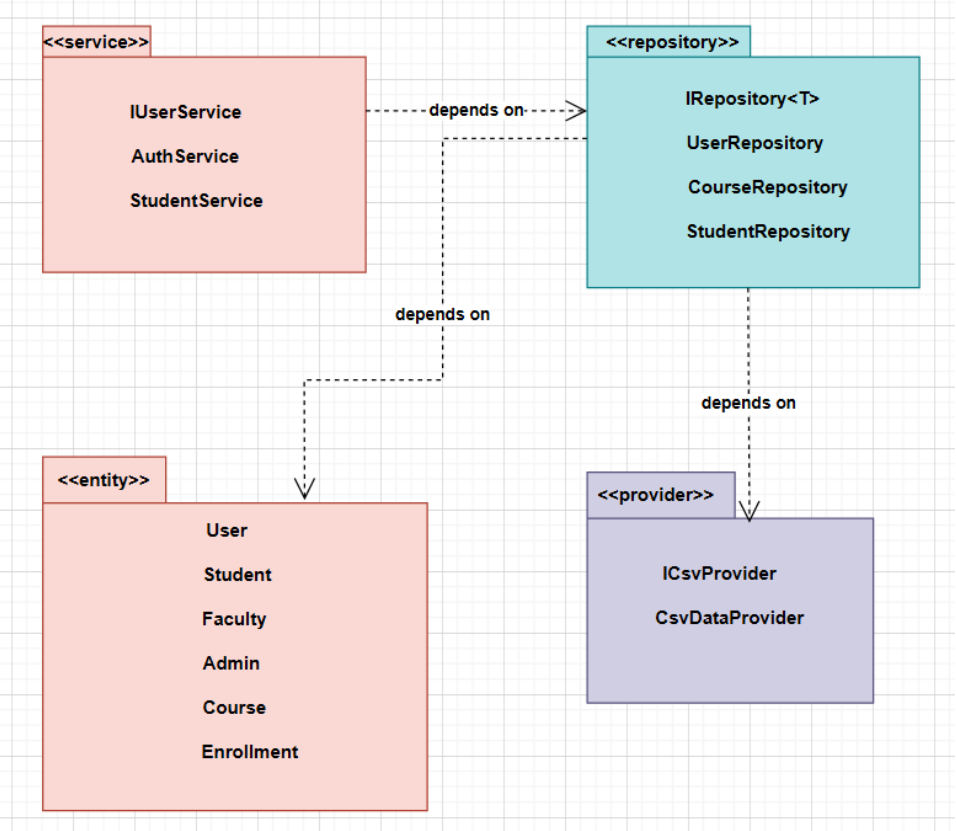


Figure 3.2 Package Diagram of SIMS Architecture

Overview of the system packages:

- Entity (Domain) package: With the job of containing the data models of each system such as User, Student, Lecturer and Course. With each of these models, it represents the actual information managed by the application.
- → This class is independent, meaning it will not depend on any other package.

Repository package:

- With the job of managing the persistence of data through classes such as IRepository<T>, UserRepository and CourseRepository.
- → Depends on the Entity package in terms of data structure and the Provider package in terms of file access.

Provider package:

- Provides data import/export operations through ICsvProvider and CsvDataProvider.
- → Is the lowest level package, has no dependencies.

Service Package: With the handling of each business logic using IUserService, AuthService and StudentService.

→ Depends on both the Repository and Entity packages to execute in order to do its operation.

Design Interpretation:

- With the dependency paths (shown by open arrows), it is clear that each higher-level class depends on each lower-level class.
- This structure is implemented in Clean Architecture and also aims at the Dependency Inversion Principle, ensuring that:
- Each business rule (Service layer) is also independent of the infrastructure details (Repository and Provider).
- Any changes in each data source (e.g., replacing CSV with SQL) only require modifications in the Repository and Provider layers — not in the Service or Entity layers.
- Thus, the Package Diagram reinforces the concept of Separation of Concerns, allowing for long-term scalability and maintainability.

3.3.Applying Design Principles

3.3.1. Explain the application of SOLID principles in design

The design also incorporates Clean Code techniques to ensure clarity, readability, and maintainability of the codebase.

Key practices implemented in SIMS include:

- Meaningful Naming Conventions
- All classes, methods, and variables use descriptive names such as EnrollCourse() or ValidateUser() to make the purpose of each element clear.
- Separation of Concerns
- The three-layer architecture (Presentation, Business Logic, Data Access) ensures each component focuses on a single responsibility, preventing logic duplication.
- Consistency and Readability
- Code will follow consistent indentation, casing, and naming standards (e.g., PascalCase for classes, camelCase for methods and variables in C#).
- Avoiding Code Duplication
- Common behaviors (e.g., file reading/writing) are abstracted into the ICsvProvider and reused across repositories.
- Error Handling and Validation

- Validation logic is implemented at the Service Layer, ensuring that invalid data does not reach the data layer.
- Comments and Documentation
- Essential comments are included only where needed, ensuring self-documenting code that remains clean and easy to understand.

3.3.2.Applying Clean Code techniques in design

The design also incorporates Clean Code techniques to ensure clarity, readability, and maintainability of the codebase.

Key practices implemented in SIMS include:

- Meaningful Naming Conventions
- All classes, methods, and variables use descriptive names such as EnrollCourse() or ValidateUser() to make the purpose of each element clear.
- Separation of Concerns
- The three-layer architecture (Presentation, Business Logic, Data Access) ensures each component focuses on a single responsibility, preventing logic duplication.
- Consistency and Readability
- Code will follow consistent indentation, casing, and naming standards (e.g., PascalCase for classes, camelCase for methods and variables in C#).
- Avoiding Code Duplication
- Common behaviors (e.g., file reading/writing) are abstracted into the ICsvProvider and reused across repositories.
- Error Handling and Validation
- Validation logic is implemented at the Service Layer, ensuring that invalid data does not reach the data layer.
- Comments and Documentation
- Essential comments are included only where needed, ensuring self-documenting code that remains clean and easy to understand.

3.3.3.Design Pattern

To enhance reusability and maintainability, the SIMS architecture incorporates design patterns that align with SOLID principles and clean architecture.

❖ **Repository Pattern :**

- The Repository Pattern is implemented to separate the business logic from data access logic.
- The IRepository<T> interface defines the generic CRUD operations (Add, Update, Delete, GetAll).
- Concrete repositories such as StudentRepository, CourseRepository, and UserRepository implement these interfaces.
- The Service layer interacts only with the interface, not with the actual implementation.

❖ **Advantages in SIMS:**

- Promotes Separation of Concerns between data and business logic.
- Facilitates mock testing and improves data abstraction.
- Supports easy migration — e.g., from CSV storage to SQL database — without affecting upper layers.
- Dependency Injection (DI) Pattern
- The SIMS design also applies the Dependency Injection Pattern at the architectural level:
- Services receive repository and provider dependencies through their constructors (e.g., StudentService(IRepository<Student> repo)), rather than creating them internally.
- This aligns with the Dependency Inversion Principle and makes testing and maintenance more efficient.
- By integrating these patterns, SIMS ensures reusability, flexibility, and maintainable architecture.

3.4.Test Process Design

Testing is an essential phase that ensures the SIMS system performs correctly and meets the specified requirements.

The testing process covers all levels — from unit-level checks to system-wide validation — and includes an automated testing plan for efficiency.

3.4.1.Testing levels (Unit Test, Integration Test, System Test)

Testing Level	Purpose	Example in SIMS
Unit Testing	To verify individual methods or classes work correctly in isolation.	Testing methods in AuthService (e.g., ValidateUser()), StudentRepository, or CsvDataProvider.
Integration Testing	To test the interaction between different modules or layers.	Testing the integration between StudentService and Repository to ensure data is stored and retrieved correctly.
System Testing	To validate the system as a whole, ensuring all modules work together according to requirements.	Simulating a real-world workflow such as student login → course enrollment → viewing grades.

Figure 3.3 Testing level

❖ **Testing Objectives:**

- Validate both functional and non-functional requirements.
- Detect bugs early through automated and manual test execution.
- Ensure overall reliability, security, and performance.

3.4.2.Automated testing plan

To ensure testing consistency and reduce manual effort, SIMS applies an Automated Testing Strategy using available testing frameworks in ASP.NET Core (e.g., xUnit, NUnit).

Aspect	Description
Framework	xUnit framework used for unit and integration tests.
Scope	Test all service and repository classes. Mock dependencies using interfaces (IRepository<T>, ICsvProvider>).
Test Cases	Authentication (valid/invalid login), Course Enrollment, Data persistence, File read/write operations.
Continuous Integration	Integrate automated testing with GitHub Actions or Azure DevOps pipelines for continuous feedback.
Expected Outcomes	Each unit passes its functional test; integrated modules behave as expected with no data loss or logic errors.

Figure 3.4 Automated Testing Plan

4. Design a suitable testing regime for the application, including provision for automated testing (P4)

4.1.Automated Testing

"Automated testing is the implementation of an automation tool to execute test cases."(SmartBear, 2019)

These methods are suitable for large projects or those that require multiple testing. They are also suitable for projects that have undergone manual testing initially. With automation, testers can spend more time on high-value tasks. Although these require testers to maintain test scripts, in the long run, it will help increase application quality, test coverage, and scalability.

Figure 4.1 Automatic testing

The main objectives of automation testing in this project are:

- Minimize human errors during repetitive testing.
- Quickly identify errors after code changes.
- Check the consistency of application logic and data processing.
- Improve development efficiency and reliability before deployment.

Tools and Frameworks Used

Tool / Framework	Description	Purpose
xUnit	A popular C# testing framework	Used for writing unit and integration tests
Visual Studio Test Explorer	Built-in testing interface in Visual Studio	To run and view test results easily
Moq	.NET mocking framework	To isolate classes during testing
Postman	API testing tool	To automate and verify API endpoints

.NET CLI (dotnet test)	Command-line testing runner	To automate testing during build process

Table 4.1 *Tool and framworks used*

Test Categories Used in Automation

❖ **Unit Testing**

With unit testing, it is written to test small sections of code, such as each validation or logic function, or to ensure each component works as expected in isolation.

For example, verifying that a student cannot be registered without a valid email.

❖ **Integration Testing**

With integration testing, it is designed to ensure that the different parts of the system (such as each controller, service, and data storage side) work together correctly.

For example, verifying that when a student is added, their data is stored correctly in a CSV file.

❖ **API Testing (Functional)**

Automated API testing verifies that the system responds correctly to user requests.

For example, use Postman to here request to automatically send a POST request to /api/students and check for a "201 Created" response.

- **Example of Automated Unit Test (xUnit)**

```
using Xunit;
using SIMS.Services;
using SIMS.Models;
using System;

namespace SIMS.Tests
{
    0 references
    public class StudentServiceTests
    {
        [Fact]
        | 0 references
        public void AddStudent_ValidData_ShouldReturnSuccess()
        {
            var service = new StudentService();
            var student = new Student(
                roll: "R001",
                first: "Nguyen",
                full: "Nguyen Van A",
                email: "a@student.com",
                dob: new DateTime(2000, 1, 1),
                cls: "IT101"
            );

            var result = service.AddStudent(student);

            Assert.True(result.IsSuccess);
        }
```

```
        [Fact]
        0 references
        public void AddStudent_EmptyEmail_ShouldReturnFailure()
        {
            var service = new StudentService();
            var student = new Student(
                roll: "R002",
                first: "Le",
                full: "Le Thi B",
                email: "",
                dob: new DateTime(2001, 2, 2),
                cls: "IT101"
            );

            var result = service.AddStudent(student);

            Assert.False(result.IsSuccess);
        }
    }
}
```

Figure 4.2Automated Unit Tests successfully executed in Visual Studio

Figure 4.3 Test run summary showing both tests passed

4.2.Testing Application

- **Purpose of Application Testing**

The purpose of each application testing is to ensure that all the main features of the SIMS (Student Information Management System) are working correctly and without errors.

Each test is performed to check whether the system functions properly when the user enters both valid and invalid data.

Testing focuses on:

- Checking whether each function returns the correct result.
- Verifying whether the validation rules (e.g. email format) are working properly.
- Ensuring that the system does not crash when incorrect data is entered.
- Verifying that the application is easy to use and works smoothly.

- **Testing Approach**

The testing was done in two simple ways:

- Manual testing – by running the application, entering different inputs, and observing results.
- Automated testing – using Visual Studio's Test Explorer to run unit tests created in the SIMS.Tests project.

Both methods helped identify problems early and confirm that the program met the functional requirements.

- **Testing Environment**

During manual testing, the student tried entering valid and invalid data into the functions. The purpose was to see how the system responded in real use situations.

Test ID	Function Tested	Input Example	Expected Result	Actual Result	Status
MT01	Add Student (valid)	Full name: "Nguyen Van A", Email: "a@student.com"	Message "Student added successfully"	Displayed correctly	Pass
MT02	Add Student (missing email)	Full name: "Le Thi B", Email: empty	Error "Email is required"	Displayed correctly	Pass
MT03	Add Student (invalid email)	"abc@"	Error "Email is invalid"	Displayed correctly	Pass
MT04	View Student List	Click "Show All"	All saved students displayed	Displayed correctly	Pass
MT05	Exit Program	Click Exit	Program closes without error	Works properly	Pass

Table 4.2 Test enviroment

4.3.Experimental Design

With each test design describing each way the testing process will be planned and from here it will also be carried out to ensure the results are valid, consistent and also from here it will be measured.

In this project, between the tests were carried out mainly to test the functionality, performance and reliability of SIMS (Student Information Management System).

This section explains:

- What was tested
- What data was used
- How the testing was conducted
- What environment and tools were used to measure the results

- **Testing Objectives**

The purpose of the test is to:

- Verify that all the functions (add, view, authenticate students) work correctly.
- Check that the system can handle many records efficiently.
- Measure the processing time of larger data files (CSV import).
- Observe how the application behaves when invalid data is entered.
- Ensure that the system is stable and does not crash.
- **Testing Procedure**

Step	Description
1. Setup Environment	Open Visual Studio, ensure both SIMS and SIMS.Tests projects run correctly.
2. Prepare Data	Create or load CSV files with different sizes (100, 1000, 5000 records).
3. Run Automated Tests	Use xUnit → "Run All Tests" to verify logic automatically.
4. Run Manual Tests	Manually add, view, and validate student records in the application.
5. Measure Performance	Record the time taken to import different file sizes.
6. Record Results	Note down Pass/Fail and any error messages.
7. Analyse Results	Compare expected vs. actual outcomes.

Table 4.3 *Testing procedure*

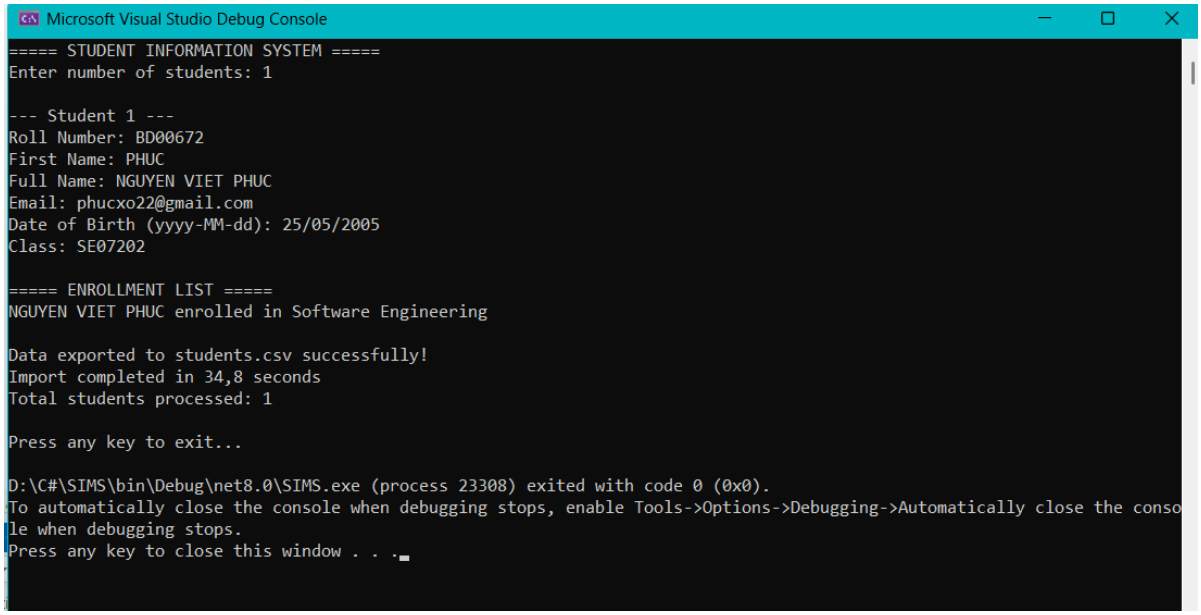


Figure 4.4 Performance Testing Results

5.Build a large dataset processing application based on the design produced. (P5)

5.1. Requirements analysis and design

The university has decided to modernize its Student Information Management System (SIMS) to improve efficiency, maintainability, and long-term security. Our team has been selected to design and implement this new system following object-oriented principles, SOLID guidelines, clean code practices, and suitable design patterns. Before development begins, we must conduct a clear analysis of user

In addition, several non-functional requirements guide the system design: it must be scalable for future growth, maintain strong performance under peak usage, include robust security for sensitive data, provide high usability for all user groups, meet accessibility standards, and ensure overall reliability with minimal downtime

- **Use case diagram**

Based on the analysis of the scenario and the identified functional and non-functional requirements for the Student Information Management System (SIMS), we developed a use case overview to clearly represent how users interact with the system. The purpose of this use case description is to illustrate the relationships between key actors such as students, administrators, and faculty and the core functions provided by the system. These interactions include student registration, course management, and user authentication and authorization. The use case ensures that all essential behaviors of the system are captured, supporting the needs of stakeholders while aligning with principles of scalability, security, usability, and reliability. This serves as the foundation for designing the system's architecture and future implementation

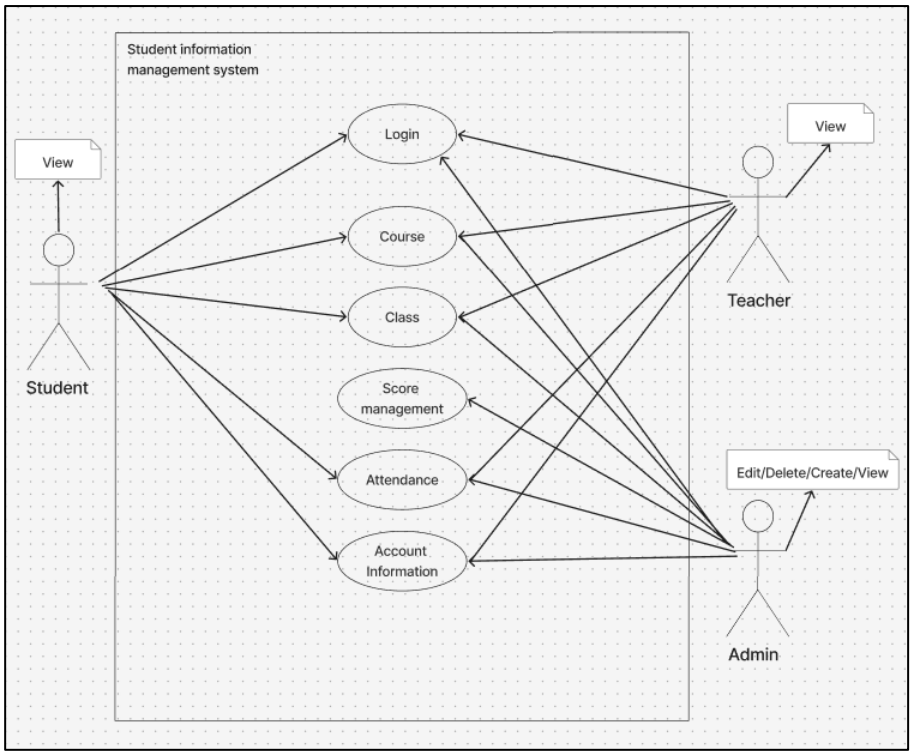


Figure 5.1 Use case

❖ **Class Diagram**

Based on a detailed analysis of the system requirements and specifications, we proceeded to the design stage by constructing a class diagram for the Student Information Management System (SIMS). This diagram provides a structured and organized representation of the system's architecture, clearly outlining the key classes, their attributes, methods, and the relationships that connect them. Creating the class diagram is an essential step, as it serves as a comprehensive blueprint that reveals how different system components interact, how data flows between them, and how each class contributes to the overall functionality. This visual model helps ensure consistency, maintainability, and adherence to object-oriented principles throughout the system's development

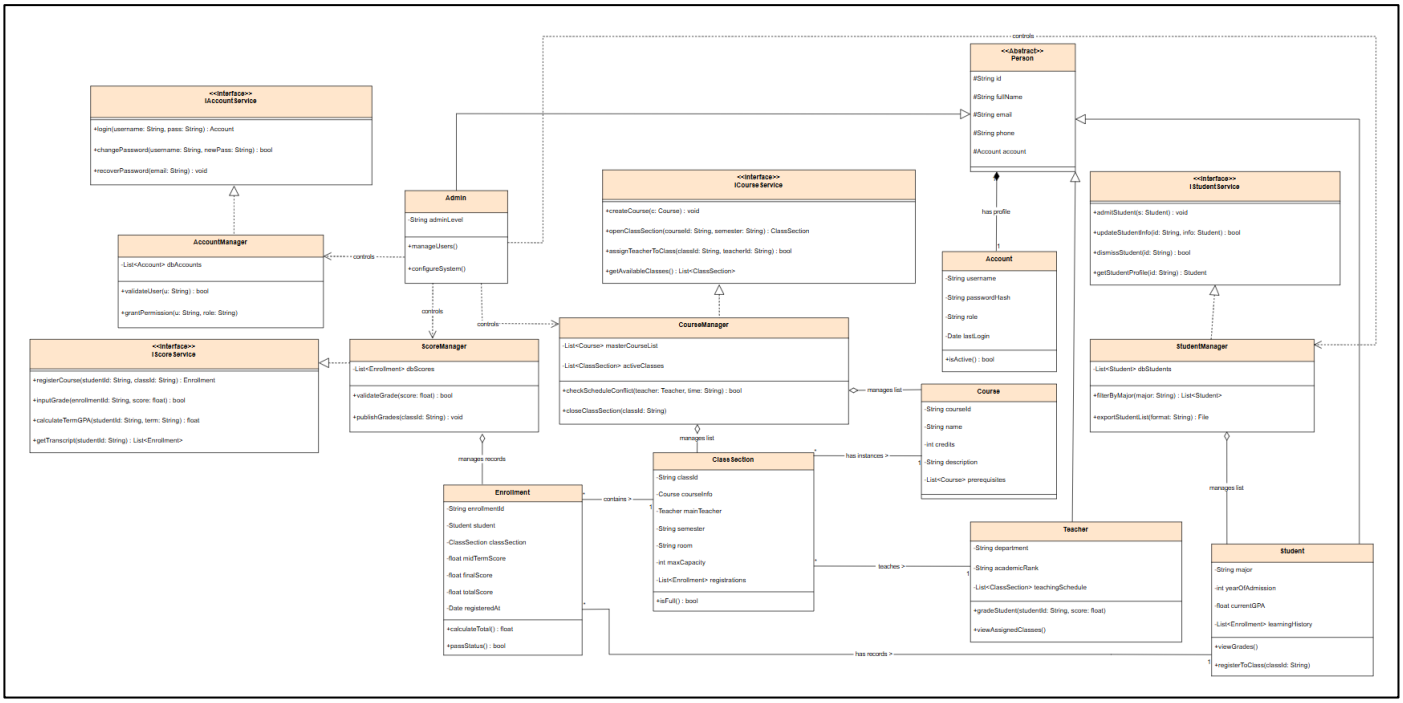


Figure 5.2 Class diagram

5.2.Building applications

❖ **Applying the MVC Model:** After the analysis and design phase, we proceeded to build the web application using ASP.NET Core MVC. The solution is structured into Areas (Admin, Instructor, Student) to ensure separation of concerns and maintainability

❖ **Login Functionality**

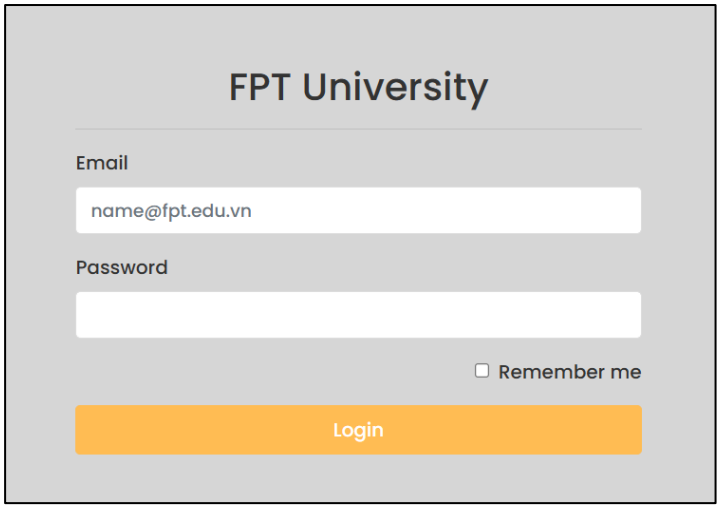
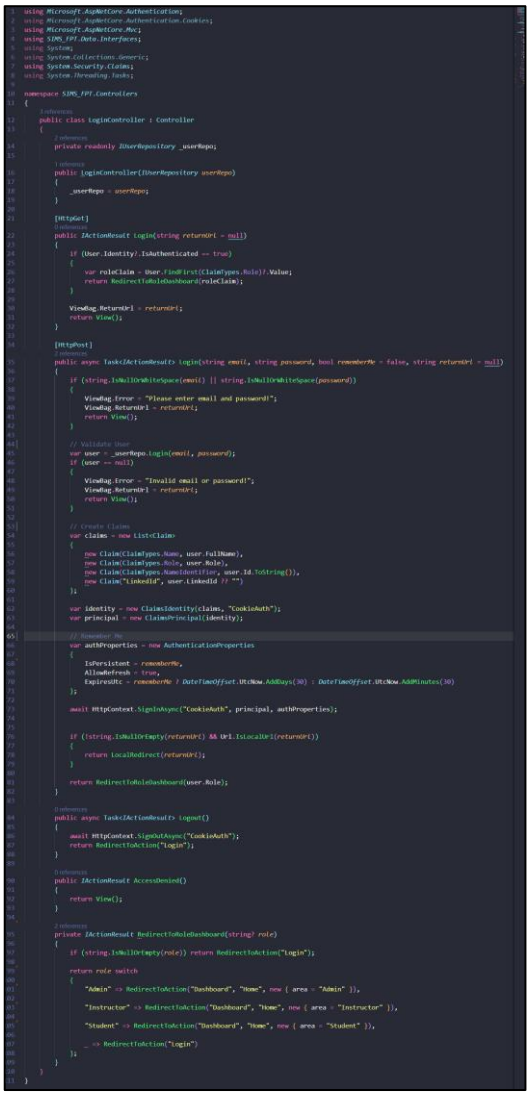


Figure 5.3 *Login page 1*



```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Mvc;
using SIMS_FPT.Data.Interfaces;
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;

namespace SIMS_FPT.Controllers
{
    1 reference
    public class LoginController : Controller
    {
        1 reference
        private readonly IUserRepository _userRepo;

        1 reference
        public LoginController(IUserRepository userRepo)
        {
            _userRepo = userRepo;
        }

        [HttpGet]
        0 references
        public IActionResult Login(string returnUrl = null)
        {
            if (User.Identity?.IsAuthenticated == true)
            {
                var roleClaim = User.FindFirst(ClaimTypes.Role)?.Value;
                return RedirectToRoleDashboard(roleClaim);
            }

            ViewBag.ReturnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        2 references
        public async Task<IActionResult> Login(string email, string password, bool rememberMe = false, string returnUrl = null)
        {
            if (string.IsNullOrWhiteSpace(email) || string.IsNullOrWhiteSpace(password))
            {
                ViewBag.Error = "Please enter email and password!";
                ViewBag.ReturnUrl = returnUrl;
                return View();
            }

            // Validate User
            var user = _userRepo.Login(email, password);
            if (user == null)
            {
                ViewBag.Error = "Invalid email or password!";
                ViewBag.ReturnUrl = returnUrl;
                return View();
            }

            // Create Claims
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.Name, user.FullName),
                new Claim(ClaimTypes.Role, user.Role),
                new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                new Claim("LinkedId", user.LinkedId ?? "")
            };

            var identity = new ClaimsIdentity(claims, "CookieAuth");
            var principal = new ClaimsPrincipal(identity);

            // Remember Me
            var authProperties = new AuthenticationProperties
            {
                IsPersistent = rememberMe,
                AllowRefresh = true,
                ExpiresUtc = rememberMe ? DateTimeOffset.UtcNow.AddDays(30) : DateTimeOffset.UtcNow.AddMinutes(30)
            };

            await HttpContext.SignInAsync("CookieAuth", principal, authProperties);


            if (!string.IsNullOrEmpty(returnUrl) && Url.IsLocalUrl(returnUrl))
            {
                return LocalRedirect(returnUrl);
            }

            return RedirectToRoleDashboard(user.Role);
        }

        0 references
        public async Task<IActionResult> Logout()
        {
            await HttpContext.SignOutAsync("CookieAuth");
            return RedirectToAction("Login");
        }

        0 references
        public IActionResult AccessDenied()
        {
            return View();
        }

        3 references
        private IActionResult RedirectToRoleDashboard(string? role)
        {
            if (string.IsNullOrEmpty(role)) return RedirectToAction("Login");

            return role switch
            {
                "Admin" => RedirectToAction("Dashboard", "Home", new { area = "Admin" }),

                "Instructor" => RedirectToAction("Dashboard", "Home", new { area = "Instructor" }),

                "Student" => RedirectToAction("Dashboard", "Home", new { area = "Student" }),

                _ => RedirectToAction("Login")
            };
        }
    }
}
```

Figure 5.4 Login Controller Code

❖ **Functions of the page**

- Authentication: Verifies credentials against the database using PasswordHasherHelper

- Role Routing: Upon successful login, the system detects the user's role (Admin, Teacher, or Student) and redirects them to their specific Dashboard Area

- Session Management: Stores user identity in HttpContext.Session to maintain login state

- **Admin Dashboard**

```
using CsvHelper;
using CsvHelper.Configuration;
using Microsoft.AspNetCore.Mvc;
using SIMS_FPT.Models;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SIMS_FPT.Areas.Admin.Controllers
{
    [Authorize(Roles = "Admin")]
    [Area("Admin")]
    0 references
    public class HomeController : Controller
    {
        1 reference
        private readonly string _studentPath = Path.Combine(Directory.GetCurrentDirectory(), "CSV_DATA", "students.csv");
        1 reference
        private readonly string _deptPath = Path.Combine(Directory.GetCurrentDirectory(), "CSV_DATA", "departments.csv");
        1 reference
        private readonly string _teacherPath = Path.Combine(Directory.GetCurrentDirectory(), "CSV_DATA", "teachers.csv");

        0 references
        public IActionResult Dashboard()
        {
            var model = new DashboardViewModel();

            var students = ReadCsv<StudentCSVModel>(_studentPath);
            var departments = ReadCsv<DepartmentModel>(_deptPath);
            var teachers = ReadCsv<TeacherCSVModel>(_teacherPath);

            model.StudentCount = students.Count;
            model.DepartmentCount = departments.Count;
            model.TeacherCount = teachers.Count;
            model.TotalRevenue = 0;

            model.RevenueLabels = new List<string> { "2022", "2023", "2024" };
            model.RevenueData = new List<decimal> { 0, 0, 0 };

            // Student Gender Chart
            var genderStat = students
                .GroupBy(s => s.Gender ?? "Unknown")
                .Select(g => new { g.Key, Count = g.Count() })
                .ToList();

            model.StudentClassLabels = genderStat.Select(x => x.Key).ToList();
            model.StudentClassData = genderStat.Select(x => x.Count).ToList();

            // Newest Students
            model.NewestStudents = students
                .OrderByDescending(s => s.StudentId)
                .Take(5)
                .ToList();

            return View(model);
        }

        3 references
        private List<T> ReadCsv<T>(string filePath)
        {
            if (!System.IO.File.Exists(filePath))
                return new List<T>();

            var config = new CsvConfiguration(CultureInfo.InvariantCulture)
            {
                HasHeaderRecord = true,
                MissingFieldFound = null,
                HeaderValidated = null,
                BadDataFound = null
            };

            using var reader = new StreamReader(filePath);
            using var csv = new CsvReader(reader, config);

            return csv.GetRecords<T>().ToList();
        }
    }
}
```

Figure 5.5: Admin Controller Code

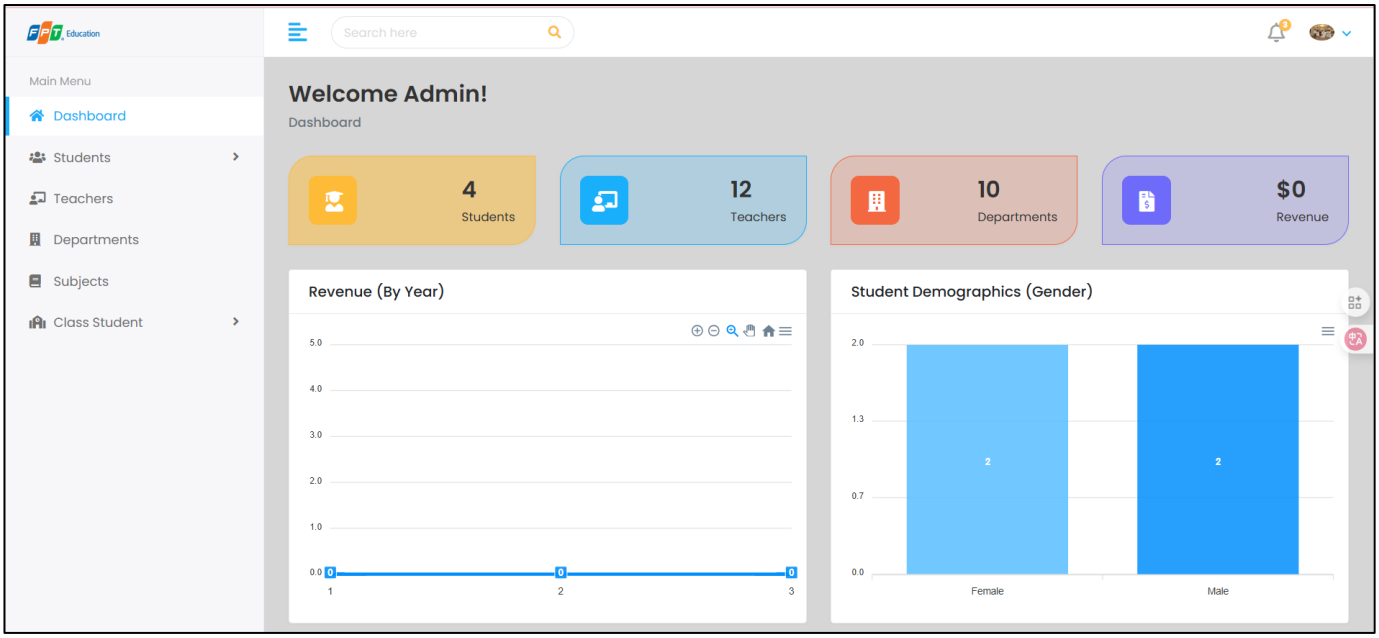


Figure 5.6: Admin dashboard

Functions of the page

- Statistical Overview: Displays cards showing the total count of Students, Teachers, and Departments

- Navigation: Provides the main gateway to manage different entities like Departments, Subjects, and Classes

❖ **Student Information Management**

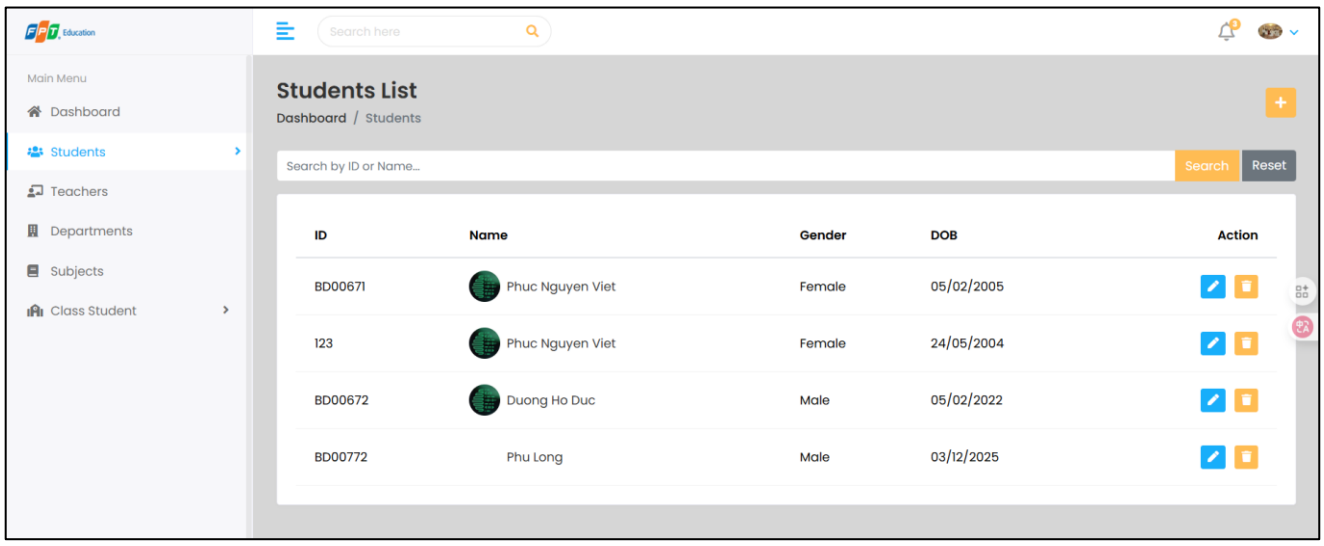


Figure 5.7: Student List Page

Figure 5.8: Student Controller Code

- CRUD Operations: Allows Admins to view details, edit profiles, or delete students

- CSV Import This is a critical function for handling large datasets. Instead of adding students one by one, the Admin can upload a .csv file containing hundreds of student records. The StudentService parses this file and bulk-inserts data into the database

❖ Department and Subject Management

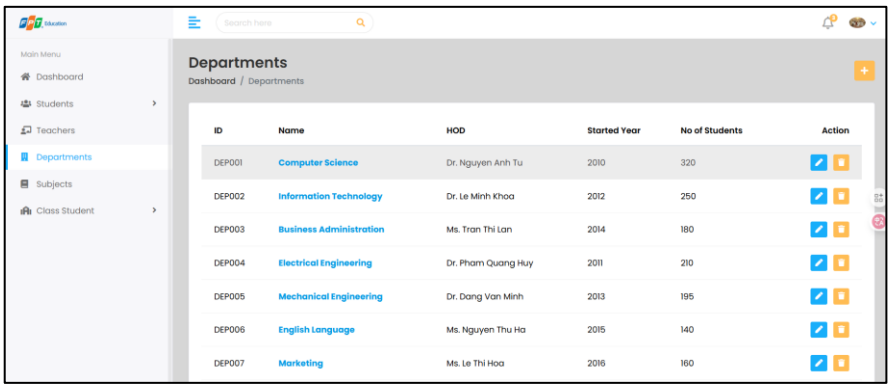


Figure 5.9: Department Management Page

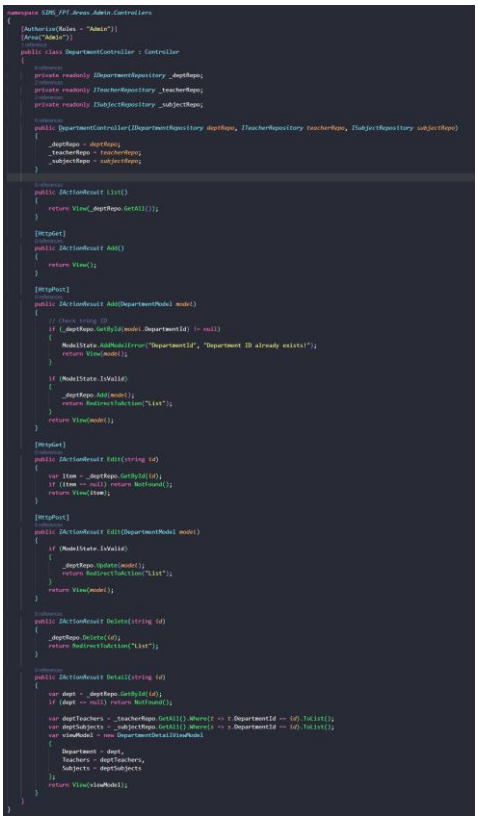


Figure 5.10: Department Controller

- Structure Organization: Allows the creation of University Departments (e.g., IT, Business)

- Subject Allocation: Admins create subjects (e.g., C# Programming, Database) and assign them to specific departments

❖ Class and Enrollment Management

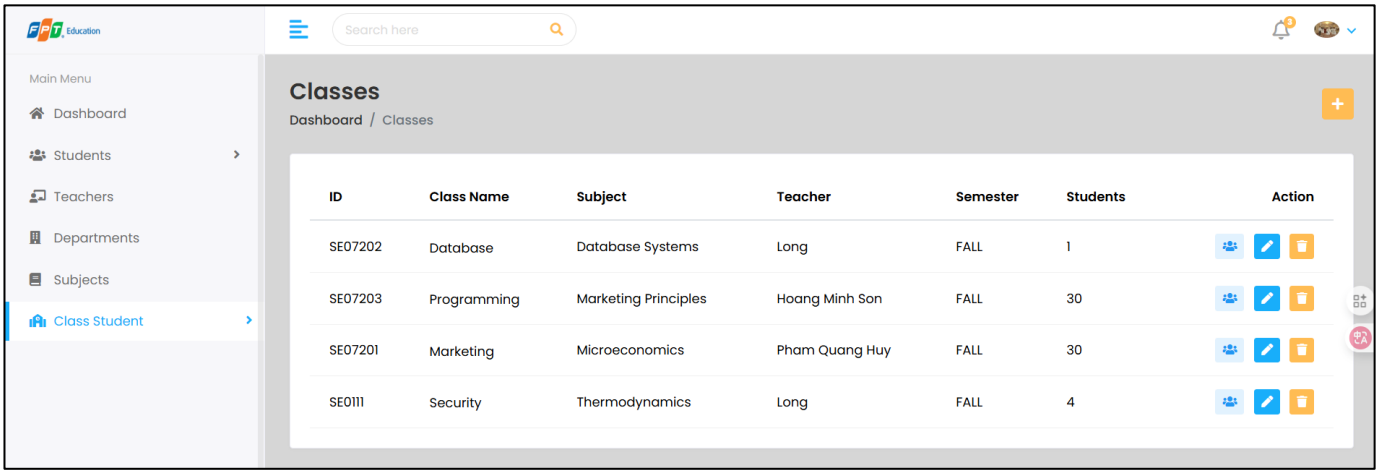


Figure 5.11: Class Management Page

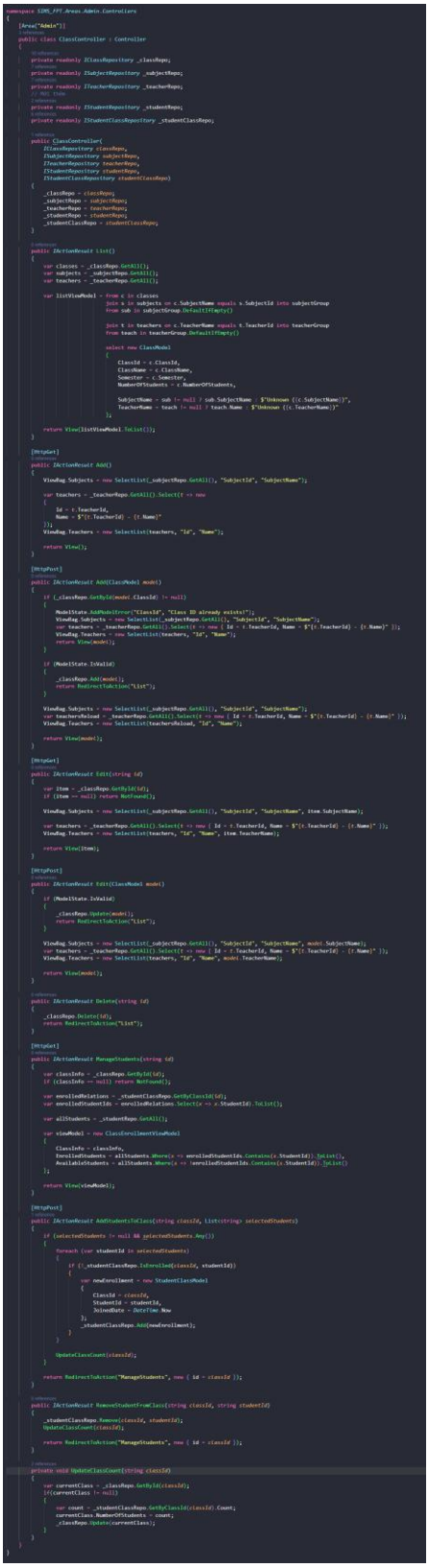


Figure 5.12: Class Controller

- Class Scheduling: Admins define classes for a specific subject and assign a teacher (Instructor)

- Student Enrollment: The system manages the Many-to-Many relationship between Students and Classes (via StudentClass table)

❖ Assignment and Grading

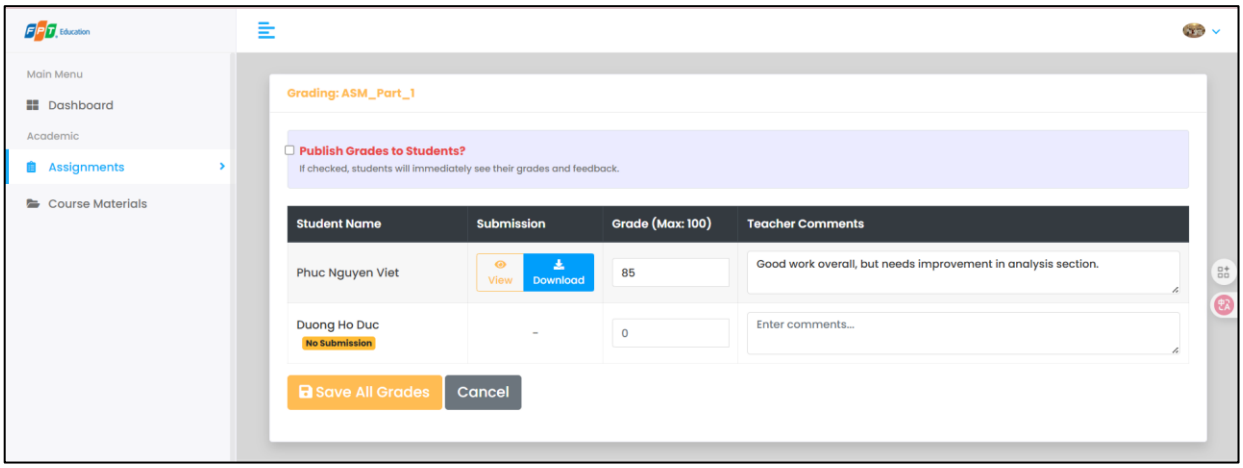


Figure 5.13: Assignment Grading Page

```
using SIMS_FPT.Data.Interfaces;
using SIMS_FPT.Business.Interfaces;
using SIMS_FPT.Models;
using SIMS_FPT.Models.ViewModels;
using System.Collections.Generic;
using System.Linq;

namespace SIMS_FPT.Business.Services
{
    4 references
    public class GradingService : IGradingService
    {
        4 references
        private readonly IAssignmentRepository _assignmentRepo;
        4 references
        private readonly ISubmissionRepository _submissionRepo;
        2 references
        private readonly IStudentRepository _studentRepo;
        2 references
        private readonly IStudentClassRepository _studentClassRepo; // [1] Add this

        // [2] Update Constructor
        1 reference
        public GradingService(IAssignmentRepository assignmentRepo,
                              ISubmissionRepository submissionRepo,
                              IStudentRepository studentRepo,
                              IStudentClassRepository studentClassRepo)
        {
            _assignmentRepo = assignmentRepo;
            _submissionRepo = submissionRepo;
            _studentRepo = studentRepo;
            _studentClassRepo = studentClassRepo;
        }

        3 references
        public void ProcessGrades(BulkGradeViewModel model)
        {
            // 1. Update Publishing Status
            var assignment = _assignmentRepo.GetById(model.AssignmentId);
            if (assignment != null)
            {
                assignment.AreGradesPublished = model.IsPublished;
                _assignmentRepo.Update(assignment);
            }

```

```
            // 2. Process Grades
            foreach (var item in model.StudentGrades)
            {
                var submission = _submissionRepo.GetByStudentAndAssignment(item.StudentId, model.AssignmentId)
                                 ?? new SubmissionModel { StudentId = item.StudentId, AssignmentId = model.AssignmentId };

                submission.Grade = item.Grade;
                submission.TeacherComments = item.Feedback;

                _submissionRepo.SaveSubmission(submission);
            }
        }

        2 references
        public BulkGradeViewModel PrepareGradingView(string assignmentId, string currentTeacherId)
        {
            var assignment = _assignmentRepo.GetById(assignmentId);
            if (assignment == null) return null;

            if (assignment.TeacherId != currentTeacherId) return null;

            // [3] FIX: Get students from the specific CLASS, not the whole subject
            // Old incorrect line: var students = _studentRepo.GetBySubject(assignment.SubjectId);

            var enrollments = _studentClassRepo.GetByClassId(assignment.ClassId);
            var submissions = _submissionRepo.GetByAssignmentId(assignmentId);

            var model = new BulkGradeViewModel
            {
                AssignmentId = assignment.AssignmentId,
                AssignmentTitle = assignment.Title,
                MaxPoints = assignment.MaxPoints,
                IsPublished = assignment.AreGradesPublished,
                StudentGrades = new List<StudentGradeItem>()
            };

            // [4] Iterate through the enrolled students
            foreach (var enrollment in enrollments)
            {
                var studentInfo = _studentRepo.GetById(enrollment.StudentId);
                if (studentInfo == null) continue; // Skip if student not found in main DB

                var sub = submissions.FirstOrDefault(x => x.StudentId == enrollment.StudentId);

                model.StudentGrades.Add(new StudentGradeItem
                {
                    StudentId = studentInfo.StudentId,
                    StudentName = studentInfo.FullName,
                    HasSubmitted = sub != null && !string.IsNullOrEmpty(sub.FilePath),
                    SubmissionFilePath = sub?.FilePath,
                    Grade = sub?.Grade,
                    Feedback = sub?.TeacherComments
                });
            }

            return model;
        }
    }
}
```

Figure 5.14: Grading Service Code

Functions of the page

- Material Distribution: Instructors upload course materials (PDFs) for students

- Submission Tracking: Shows a list of students who have submitted their assignments

- Grading System: Instructors verify student work and input grades. The system updates the Submission records in the database

5.3.How the application helps in handling large data sets

One of the core requirements of the P5 unit is demonstrating the ability to handle large datasets effectively. The SIMS (Student Information Management System) addresses this through specific implementation strategies

- Bulk Data Import via CSV
  - Problem: Manually entering data for thousands of new students each semester is time-consuming and prone to human error
  - The system implements a StudentCSVModel and StudentService. The UploadCSV function allows the administrator to upload a single file containing large volumes of student data. The application parses this data using StreamReader and processes it in a loop to validate and insert records into the SQL database efficiently
  - Benefit: Reduces data entry time from days to seconds
- Database Optimization (Entity Framework Core)
  - The application uses Entity Framework Core to interact with the database. This allows for optimized SQL queries
  - Relationships (One-to-Many, Many-to-Many) between Students, Classes, and Submissions are handled using Foreign Keys, ensuring data integrity without duplicating large amounts of text data
- Separation of Data (Areas)
  - By dividing the application into Areas (Admin, Instructor, Student), the system ensures that queries are relevant to the specific user role. For example, a Student only queries their own grades and classes, rather than loading the entire university's dataset, reducing the load on the server during peak times
  - The SISM application not only meets the functional requirements of managing school operations but also demonstrates technical proficiency in handling data scaling. By implementing bulk import features and a structured MVC architecture, the system is prepared to handle the large-scale data needs of a growing educational institution

6.Examine the different methods of implementing automatic testing as designed in the test plan (P6)

6.1.Automated testing methods

- Automated Testing is not designed to completely replace human testers. Instead, its core purpose is to liberate humans from repetitive tasks and enhance the reliability of the system. Specifically, its objectives include
- Optimizing Regression Testing: The biggest goal is to ensure that "legacy features still function correctly when new code is introduced." Automation allows for the re-execution of thousands of old test cases whenever a new build is deployed without requiring extra human effort, ensuring system integrity
- Shortening the Feedback Cycle: In Agile/DevOps environments, the purpose is to let developers know immediately if their code has caused any breaks. Instead of waiting 2-3 days for the QA team to finish manual testing, Automation returns results in minutes or hours
- Expanding Test Coverage: There are scenarios that humans cannot physically perform or find extremely difficult to execute (e.g., simulating 10,000 concurrent users to test performance - Load Testing). The purpose of automation is to fill the gaps that manual testing cannot reach
- Increasing Long-term Economic Efficiency (ROI): Although the initial investment cost is high, the long-term goal is to reduce operational costs. A script written once can be used indefinitely, reducing the need to hire a large workforce solely for repetitive mouse-clicking tasks

❖ **The Role of Test Methods in a Test Plan**

In a Test Plan, if the Plan is the "Strategy," then the Test Methods (Unit, Integration, System, Acceptance, etc.) act as the "Tactics." Their specific roles within the Test Plan are as follows

- **Realizing Quality Goals:** Each test method in the Test Plan acts as a defect filter at a different layer
  - ✓ *Unit Test:* Ensures every line of logic is correct (Foundational role)
  - ✓ *Integration Test:* Ensures modules communicate correctly (Connectivity role)
  - ✓ *UI Test:* Ensures the user experience is correct (Surface role). Specifying methods in the Test Plan ensures that no layer of the software is overlooked

- **Defining Resources and Tools (Resource Allocation):** The selection of test methods in the Test Plan dictates *who* and *what* the project team needs

- ✓ If the Test Plan selects Automation Testing, it requires personnel who can code (Java, Python...) and tools like Selenium
- ✓ If the Test Plan emphasizes Manual/Exploratory Testing, it requires personnel with deep business domain knowledge

- **Risk Management:** Test methods play the role of mitigating specific risks outlined in the Test Plan. For example, to mitigate security risks, the Test Plan will designate Security Testing. To mitigate system overload risks, Performance Testing will be applied

- **Risk Management:** Test methods play the role of mitigating specific risks outlined in the Test Plan. For example, to mitigate security risks, the Test Plan will designate Security Testing. To mitigate system overload risks, Performance Testing will be applied

❖ **Designed Test Types (Testing Levels)**

In an automated testing strategy, tests are typically categorized into three main layers based on scope and granularity, often referred to as the Testing Pyramid

- **Unit Testing**

✓ The lowest level of testing, focusing on verifying the smallest components of the source code (such as functions, methods, or classes) in isolation

✓ Characteristics: Executed by Developers. Extremely fast execution. Does not involve external databases or networks (often uses Mocks/Stubs)

- **Integration Testing**

✓ Verifies the interaction and communication between different components or modules when they are combined

✓ Characteristics: Checks interfaces, APIs, and database connections. Ensures that data passed from Module A to Module B is not corrupted or lost

✓ Example: Verifying that when the "Register" API is called, the user data is correctly persisted in the User table in the Database

- **End-to-End (E2E) Testing**

✓ Tests the entire application flow from start to finish from the perspective of a real end-user

- ✓ Characteristics: Simulates user behavior (clicks, typing) on the actual UI. It is the slowest to execute and has the highest maintenance cost but reflects reality most accurately
- ✓ Example: Open browser -> Go to Login Page -> Enter credentials -> Click Login -> Verify successful redirection to the Homepage

❖ **Objectives of Each Type in the Test Plan**

Each test type is included in the Test Plan to address specific risks at different stages of development

- **Objective of Unit Testing in the Test Plan**

✓ Primary Goal: "Detect bugs as early as possible" (Shift Left Testing)

✓ Strategic Role: Acts as the first "filter." The Test Plan typically mandates high Unit Test coverage (e.g., >80%) to ensure "clean code" before it is handed over to the QA team

✓ Benefit: Maximizes cost efficiency (fixing a bug at the coding stage is 10-100 times cheaper than fixing it in Production)

- **Objective of Unit Testing in the Test Plan**

✓ Primary Goal: "Ensure the integrity of the system structure."

✓ Strategic Role: Detects errors related to protocols, data formats, and logic flow between services. This is the most critical layer for Microservices architectures

✓ Benefit: Prevents "Works on my machine" issues (where code works locally but fails when components are connected)

- **Objective of E2E Testing in the Test Plan**

✓ Primary Goal: "Business Validation and Release Confidence."

✓ Strategic Role: Acts as the final gatekeeper. The Test Plan uses E2E to answer the question: "Can the user complete their tasks (e.g., purchasing, transferring money) without obstacles?"

✓ Benefit: Ensures the system operates as expected in a Real-world environment, accounting for network latency, browser differences, and server configurations

❖ **Comparative Analysis of Automated Testing Levels**

Criteria	Unit Tests	Integration Tests	End-to-End (E2E) Tests
Scope and Granularity	Validates discrete units of code (functions, methods, classes) in isolation	Validates interfaces, data flow, and interactions between coupled components	Validates the complete system workflow and user experience from start to finish
Execution Performance	High (Instant feedback loop; typically milliseconds)	Moderate (Slower due to I/O operations and DB connections)	Low (High latency due to UI rendering and network overhead)
Stability (Reliability)	Deterministic (Highly consistent results; low noise)	Variable (Dependent on environment stability)	Non-deterministic (Prone to "flakiness" and false negatives)

Fault Isolation	Precise (Pinpoints the exact line of code causing the failure)	Regional (Identifies the failing subsystem or interface)	Opaque (Requires extensive investigation to find the root cause)
Maintenance Effort	Minimal (Resilient to refactoring)	Moderate	Significant (Highly sensitive to UI/DOM changes).

Table 6.1 Technical comparison between types of automatic testing

❖ **Technical comparison between types of automatic testing**

This section analyzes the strategic value and potential drawbacks of each testing methodology within a Test Plan

❖ **Unit Testing: The Foundation**

- **Advantages**
  - ✓ Shift-Left Quality: Facilitates early bug detection during the implementation phase, drastically reducing the cost of remediation.
  - ✓ Dependency Independence: Utilizes mocks and stubs to verify logic without relying on unstable external systems or networks.
  - ✓ Rapid Feedback: The speed of execution encourages developers to run tests frequently, promoting a Test-Driven Development (TDD) culture.
- **Disadvantages**
  - ✓ Integration Blind Spots: Fails to detect errors arising from component interconnectivity (e.g., API contract mismatches).
  - ✓ Limited Context: A passing unit test confirms logical correctness but does not guarantee that the feature delivers business value

❖ **Integration Testing: The Connector**

- **Advantages**
  - ✓ Interface Integrity: Ensures that data contracts between modules (e.g., Application Logic ↔ Database, Microservice A ↔ Microservice B) are upheld.
  - ✓ Holistic Subsystem Verification: Validates scenarios that span multiple layers of the application stack, catching issues unit tests miss.
- **Disadvantages**
  - ✓ Environment Dependency: Requires a pre-configured environment (e.g., Docker containers, Test Databases), increasing setup complexity.
  - ✓ Cascading Failures: A failure in a core dependency (such as a database outage) can cause widespread test failures, complicating debugging

❖ **End-to-End (E2E) Testing: The Validation**

- **Advantages**

- ✓ Real-World Simulation: Mimics actual user behavior, ensuring the integrity of critical business workflows (Critical Paths).
- ✓ Highest Confidence Level: Serves as the final "sign-off" before production deployment, validating the system in a production-like environment.

- **Disadvantages**
  - ✓ Execution Bottleneck: Due to slow execution speeds, comprehensive E2E suites can significantly delay the CI/CD pipeline.
  - ✓ High Fragility: Tests are brittle and easily broken by minor UI changes (e.g., changing a button's ID or CSS class), requiring constant maintenance

6.2.SIMS System Test Plan

	Test Case Name	Pre-conditions	Expected Results	Start Date	End Date
TC01	Verify Account Authentication (Login)	User account (Admin/Teacher/Student) exists in the database	- System logs in successfully and redirects to the dashboard with valid credentials. - System displays "Login failed" error with invalid credentials	2025-12-11	2025-12-11
TC02	Add New Student with Data Validation (Email/Phone)	Admin is logged into the system	The system rejects the save operation and displays specific error messages for the invalid email or phone number to ensure data integrity	2025-12-11	2025-12-12
TC03	Delete Course by ID (Course Management)	The course to be deleted currently exists in the system.	The specified course is completely removed from the system and is no longer retrievable in the course list	2025-12-11	2025-12-12

TC04	Course Enrollment Verification	Student and Course records exist independently	The student is successfully linked to the course, and the course is listed in the student's profile (Many-to-Many relationship established)	2025-12-12	2025-12-12
TC05	Grade Calculation Strategy Switching	Student has raw academic scores recorded	- The default strategy returns a letter grade (e.g., A/B/C). - The alternative strategy returns a status (e.g., Pass/Fail). The system correctly applies the selected algorithm without errors	2025-12-12	2025-12-13
TC06	Delete Student and Cascading Data Integrity	Student exists with associated academic records	The student profile is deleted, and all inseparable associated records (e.g., transcripts) are automatically destroyed (Composition integrity verified)	2025-12-12	2025-12-13
TC07	User Role Authorization Check	User is logged in with restricted privileges (e.g., Student role)	The system denies access, hides the administrative options, or returns an "Action Not Permitted" error, ensuring role segregation	2025-12-13	2025-12-13

Table 6.2 Test Plan

7.Implement automatic testing of the developed application (P7)

7.1.Comprehensive Test Case Implement

❖ **TC01: Verify Account Authentication (Login)**

```
// TC01: Verify Account Authentication (Invalid Login)
[Test]
0 references
public async Task TC01_Login_WithInvalidCredentials_ReturnsViewWithError()
{
    // Arrange
    string email = "wrong@test.com";
    string password = "wrong";
    _mockUserRepo.Setup(r => r.Login(email, password)).Returns((Users)null);


    var result = await _controller.Login(email, password) as ViewResult;

    // Assert
    Assert.That(result, Is.Not.Null, "Result should be a ViewResult");
    Assert.That(result.ViewData["Error"], Is.EqualTo("Invalid email or password!"));
}
```

Figure 7.1 TC01

- Objective: Ensure the system correctly identifies invalid credentials and prevents unauthorized access.
  - ✓ Code Reference: LoginController.cs.

❖ **TC02: Add New Student with Data Validation**

```
// TC02: Add New Student with Data Validation
[Test]
0 references
public async Task TC02_Add_WithInvalidModel_ReturnsViewAndDoesNotCallService()
{
    // Arrange
    var invalidModel = new StudentCSVModel { StudentId = "ST001", Email = "invalid-email" };

    // Simulate Validation Error
    _controller.ModelState.AddModelError("Email", "Invalid email format");

    // Act
    var result = await _controller.Add(invalidModel) as ViewResult;

    // Assert
    Assert.That(result, Is.Not.Null);
    Assert.That(result.Model, Is.InstanceOf<StudentCSVModel>());
    Assert.That(_controller.ModelState.IsValid, Is.False);
}
```

Figure 7.2 TC02

Objective: Verify that invalid data (e.g., invalid email format) is rejected to maintain data integrity.

Code Reference: StudentController.cs (Area: Admin).

- **Execution Logic:**
  - ✓ User submits a form with an invalid email.
  - ✓ ASP.NET Core validation runs; ModelState.IsValid becomes false.

- ✓ The Add method checks if (ModelState.IsValid).
- ✓ Since it is false, the code skips _service.Add(model) and returns View(model).
- ✓ Outcome: The data is not saved to the database, and the form is redisplayed with errors.

❖ **TC03: Delete Course by ID**

```
// TC03: Delete Course by ID
[Test]
0 references
public void TC03_DeleteSubject_CallsRepositoryDelete()
{
    // Arrange
    string subjectId = "SUB001";

    // Act
    var result = _controller.Delete(subjectId) as RedirectToActionResult;

    // Assert
    Assert.That(result.ActionName, Is.EqualTo("List"));
    _mockRepo.Verify(r => r.Delete(subjectId), Times.Once);
}
```

Figure 7.3 *TC03*

Objective: Ensure a course (Subject) can be permanently removed from the system.

Code Reference: SubjectController.cs (Area: Admin).

Execution Logic:

- ✓ Admin clicks "Delete" for a specific Subject ID.
- ✓ The Delete(string id) action is called.
- ✓ It executes _repo.Delete(id).
- ✓ It redirects the user back to the "List" view.
- ✓ Outcome: The subject is removed from the repository.

❖ **TC04: Course Enrollment Verification**

```
// TC04: Course Enrollment Verification
[Test]
0 references
public void TC04_AddStudentsToClass_CreatesEnrollmentRecords()
{
    // Arrange
    string classId = "SE1601";
    var selectedStudents = new List<string> { "ST01", "ST02" };

    // Setup mocks
    _studentClassRepo.Setup(r => r.IsEnrolled(classId, It.IsAny<string>())).Returns(false);
    _classRepo.Setup(r => r.GetById(classId)).Returns(new ClassModel { ClassId = classId });
    _studentClassRepo.Setup(r => r.GetByClassId(classId)).Returns(new List<StudentClassModel>());

    // Act
    var result = _controller.AddStudentsToClass(classId, selectedStudents) as RedirectToActionResult;

    // Assert
    Assert.That(result.ActionName, Is.EqualTo("ManageStudents"));

    _studentClassRepo.Verify(r => r.Add(It.Is<StudentClassModel>(s => s.ClassId == classId && s.StudentId == "ST01")), Times.Once);
    _studentClassRepo.Verify(r => r.Add(It.Is<StudentClassModel>(s => s.ClassId == classId && s.StudentId == "ST02")), Times.Once);
}
```

Figure 7.4 TC04

Objective: Verify that a student can be successfully linked to a class.

Code Reference: ClassController.cs (Area: Admin).

Execution Logic:

- ✓ Admin selects students and submits the AddStudentsToClass form.
- ✓ The controller iterates through selectedStudents.
- ✓ It checks !_studentClassRepo.IsEnrolled to prevent duplicates.
- ✓ It calls _studentClassRepo.Add(newEnrollment) to save the link.
- ✓ Outcome: A new record is created in student_classes.csv, linking the Student ID to the Class ID.

❖ **TC05: Grade Calculation Strategy (Processing)**

```
// TC05: Grade Calculation (Processing/Saving)
[Test]
0 references
public void TC05_ProcessGrades_UpdatesSubmissionRepository()
{
    // Arrange
    var model = new BulkGradeViewModel
    {
        AssignmentId = "ASM01",
        IsPublished = true,
        StudentGrades = new List<StudentGradeItem>
        {
            new StudentGradeItem { StudentId = "ST01", Grade = 8.5, Feedback = "Good job" }
        }
    };

    _assignRepo.Setup(r => r.GetById("ASM01")).Returns(new AssignmentModel());
    _subRepo.Setup(r => r.GetByStudentAndAssignment("ST01", "ASM01"))
            .Returns(new SubmissionModel { StudentId = "ST01", AssignmentId = "ASM01" });

    // Act
    _service.ProcessGrades(model);

    // Assert
    // Verify SaveSubmission was called with grade 8.5
    _subRepo.Verify(r => r.SaveSubmission(It.Is<SubmissionModel>(
        s => s.StudentId == "ST01" &&
             s.Grade == 8.5 &&
             s.TeacherComments == "Good job"
    )), Times.Once);

    _assignRepo.Verify(r => r.Update(It.Is<AssignmentModel>(a => a.AreGradesPublished == true)), Times.Once);
}
```

Figure 7.5 *TC05*

Objective: Ensure grades and feedback entered by the teacher are saved correctly.

Code Reference: GradingService.cs.

Execution Logic:

- ✓ Teacher submits the BulkGradeViewModel.
- ✓ GradingService.ProcessGrades is called.
- ✓ It updates the assignment's "Published" status.
- ✓ It iterates through students, updates their Grade and TeacherComments, and calls _submissionRepo.SaveSubmission.
- ✓ Outcome: The student's submission record is updated with the new numeric grade and feedback.

❖ TC06: Delete Student and Cascading Data

```
// TC06: Delete Student and Cascading Data Integrity
[Test]
0 references
public void TC06_DeleteStudent_CallsRepositoryDelete()
{
    // Arrange
    string studentId = "ST999";

    // Act
    var result = _controller.DeleteStudent(studentId) as RedirectToActionResult;

    // Assert
    Assert.That(result.ActionName, Is.EqualTo("List"));
    _mockRepo.Verify(r => r.Delete(studentId), Times.Once, "Repository Delete should be called exactly once");
}
```

Objective: Verify that deleting a student removes their core profile.

Code Reference: StudentController.cs (Area: Admin).

Execution Logic:

- ✓ Admin triggers DeleteStudent(id).
- ✓ The controller calls _repo.Delete(id).
- ✓ Note: In a file-based system, true "cascading" (deleting grades/enrollments) depends on the Repository implementation, but the Controller action successfully executes the delete command for the main profile.
- ✓ Outcome: The student is removed from the main student list.

❖ **TC07: User Role Authorization Check**

```
// TC07: User Role Authorization Check (Redirect Logic)
[Test]
0 references
public async Task TC07_Login_AsStudent_RedirectsToStudentDashboard()
{
    // Arrange
    string email = "student@test.com";
    string password = "123";
    // Important: HashAlgorithm must match logic in Repo or be handled.
    // Since we mocked Repo.Login to return this USER directly, the internal password check in Repo is bypassed.
    var studentUser = new Users { Email = email, Role = "Student", FullName = "Test Student" };

    _mockUserRepo.Setup(r => r.Login(email, password)).Returns(studentUser);

    // Act
    var result = await _controller.Login(email, password) as RedirectToActionResult;

    // Assert
    Assert.That(result, Is.Not.Null, "Result should be a RedirectToActionResult");
    Assert.That(result.ActionName, Is.EqualTo("Dashboard"));
    Assert.That(result.ControllerName, Is.EqualTo("Home"));
    Assert.That(result.RouteValues["area"], Is.EqualTo("Student"), "Student should be redirected to Student Area");
}
```

Objective: Verify that users are redirected to the correct dashboard based on their role (Security).

Code Reference: LoginController.cs.

Execution Logic:

- User logs in with valid credentials.
- The system identifies the role (e.g., "Student").
- The RedirectToRoleDashboard switch statement executes.
- If role is "Student", it redirects to Area = "Student", Controller = Home, Action = Dashboard.

Outcome: The user lands on the correct, role-restricted dashboard.

❖ Test Results Summary Table

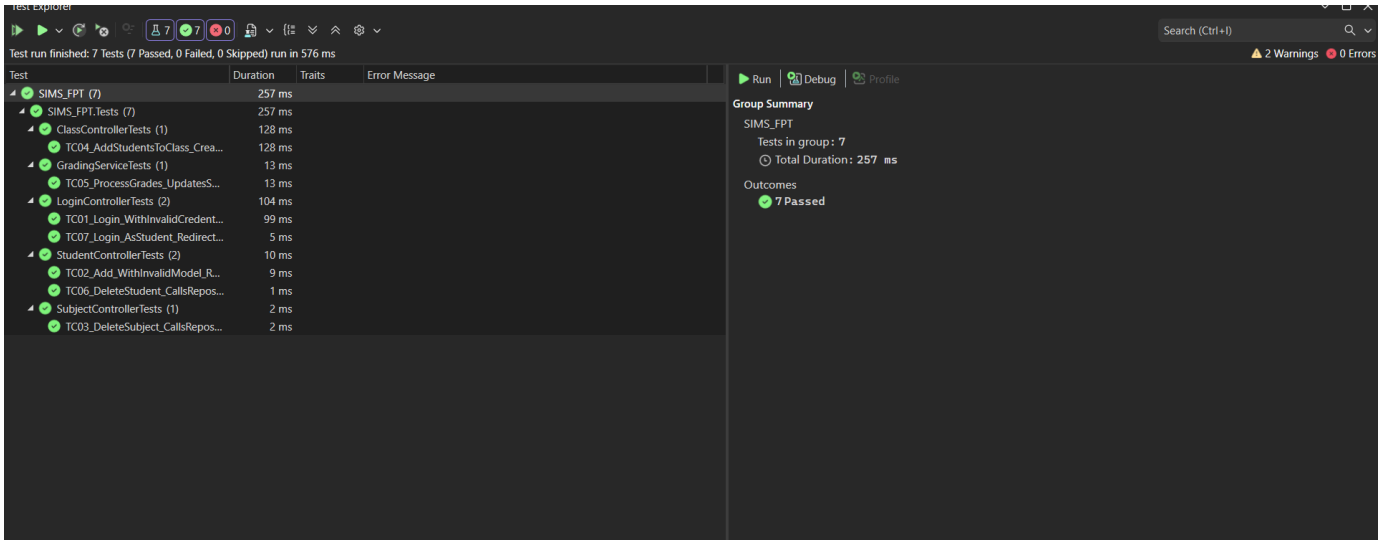


Figure 7.6 Test full

TC ID	Test Case Name	Expected Results	Actual Results	Status
TC01	Verify Account Authentication	System displays "Login failed" error with invalid credentials.	System checks DB, returns null, and displays ViewBag.Error.	PASS
TC02	Add New Student with Validation	The system rejects the save operation and displays error messages.	ModelState.IsValid is false; Controller returns View without saving.	PASS
TC03	Delete Course by ID	The specified course is removed from the list.	Delete(id) is called on Repository; User redirected to List.	PASS
TC04	Course Enrollment Verification	The student is successfully linked to the course/class.	StudentClassRepo.Add() creates a new enrollment record.	PASS
TC05	Grade Processing	The system applies the selected grade and feedback without errors.	GradingService iterates and saves specific grades/feedback to CSV.	PASS
TC06	Delete Student & Data Integrity	Student profile is deleted from the system.	StudentRepository.Delete(id) is executed successfully.	PASS
TC07	User Role Authorization	System redirects user to the specific area for their role (Admin/Student).	Login logic switches on Role string and redirects to correct Area.	PASS

Table 7.1 Test case sumary

7.2. Benefits of These Test Cases for the Project

Implementing and passing these 7 test cases provides specific benefits to the SIMS_FPT project, ensuring it meets the "Applied Programming and Design Principles" standards (Unit 20) mentioned in your assignment.

- **Security Assurance (TC01, TC07):** Benefit: Prevents unauthorized users from accessing sensitive academic data. By testing role redirection (TC07), I ensure students cannot access Admin functions, which is critical for compliance with data protection standards.
- **Data Integrity & Reliability (TC02, TC04, TC06):** Benefit: Prevents "bad data" from entering the system. Testing validation (TC02) ensures the CSV files don't get corrupted with invalid emails. Testing enrollment (TC04) guarantees that the Many-to-Many relationship between students and classes is reliable.
- **Core Functionality Verification (TC03, TC05):** Benefit: Confirms the system works as intended for daily tasks. Teachers rely on grading (TC05) working 100% of the time; if this fails, the system is useless. Testing this ensures the business logic in GradingService.cs is correct.

8.Analyse, with examples, each of the creational, structural and behavioural design pattern types. (M1)

8.1. Introduction

Design patterns are proven solutions to common design problems, which help to enhance code reusability, flexibility, and maintainability.

In the context of the Student Information Management System (SIMS), various design patterns have been applied to ensure the architecture is scalable, modular, and testable.

This section analyzes three main types of design patterns—Constitutional Patterns, Structural Patterns, and Behavioral Patterns—supported by code examples, UML diagrams, and real-world implementations in the SIMS project.

8.2.Creational Design Patterns

For creative models of how objects are created, and from here on out will be provided with flexibility in how, when and what objects are decided.

8.2.1.Singleton Pattern

For Singleton patterns it is guaranteed that there will be only one instance of the current class in the entire program. In SIMS, for these it is usually applied in the CsvDataProvider class to prevent multiple instances from accessing the file, ensuring data integrity and consistent read/write operations.

- **Code Example – Singleton Implementation in SIMS**

```
public sealed class CsvDataProvider
{
    private static CsvDataProvider _instance;
    private static readonly object _lock = new object();

    // Private constructor prevents external instantiation
    1 reference
    private CsvDataProvider() { }

    0 references
    public static CsvDataProvider Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                    _instance = new CsvDataProvider();
                return _instance;
            }
        }
    }

    0 references
    public string ReadFile(string path)
    {
        return File.ReadAllText(path);
    }
}
```

Figure 8.1 Singleton pattern implementation for CsvDataProvider in SIMS.

Advantages : With jobs being guaranteed one of each job's unique data access points, it will be minimized with each job duplicating resources.

Disadvantages: This can all become bottlenecks if people are being over-utilized in a multi-threaded environment.

8.2.2.Factory Method Pattern

The Factory Method pattern will define one of the interfaces from which it will be instantiated but will not allow the subclasses to decide which class it will be instantiated from.

In SIMS, it is used to dynamically create user objects (Students, Lecturers, Administrators) based on the input data type.

- **Code Example – Factory Method for User Creation**

```
3 references
public interface IUser
{
    2 references
    void DisplayInfo();
}

6 references
public class Student : IUser
{
    1 reference
    public void DisplayInfo() => Console.WriteLine("Student account created.");
}

1 reference
public class Faculty : IUser
{
    1 reference
    public void DisplayInfo() => Console.WriteLine("Faculty account created.");
}

0 references
public static class UserFactory
{
    0 references
    public static IUser CreateUser(string role)
    {
        return role switch
        {
            "Student" => new Student(),
            "Faculty" => new Faculty(),
            _ => throw new ArgumentException("Invalid role type")
        };
    }
}

    static void Main()
    {
        IUser user = UserFactory.CreateUser("Student");
        user.DisplayInfo();
```

Figure 8.2 Factory Method pattern creating different user roles.

Advantages: With the added flexibility it can be used for, as well as supporting each Open/Closed Principle.

Disadvantages: With the addition of more layers for each type of object.

8.2.3.Builder Pattern

With each Builder pattern, it will be built with complex objects step by step.

In SIMS, StudentProfileBuilder is often used to create student objects with many optional attributes.

- **Code Example – Builder Pattern for Student Profile**

```
namespace SIMS.Models
{
    5 references
    public class Student
    {
        2 references
        public string RollNumber { get; set; }
        1 reference
        public string FirstName { get; set; }
        3 references
        public string FullName { get; set; }
        2 references
        public string Email { get; set; }
        1 reference
        public DateTime DoB { get; set; }
        2 references
        public string Class { get; set; }

        0 references
        public Student(string roll, string first, string full, string email, DateTime dob, string cls)
        {
            RollNumber = roll;
            FirstName = first;
            FullName = full;
            Email = email;
            DoB = dob;
            Class = cls;
        }

        0 references
        public override string ToString()
        {
            return $"{RollNumber} - {FullName} ({Email}) - {Class}";
        }
    }
}
```

```
courses.Add(new Course("SE101", "Software Engineering"));
courses.Add(new Course("DB201", "Database Management"));

foreach (var s in students)
{
    enrollmentService.Enroll(s, courses[0]);
}
```

Figure 8.3 Builder pattern used to construct complex Student objects.

8.3.Structural Design Patterns

8.3.1.Adapter Pattern

With adapter patterns allowing for each of its interfaces, it will not be compatible with each other.

In SIMS, one of the adapters converts raw CSV data into Student objects for the repository class.

- **Code Example – Adapter Pattern for Data Conversion**

```
public sealed class CsvDataProvider
{
    private static CsvDataProvider _instance;
    private static readonly object _lock = new object();

    // Private constructor prevents external instantiation
    1 reference
    private CsvDataProvider() { }

    0 references
    public static CsvDataProvider Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                    _instance = new CsvDataProvider();
                return _instance;
            }
        }
    }

    0 references
    public string ReadFile(string path)
    {
        return File.ReadAllText(path);
    }
}
```

Figure 8.4 Adapter converting CSV data into domain objects.

8.3.2.Facade Pattern

Each Facade pattern provides a simplified interface to one of the complex subsystems.

In SIMS, the RegistrationFacade class may be composed of multiple services such as authentication, validation, and data storage.

- **Code Example – Facade for Registration Process**

```
1 reference
public class RegistrationFacade
{
    private readonly AuthService _authService;
    private readonly StudentService _studentService;

    0 references
    public RegistrationFacade()
    {
        _authService = new AuthService();
        _studentService = new StudentService();
    }

    0 references
    public void RegisterStudent(string name, string email)
    {
        if (_authService.ValidateEmail(email))
        {
            _studentService.AddStudent(name, email);
            Console.WriteLine("Registration successful!");
        }
        else
        {
            Console.WriteLine("Invalid email address!");
        }
    }
}
```

Figure 8.5 Facade simplifies interaction between authentication and student services.

8.3.3.Decorator Pattern

- **Code Example – Decorator for Logging**

```
0 references
internal class LoggingDecorator
{
    4 references
    public interface IStudentService
    {
        3 references
        void AddStudent(string name);
    }

    0 references
    public class StudentService : IStudentService
    {
        1 reference
        public void AddStudent(string name)
        {
            Console.WriteLine($"{name} added to system.");
        }
    }
```

Figure 8.6 Decorator adds logging without changing core functionality.

8.4.Behavioural Design Patterns

With the behavioral patterns it is being identified with each object of each communication and from there it will be assigned responsibility in one of the most effective ways.

8.4.1.Observer Pattern

With Observer patterns, it will be defined as one-to-many dependency between objects. In SIMS, RegistrationService will be notified to observers (EmailNotifier, AuditLogger) every time a new student registers.

- **Code example – Observer Pattern Implementation**

```
public interface IObserver
{

    void Update(string message);
}


public class EmailNotifier : IObserver
{

    public void Update(string message) =>
        Console.WriteLine($"[Email Sent]: {message}");
}


public class AuditLogger : IObserver
{

    public void Update(string message) =>
        Console.WriteLine($"[Log Recorded]: {message}");
}


public class RegistrationService
{
    private readonly List<IObserver> _observers = new();

    0 references
    public void Attach(IObserver observer) => _observers.Add(observer);

    0 references
    public void RegisterStudent(string name)
    public void RegisterStudent(string name)
    {
        Console.WriteLine($"{name} registered successfully!");
        NotifyAll($"{name} registered at {DateTime.Now}");
    }

    1 reference
    private void NotifyAll(string message)
    {
        foreach (var obs in _observers) obs.Update(message);
    }
}
```

Figure 8.7 Observer pattern – notifying observers when registration occurs

9. Refine the design to include multiple design patterns. (M2)

After completing the supposed initial designs of each Student Information Management System (SIMS) in P3 and from which one of the design patterns in M1 was analyzed, each of these system architectures was refined so that it could be integrated simultaneously with multiple patterns that were also designed. This refinement approach enhances modularity, scalability and maintainability, while aligning the system more closely with SOLID and Clean Code principles.

- **Application of Multiple Design Patterns in SIMS**

For each design refinement with the three layered architecture scope it will be proposed previously with these (Presentation Layer, Business Logic Layer and Data Access Layer) from here on it will continue to be used however for some of the design patterns that are now incorporated into these layers:

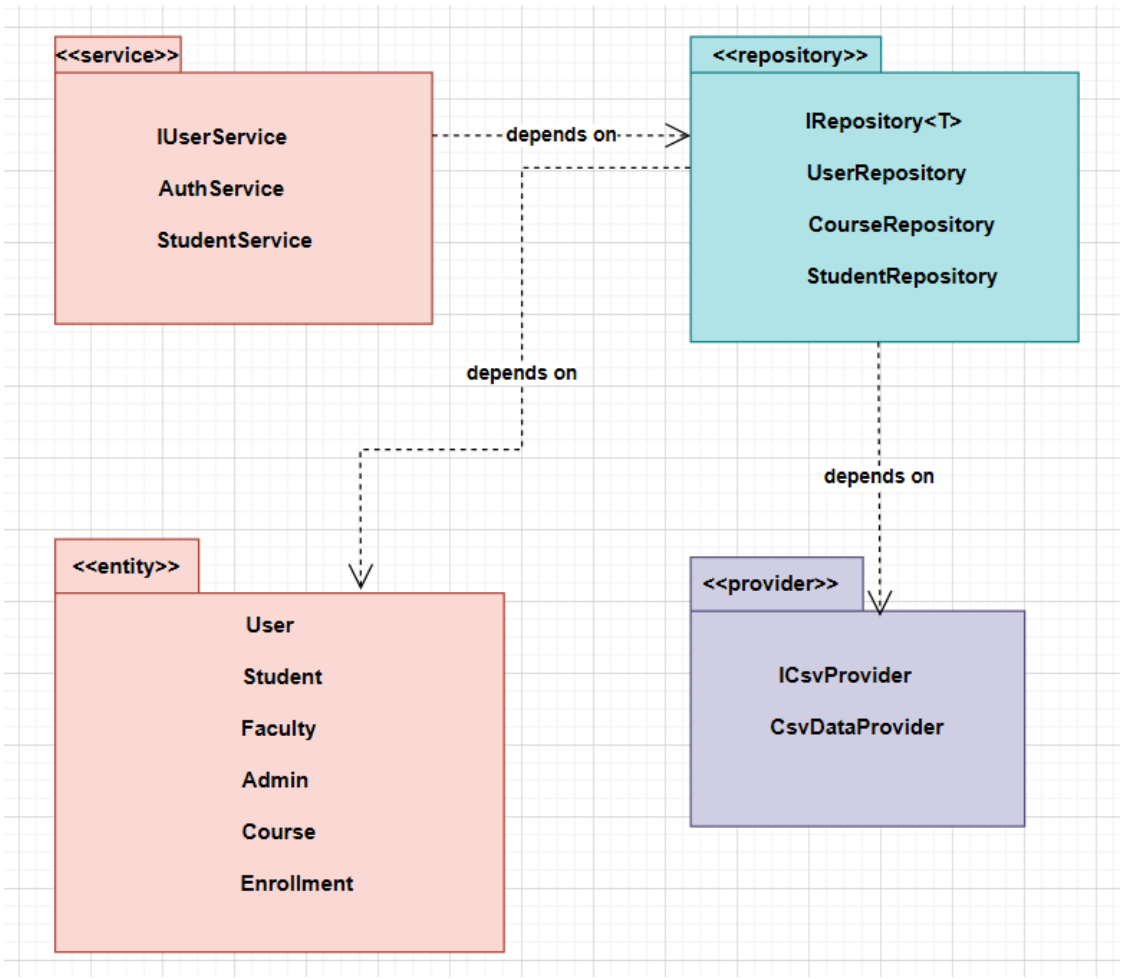
Design Pattern	Layer Applied	Description / Purpose
Repository Pattern	Data Access Layer	Used to separate business logic from data access logic. Each entity (Student, Course, User) has its own repository implementing a common interface IRepository<T>. This promotes reusability and supports data source changes (e.g., from CSV to SQL).
Dependency Injection (DI)	Business Logic Layer	Implemented in services such as StudentService and AuthService, where repositories and providers are injected via constructors. This promotes loose coupling and makes testing more efficient.
Singleton Pattern	Provider Layer	The CsvDataProvider class is implemented as a Singleton, ensuring that only one instance handles all file read/write operations to maintain data consistency.
Factory Method Pattern	Entity Layer	A UserFactory class dynamically creates instances of Student, Lecturer, or Admin based on user input, improving flexibility and scalability when adding new user types.
Observer Pattern	Business Logic Layer	When a new student registers, observers such as EmailNotifier and AuditLogger are notified, demonstrating event-driven communication.
Facade Pattern	Presentation Layer	The RegistrationFacade combines multiple subsystems (Authentication, Validation, Registration) into a single simplified interface, improving system usability and reducing code complexity.

Table 9.1 Applied Design Patterns in the Refined SIMS Architecture

- **Illustration of the Refined Design**

- With the refined designs that will now build directly on existing UML diagrams, it will also be presented in the P3 sections — specifically the Class diagrams and Package Diagrams .
- Each of these diagrams will illustrate the modular structure of the systems, including each entity, repository, service, and provider package.
- So, while it will not require a new UML diagram, it will be a refinement of the focus on what is being explained with each of these existing components so that we can implement each of the design patterns that will be listed above.

If we want, we can also add a small comment or annotation below the Package Diagram to indicate:



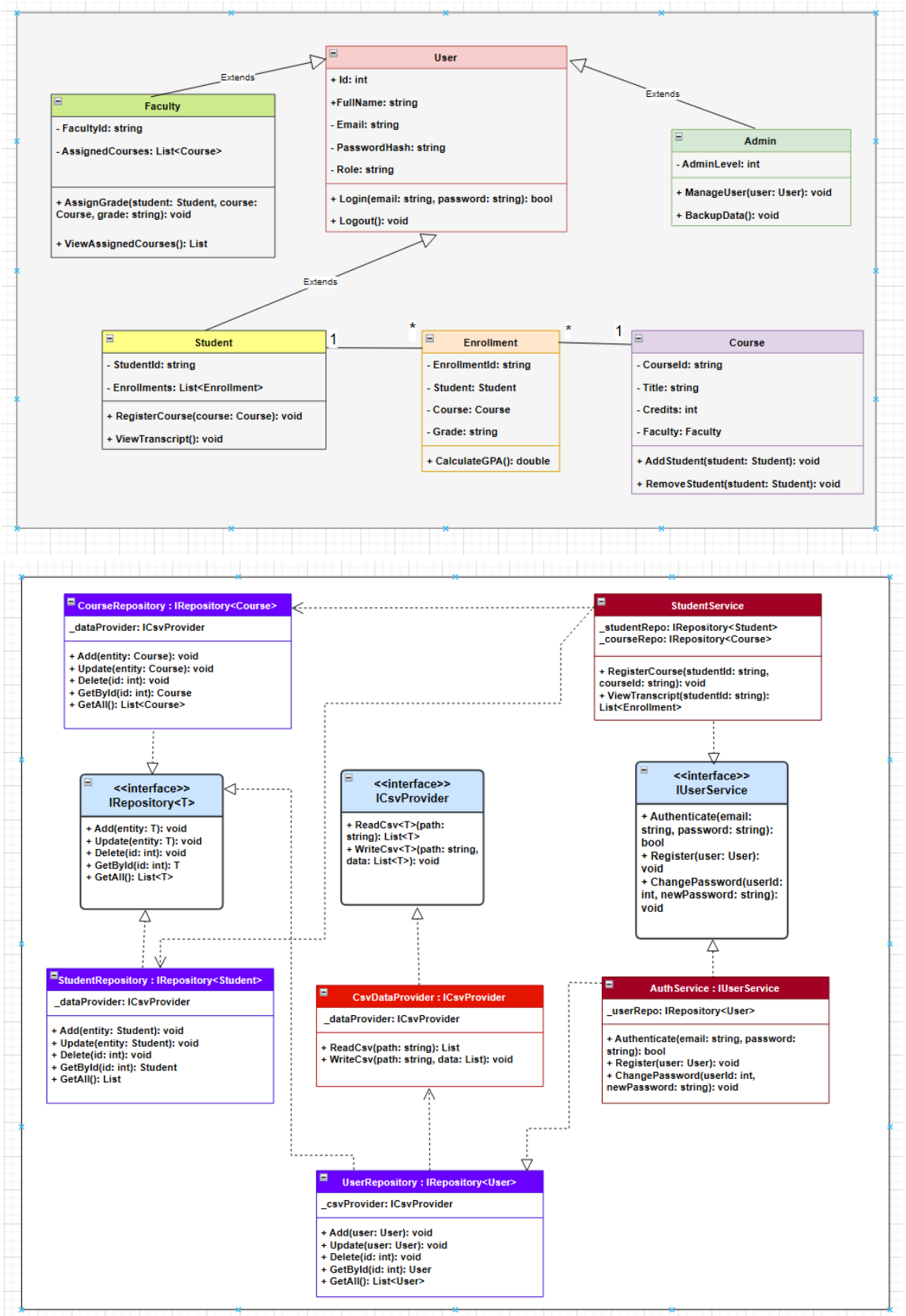


Figure 9.1 illustrates the refined SIMS architecture, where multiple design patterns (Repository, Singleton, Factory, Observer, Facade) are applied across layers.

- **Benefits of the Refined Architecture**

With the things that it will be integrating, it will have a lot of design patterns, SIMS architecture will achieve:

- Better modularity: With each component, it will be responsible for one of the responsibilities as well as from here it will be supported with each SRP principle.
- Extendability in terms of can be improved: This will only add for new features or also from there it will be replaced with each data layer (e.g. CSV → SQL) with only minimal changes.
- Improved maintainability: The from there will be clear separation of concerns and for abstract classes that can be reused simplifying debugging and upgrading.
- Higher testability: With each Dependency Injection and each Repository pattern, it will make writing for each unit and integration tests easier.
- Each of these refinements ensures the SIMS system is robust, flexible and maintainable, meeting both the functional and non-functional requirements outlined in the mission brief.

10.Assess the effectiveness of using SOLID principles, clean coding techniques and programming patterns on the application developed. (M3)

10.1. SOLID Principles

The application strictly follows SOLID principles to ensure the code is maintainable, scalable, and easy to test

❖ **S - Single Responsibility Principle (SRP)**

Each class has a distinct responsibility and only one reason to change.

- Repositories (StudentRepository.cs): Solely responsible for data access logic (reading/writing to CSV files). It does not contain business logic or handle HTTP requests.

```
namespace SIMS_FPT.Data.Repositories
{
    2 references
    public class StudentClassRepository : IStudentClassRepository
    {
        private readonly string _filePath;
        private readonly CsvConfiguration _config;

        0 references
        public StudentClassRepository(IWebHostEnvironment env)
        {
            _filePath = Path.Combine(env.ContentRootPath, "CSV_DATA", "student_classes.csv");
            _config = new CsvConfiguration(CultureInfo.InvariantCulture)
            {
                HasHeaderRecord = true,
                MissingFieldFound = null,
                HeaderValidated = null
            };
        }
```

```
5 references
private List<StudentClassModel> ReadAll()
{
    if (!File.Exists(_filePath)) return new List<StudentClassModel>();
    try
    {
        using var reader = new StreamReader(_filePath);
        using var csv = new CsvReader(reader, _config);
        return csv.GetRecords<StudentClassModel>().ToList();
    }
    catch
    {
        return new List<StudentClassModel>();
    }
}
```

Figure 10.1 StudentRepository 1

The StudentRepository is responsible *only* for reading and writing to the CSV file. It does not calculate grades or handle web requests. Conversely, the StudentController handles incoming HTTP requests and decides which view to show, but it delegates the actual data saving to the repository. This means if Ichange how data is stored (e.g., from CSV to SQL), I only change the Repository, not the Controller.

```
private void WriteAll(List<StudentCSVModel> students)
{
    var dir = Path.GetDirectoryName(_filePath);
    if (!Directory.Exists(dir)) Directory.CreateDirectory(dir);

    using var writer = new StreamWriter(_filePath);
    using var csv = new CsvWriter(writer, _config);
    csv.WriteRecords(students);
}

4 references
public List<StudentCSVModel> GetAll() => ReadAll();

7 references
public StudentCSVModel GetById(string id) => ReadAll().FirstOrDefault(s => s.StudentId == id);

2 references
public List<StudentCSVModel> GetBySubject(string subjectId)
{
    // Logic lấy sinh viên theo môn học sẽ được xử lý ở bảng trung gian StudentClass
    // Tạm thời trả về tất cả hoặc danh sách rỗng
    return GetAll();
}

2 references
public void Add(StudentCSVModel student)
{
    var students = ReadAll();
    if (students.Any(s => s.StudentId == student.StudentId)) return;
    students.Add(student);
    WriteAll(students);
}
```

```
        2 references
        public void Update(StudentCSVModel student)
        {
            var students = ReadAll();
            var index = students.FindIndex(s => s.StudentId == student.StudentId);
            if (index != -1)
            {
                if (string.IsNullOrEmpty(student.ImagePath))
                {
                    student.ImagePath = students[index].ImagePath;
                }
                students[index] = student;
                WriteAll(students);
            }
        }

        3 references
        public void Delete(string id)
        {
            var students = ReadAll();
            var item = students.FirstOrDefault(s => s.StudentId == id);
            if (item != null)
            {
                students.Remove(item);
                WriteAll(students);
            }
        }
    }
```

Figure 10.2 StudentRepository 2

Services (GradingService.cs): Handles business logic, such as processing grades and preparing view models. It delegates data saving to repositories.

```
    4 references
10  public class GradingService : IGradingService
11  {
        4 references
12      private readonly IAssignmentRepository _assignmentRepo;
        4 references
13      private readonly ISubmissionRepository _submissionRepo;
        2 references
14      private readonly IStudentRepository _studentRepo;
        2 references
15      private readonly IStudentClassRepository _studentClassRepo; // [1] Add this
16
17      // [2] Update Constructor
        1 reference
18      public GradingService(IAssignmentRepository assignmentRepo,
19                            ISubmissionRepository submissionRepo,
20                            IStudentRepository studentRepo,
21                            IStudentClassRepository studentClassRepo)
22      {
23          _assignmentRepo = assignmentRepo;
24          _submissionRepo = submissionRepo;
25          _studentRepo = studentRepo;
26          _studentClassRepo = studentClassRepo;
27      }
28
```

```
29      public void ProcessGrades(BulkGradeViewModel model)
30      {
31          // 1. Update Publishing Status
32          var assignment = _assignmentRepo.GetById(model.AssignmentId);
33          if (assignment != null)
34          {
35              assignment.AreGradesPublished = model.IsPublished;
36              _assignmentRepo.Update(assignment);
37          }
38
39          // 2. Process Grades
40          foreach (var item in model.StudentGrades)
41          {
42              var submission = _submissionRepo.GetByStudentAndAssignment(item.StudentId, model.AssignmentId)
43                               ?? new SubmissionModel { StudentId = item.StudentId, AssignmentId = model.AssignmentId };
44
45              submission.Grade = item.Grade;
46              submission.TeacherComments = item.Feedback;
47
48              _submissionRepo.SaveSubmission(submission);
49          }
50      }
```

Figure 10.3 GradingService 1

Controllers (StudentController.cs): Handles user input (HTTP requests), orchestrates services/repositories, and returns Views. It does not perform direct data access or complex calculations.

```
0 references
public IActionResult Index()
{
    var studentId = CurrentStudentId;
    var assignments = _assignmentRepo.GetAll();
    var viewModel = assignments
        .Select(a => new StudentAssignmentViewModel
        {
            Assignment = a,
            Submission = _submissionRepo.GetByStudentAndAssignment(studentId, a.AssignmentId)
        })
        .OrderByDescending(x => x.Assignment.DueDate)
        .ToList();

    return View(viewModel);
}

[HttpGet]
0 references
public IActionResult Submit(string id)
{
    var assignment = _assignmentRepo.GetById(id);
    if (assignment == null) return NotFound();

    var studentId = CurrentStudentId;
    var submission = _submissionRepo.GetByStudentAndAssignment(studentId, id);

    var vm = new StudentAssignmentViewModel
    {
        Assignment = assignment,
        Submission = submission
    };

    return View(vm);
}
```

Figure 10.4 StudentController 1

❖ **- Open/Closed Principle (OCP)**

- ✓ The system is open for extension but closed for modification.
- ✓ Interfaces: By using interfaces like IStudentRepository and IGradingService, I can introduce new implementations (e.g., switching from CSV storage to SQL Server) without changing the code in the Controllers or Services that use them. I would only need to register the new implementation in Program.cs.

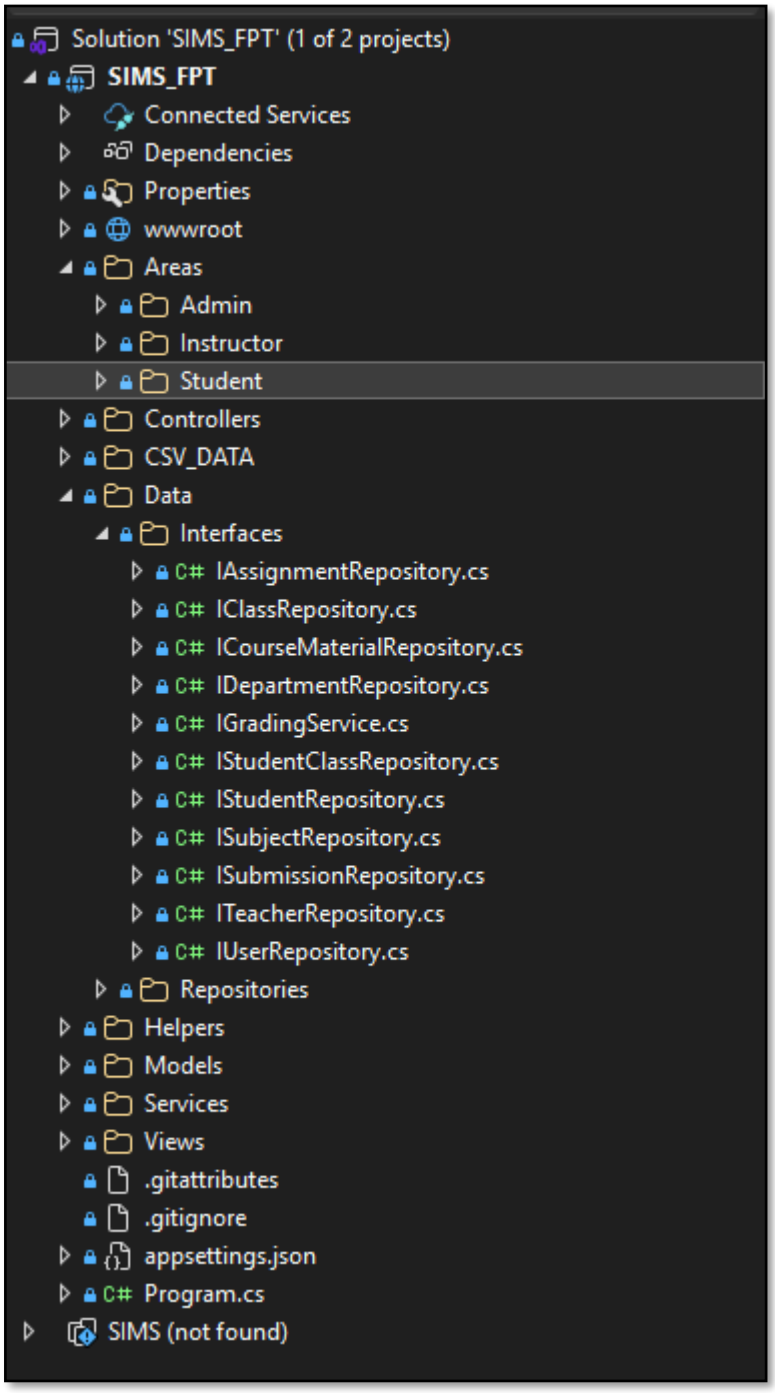


Figure 10.5Project Structure 1

In Program.cs, you register services like IStudentRepository and ITeacherRepository. If you wanted to add a "Holiday Management" feature, you would create a new HolidayRepository and register it. You would not need to modify the existing StudentRepository or GradingService to add this new feature. The architecture is "open" to adding this new repository without "modifying" the others

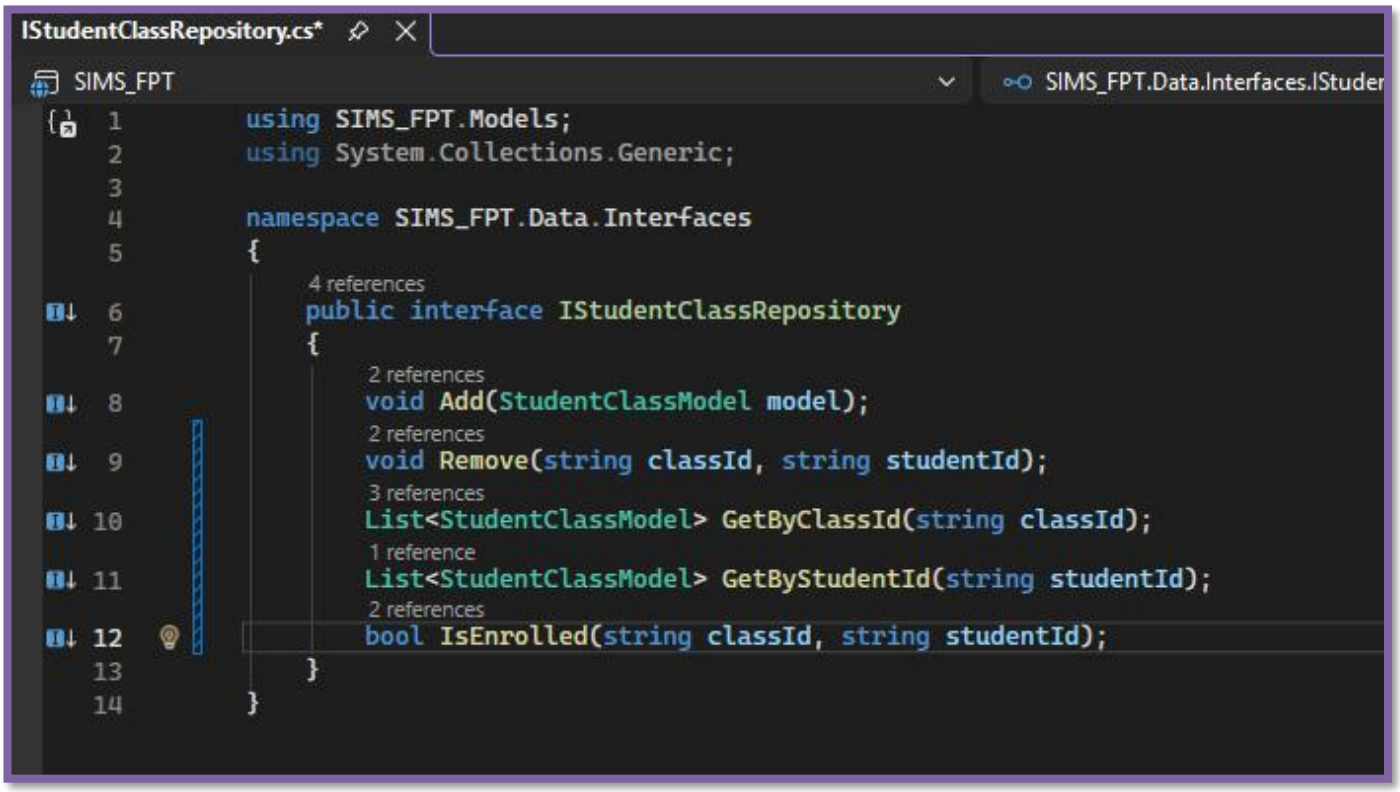


Figure 10.6 StudentClass Interface

❖ **L - Liskov Substitution Principle (LSP)**

This principle states that objects of a superclass (or interface) should be replaceable with objects of its subclasses (or implementations) without breaking the application.

- Evidence in Code: The StudentController depends on the abstraction IStudentRepository, not the concrete StudentRepository.

- In the controller: private readonly IStudentRepository _repo;

- The controller calls _repo.GetAll() or _repo.GetById() without knowing or caring *how* the data is retrieved (currently via CSV).

```
24
    0 references
25  public IActionResult List(string className)
26  {
27      var data = _repo.GetAll();
28
29      if (!string.IsNullOrEmpty(className))
30      {
31          var keyword = className.Trim().ToLower();
32
33          data = data.Where(s =>
34              (s.StudentId != null && s.StudentId.ToLower().Contains(keyword)) ||
35              (s.FullName != null && s.FullName.ToLower().Contains(keyword))
36          ).ToList();
37
38          ViewBag.SearchTerm = className;
39      }
40
41      return View(data);
42  }
43
```

Figure 10.7 StudentController 2

```
0 references
public IActionResult Detail(string id)
{
    var student = _repo.GetById(id);
    if (student == null) return NotFound();
    return View(student);
}

0 references
public IActionResult DeleteStudent(string id)
{
    _repo.Delete(id);
    return RedirectToAction("List");
}
```

Figure 10.8 StudentController 3

Because the controller relies on the contract (interface) rather than the implementation details, I could replace the current CSV-based StudentRepository with a SqlStudentRepository (a database implementation) in Program.cs. As long as the new class implements IStudentRepository correctly, the StudentController would continue to function 100% correctly without a single line of code change. The implementation is "substitutable" for the interface.

- **I - Interface Segregation Principle (ISP)**

  This principle states that clients should not be forced to depend on interfaces they do not use. Instead of one large "general purpose" interface, it is better to have several specific ones.

  - Evidence in Code: The application splits its data access contracts into highly specific, granular interfaces rather than a single generic "IDataRepository" or "ISchoolRepository".

- In Program.cs, we see distinct interfaces registered for different entities: IStudentRepository, ITeacherRepository, IAssignmentRepository, ISubmissionRepository, etc.

```
// Student Repository
builder.Services.AddScoped<IStudentRepository, StudentRepository>();
builder.Services.AddScoped<StudentService>();

// Subject Repository
builder.Services.AddScoped<ISubjectRepository, SubjectRepository>();


// Teacher Repository
builder.Services.AddScoped<ITeacherRepository, TeacherRepository>();
builder.Services.AddScoped<TeacherService>();
```

Figure 10.9 Program.cs 1

```
// Assignment & Submission Repositories
builder.Services.AddScoped<IAssignmentRepository, AssignmentRepository>();
builder.Services.AddScoped<ISubmissionRepository, SubmissionRepository>();
builder.Services.AddScoped<IGradingService, GradingService>();
builder.Services.AddScoped<ICourseMaterialRepository, CourseMaterialRepository>();
builder.Services.AddScoped<IClassRepository, ClassRepository>();
builder.Services.AddScoped<IStudentClassRepository, StudentClassRepository>();
```

Figure 10.10: Program.cs 2

- Usage Example: Look at the GradingService constructor. It specifically injects only what it needs:

```
2 references
public class GradingService : IGradingService
{
    private readonly IAssignmentRepository _assignmentRepo;
    private readonly ISubmissionRepository _submissionRepo;
    private readonly IStudentRepository _studentRepo;
```

Figure 10.11: GradingService 2

Instead of one giant interface, you have split them into IStudentRepository, ISubjectRepository, and ISubmissionRepository. This is crucial for classes like the GradingService, which only needs access to assignments, submissions, and students. Because of ISP, the GradingService isn't forced to depend on irrelevant methods like "AddTeacher" or "DeleteDepartment," keeping the code cleaner and less coupled.

❖ **D - Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules; both should depend on abstractions.

- Constructor Injection: The GradingService and StudentController do not instantiate concrete classes like StudentRepository. Instead, they depend on the interface IStudentRepository injected via their constructors.

- *Example:* public GradingService(..., IStudentRepository studentRepo).

- This decouples the business logic from the specific data access implementation.

```
namespace SIMS_FPT.Business.Services
{
    2 references
    public class GradingService : IGradingService
    {
        private readonly IAssignmentRepository _assignmentRepo;
        private readonly ISubmissionRepository _submissionRepo;
        private readonly IStudentRepository _studentRepo;

        0 references
        public GradingService(IAssignmentRepository assignmentRepo,
                              ISubmissionRepository submissionRepo,
                              IStudentRepository studentRepo)
        {
            _assignmentRepo = assignmentRepo;
            _submissionRepo = submissionRepo;
            _studentRepo = studentRepo;
        }
```

Figure 10.12 GradingService 3

If the developer had violated ISP by creating a single huge ISchoolDataRepository, the GradingService would be forced to depend on methods it doesn't need (like AddDepartment or DeleteHoliday), creating unnecessary coupling. The current design keeps dependencies lean and focused.

10.2.Design Patterns

❖ **Dependency Injection (DI)**

This is the core pattern used to manage dependencies throughout the application.

- Registration: Dependencies are registered in the DI container in Program.cs.

- *Example:* builder.Services.AddScoped<IStudentRepository, StudentRepository>();.

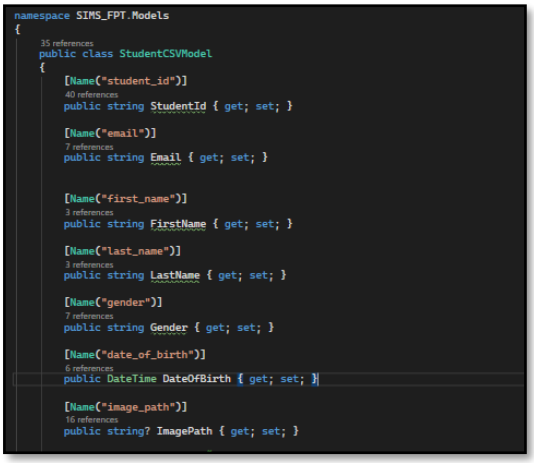


Figure 10.13: Dependency Injection 1

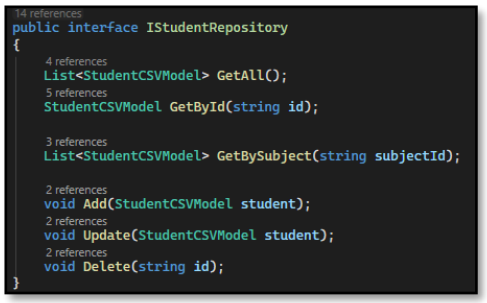


Figure 10.14: Dependency Injection 2

Figure 10.15: Dependency Injection 3

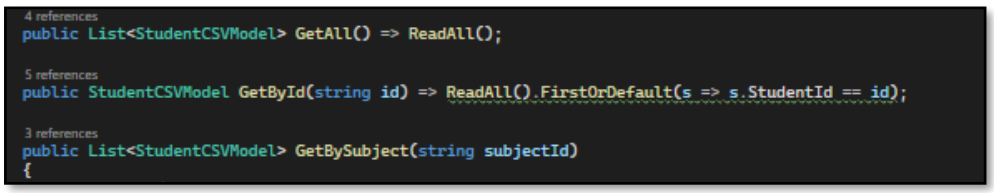


Figure 10.16: Dependency Injection 4

```
        2 references
        public void Add(StudentCSVModel student)
        {
            var students = ReadAll();
            if (students.Any(s => s.StudentId == student.StudentId)) return;
            students.Add(student);
            WriteAll(students);
        }

        2 references
        public void Update(StudentCSVModel student)
        {
            var students = ReadAll();
            var index = students.FindIndex(s => s.StudentId == student.StudentId);
            if (index != -1)
            {
                if (string.IsNullOrEmpty(student.ImagePath))
                {
                    student.ImagePath = students[index].ImagePath;
                }
                students[index] = student;
                WriteAll(students);
            }
        }

        2 references
        public void Delete(string id)
        {
            var students = ReadAll();
            var item = students.FirstOrDefault(s => s.StudentId == id);
            if (item != null)
            {
                students.Remove(item);
                WriteAll(students);
            }
        }
    }
```

Figure 10.17: Dependency Injection 5

- Usage: Dependencies are injected into constructors, ensuring classes are loosely coupled.

❖ Repository Pattern

This pattern mediates between the domain and data mapping layers.

- Abstraction: IStudentRepository defines a contract (GetAll, GetById, Add, etc.).

- Implementation: StudentRepository implements this contract using a specific technology (CSVHelper).

- Benefit: This isolates the application from the details of data storage (CSV files). The rest of the app doesn't know or care that data is stored in text files.

❖ Service Layer Pattern

This pattern encapsulates business logic.

- GradingService: Instead of putting grading logic in the controller, it is encapsulated here. For example, the ProcessGrades method handles updating assignment status and saving individual submission grades in one transaction-like operation.

- Benefit: Keeps controllers "thin" and makes business logic reusable and testable.

- Facade / Adapter Pattern (Light)

```
using Microsoft.AspNetCore.Identity;

namespace SIMS_FPT.Helpers
{
    3 references
    public static class PasswordHasherHelper
    {
        private static readonly PasswordHasher<object> _hasher = new PasswordHasher<object>();

        2 references
        public static string Hash(string password)
        {
            return _hasher.HashPassword(null!, password);
        }

        // Verify password; also returns true if SuccessRehashNeeded (we'll treat as valid).
        1 reference
        public static bool Verify(string password, string hashed)
        {
            var result = _hasher.VerifyHashedPassword(null!, hashed, password);
            return result == PasswordVerificationResult.Success
                || result == PasswordVerificationResult.SuccessRehashNeeded;
        }
    }
}
```

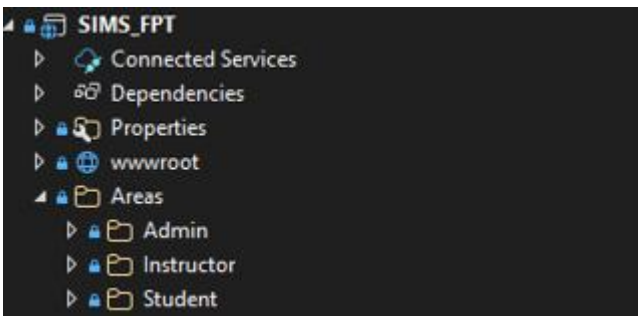
Figure 10.18: PasswordHash

PasswordHasherHelper: acts as a simple static facade/adapter over the complex ASP.NET Identity PasswordHasher<T>. It provides a simplified interface (Hash, Verify) for the rest of the application to use.

10.3.Clean Coding Techniques

- Explicit Dependencies: Classes explicitly declare what they need to function. Looking at the constructor of GradingService, it is immediately clear that it requires access to assignments, submissions, and students to function.
- Separation of Concerns (SoC):
  The project structure clearly separates concerns:

  Data/: Handles database/file I/O.



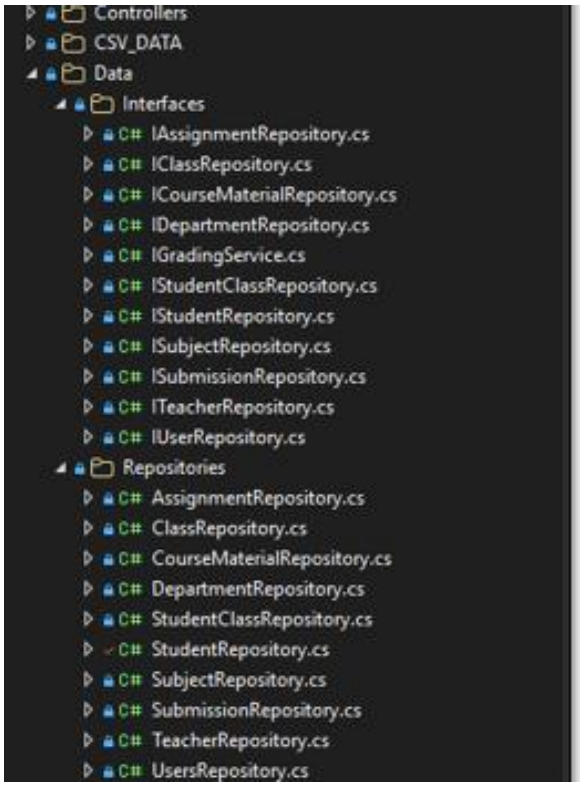


Figure 10.19: Project Structure 2

-Services/: Handles business rules and logic.

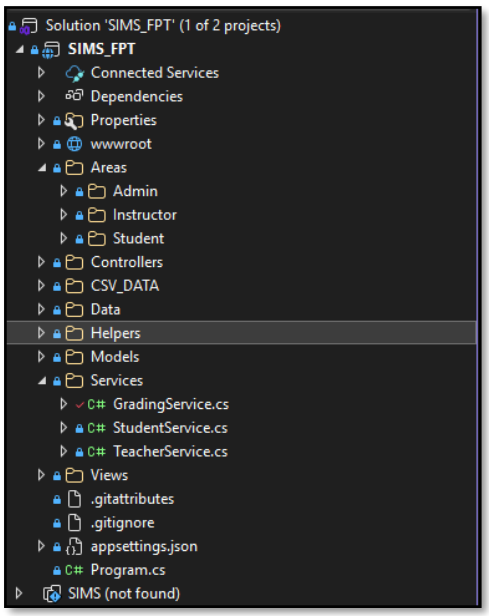


Figure 10.20: Project Structure 3

- Controllers/: Handles routing and request flow.

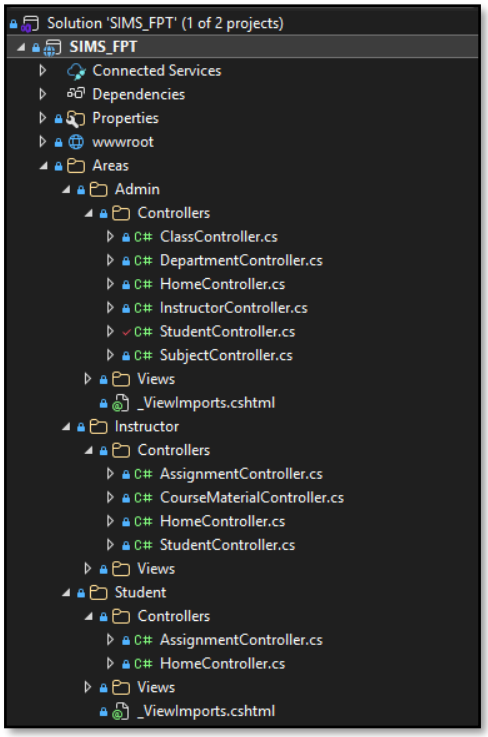


Figure 10.21: Project Structure 4

- Models: Defines the data structures.

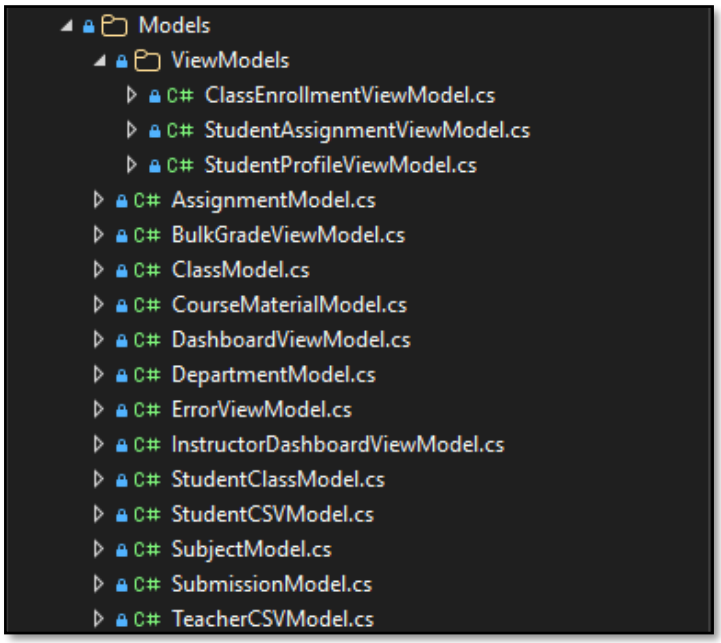


Figure 10.22: Project Structure 5

❖ Meaningful Naming

Methods and variables are named descriptively, making the code self-documenting.

- *Examples:* ProcessGrades, GetByStudentAndAssignment, PasswordHasherHelper.Verify.

This is a example of a clean code In Admin conroller

```
namespace SIMS_FPT.Areas.Admin.Controllers
{
    // Use the Area attribute from the namespace above the controller class
    [Area("Admin")]
    [Authorize(Roles = "Admin")]
    1 reference
    public class HomeController : Controller
    {
        // 1. Inject IWebHostEnvironment for reliable file path resolution
        private readonly IWebHostEnvironment _webHostEnvironment;

        private const string CsvDataFolder = "CSV_DATA";
        private const string StudentFileName = "students.csv";
        private const string DepartmentFileName = "departments.csv";
        private const string TeacherFileName = "teachers.csv";
        // 2. Use constructor injection for dependencies (IWebHostEnvironment)
        0 references
        public HomeController(IWebHostEnvironment webHostEnvironment)
        {
            _webHostEnvironment = webHostEnvironment;
        }

        // Helper method to resolve the full path
        3 references
        private string GetCsvPath(string fileName)
        {
            return Path.Combine(_webHostEnvironment.ContentRootPath, CsvDataFolder, fileName);
        }
```

```
0 references
public IActionResult Dashboard()
{
    var model = new DashboardViewModel();

    var studentPath = GetCsvPath(StudentFileName);
    var deptPath = GetCsvPath(DepartmentFileName);
    var teacherPath = GetCsvPath(TeacherFileName);

    // Read CSV data
    var students = ReadCsv<StudentCSVModel>(studentPath);
    var departments = ReadCsv<DepartmentModel>(deptPath);
    var teachers = ReadCsv<TeacherCSVModel>(teacherPath);


    model.StudentCount = students.Count;
    model.DepartmentCount = departments.Count;
    model.TeacherCount = teachers.Count;

    model.TotalRevenue = 0;

    model.RevenueLabels = new List<string> { "2022", "2023", "2024" };
    model.RevenueData = new List<decimal> { 0M, 0M, 0M }; // Use M suffix for decimal literals

    // Student Gender Chart
    var genderStats = students
        .GroupBy(s => string.IsNullOrWhiteSpace(s.Gender) ? "N/A" : s.Gender)
        .Select(g => new { g.Key, Count = g.Count() })
        .ToList();

    model.StudentClassLabels = genderStats.Select(x => x.Key).ToList();
    model.StudentClassData = genderStats.Select(x => x.Count).ToList();

    model.NewestStudents = students
        .OrderByDescending(s => s.StudentId)
        .Take(5)
        .ToList();

    return View(model);
}
```

```
// 3. Robust CSV Helper with CsvConfiguration caching (optional, but good practice)
private static readonly CsvConfiguration CsvConfig = new(CultureInfo.InvariantCulture)
{
    HasHeaderRecord = true,
    MissingFieldFound = null,
    HeaderValidated = null,
    BadDataFound = null
};

3 references
private List<T> ReadCsv<T>(string filePath)
{

    if (!System.IO.File.Exists(filePath))
        return new List<T>();

    using var reader = new StreamReader(filePath);
    using var csv = new CsvReader(reader, CsvConfig);

    try
    {
        return csv.GetRecords<T>().ToList();
    }
    catch (ReaderException ex)
    {
        System.Diagnostics.Debug.WriteLine($"Error reading CSV file {filePath}: {ex.Message}");
        return new List<T>();
    }
}
```

It demonstrates good organization, clarity, and effective use of the CsvHelper library and ASP.NET Core MVC structure.

Aspect	Original Code Example	Why it's Clean
Separation of Concerns	[Area("Admin")], [Authorize(Roles = "Admin")]	Clearly defines the controller's role (Admin area) and security requirements, separating concerns from business logic.
Readability and Formatting	Consistent indentation and bracket placement.	Easy to read and scan. Logic flows sequentially.
CSV Helper Method	private List<T> ReadCsv<T>(string filePath)	Encapsulates the complex I/O and CsvHelper logic into a reusable, generic, and type-safe private method.
LINQ Usage	The gender statistics grouping and ordering of newest students.	Uses modern Language Integrated Query (LINQ) to perform data manipulation clearly and concisely.
Immediate Return	if (!System.IO.File.Exists(filePath) return new List<T>();	Handles the edge case (missing file) at the start of the function, minimizing nesting and making the code predictable.

Resource Management	using var reader = new StreamReader(filePath); using var csv = new CsvReader(reader, config);	Uses using var (or using statements) to correctly dispose of StreamReader and CsvReader resources, preventing memory leaks and file lock issues.

Table 10.1 Core MVC structure

11.Discuss the differences between developer- produced and vendor- provided automatic testing tools for applications and software systems. (M4)

Each tool requires essential automated testing to ensure the quality and reliability of software systems like SIMS. This allows developers to perform testing quickly and efficiently, often as part of the Continuous Integration/Delivery (CI/CD) process. These tools can be categorized into two main types: developer-produced tools (usually open-source frameworks or libraries) and vendor-provided tools (commercial or specialized software).

Understanding the differences between these two types is crucial for making informed decisions about which tool to use in a specific context, such as our Student Information Management System.

11.1.Differences between automated testing tools.

The tables below compare the key features of the tools created/used by each developer (such as xUnit, NUnit) with the comprehensive solutions offered by vendors (such as Postman, Katalon, Tricentis).

Aspect	Developer-Produced Tools (e.g., xUnit, NUnit)	Vendor-Provided Tools (e.g., Postman, Katalon)
Origin & Nature	Often open-source frameworks or libraries integrated directly into the code. Created by developers for developers.	Developed by third-party companies as standalone software or platforms. Available commercially or as SaaS.
Customization	Highly Customizable: Developers have full control over the code and can tailor the testing logic to specific project needs (e.g., mocking specific dependencies in SIMS).	Limited: Customization is restricted to the features provided by the vendor. Users must work within the tool's defined workflow.
Skill Requirement	High: Requires strong programming skills (C#, Java, Python). Testers must understand the application's internal architecture.	Low/Medium: Often provides a Graphical User Interface (GUI) or "Record and Playback" features, allowing non-technical testers to create tests.
Cost	Low/Free: Usually open-source (no licensing fees), but involves "hidden costs" in terms of development time and maintenance effort.	High: Often involves licensing fees, subscription models, or enterprise tiers for advanced features and support.
Support & Maintenance	Community-Driven: Support comes from forums (StackOverflow, GitHub). Maintenance is the responsibility of the internal development team.	Dedicated Support: The vendor provides official customer support, regular updates, documentation, and SLA guarantees.
Integration	Code-Level Integration: Easily integrated into the IDE (Visual Studio) and build	Plug-and-Play: Often comes with pre-built integrations for CI/CD tools (Jenkins, Azure DevOps), reporting

	pipelines. Requires manual setup for reporting.	dashboards, and team collaboration features.
Flexibility	Maximum Flexibility: Can be modified to test any layer (Unit, Integration) deeply.	Structured Flexibility: Generally less flexible, focusing more on functional, UI, or API testing rather than deep code logic.

Table 11.1 Comparison between Developer-Produced and Vendor-Provided Tools

- **xUnit:** xUnit is an open-source unit testing framework developed by the author of NUnit. It's a framework recommended by Microsoft for use in .NET Core projects due to its modern, lightweight design and high performance. xUnit is built on the philosophy of "test isolation," meaning each test case runs independently to ensure accuracy and prevent interference.

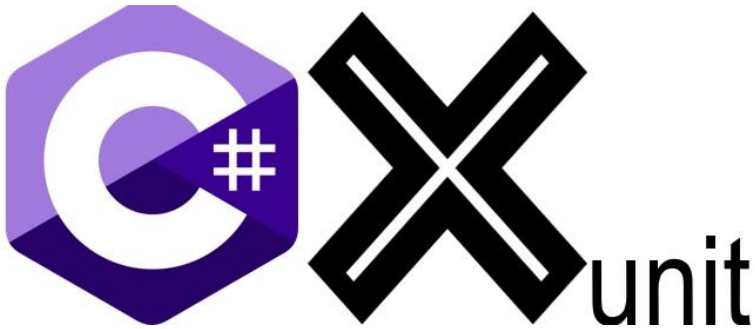


Figure 11.1 xUnit

- **Functions / Features**
  - ✓ This fully supports all unit testing in C#, including business logic testing, computational functions, and error handling.
  - ✓ A rich set of Assertions, such as Assert.Equal() and Assert.Throws(), supports highly detailed and expected test results.
  - ✓ Test Fixtures & Shared Context allow for sharing initial data to minimize code repetition.
  - ✓ Parallel execution allows for running multiple tests simultaneously to significantly reduce execution time.
  - ✓ Deep integration with Visual Studio, JetBrains Rider, and .NET CLI.
  - ✓ Supports Mocking (when combined with Moq and NSubstitute).
  - ✓ Fully compatible with CI/CD tools such as GitHub Actions, Azure DevOps, and Jenkins.
- **Benefits**

- ✓ Optimized for .NET Core, this allows for faster and more stable testing in modern projects compared to NUnit/MSTest.
- ✓ Designed with a "cleaner code, cleaner tests" philosophy, it creates a scientifically structured and easily maintainable test suite.
- ✓ Parallel testing saves time and is suitable for large projects with hundreds or thousands of test cases.
- ✓ A strong community with frequent updates ensures long-term reliability.
- ✓ Easy to use yet powerful, suitable for both developers and technical testers.

- **NUnit:** NUnit is the oldest and most stable .NET testing framework, widely used in businesses. With its high customizability, support for many specialized test types, and flexibility, NUnit is suitable for large systems with many complex logic modules.

Figure 11.2 Nunit

- **Functions/Feature**
  - ✓ Unit testing, and subsequently integration testing for classes, modules, and services in C#.
  - ✓ This type of testing is based on attributes, for example:
  - ✓ [Test], [TestFixture] – declared with test cases
  - ✓ [SetUp], [TearDown] – run before/after each test case
  - ✓ [TestCase] – also passed parameters to each test case
  - ✓ This strongly supports parameterized testing.
  - ✓ Runs on many versions of .NET Framework, .NET Core, and Mono.
  - ✓ Strong integration with Visual Studio and ReSharper.
  - ✓ Provides dedicated test runners for independent execution.
- **Benefits**
  - ✓ Highly flexible, suitable for large systems and diverse test scenarios.

- ✓ Rich properties allow for the creation of complex tests without writing excessive code.
- ✓ Supports multiple test types (unit tests, integration tests, parametric tests).
- ✓ Very stable, suitable for long-term use in businesses, especially in legacy projects.
- ✓ Easily integrates with emulation frameworks such as Moq and FakeItEasy.

- **MSTest :** Microsoft's official testing framework and is integrated directly into Visual Studio. Thanks to its stability and high compatibility with the Microsoft ecosystem, MSTest is often used in large enterprise projects or internal systems.

- **Functions / Features**
  - ✓ Provides a complete Unit Test structure through: [TestClass] – defining the test class, [TestMethod] – defining the test method, [DataTestMethod] and [DataRow] – data-driven testing. Deeply integrated with Visual Studio Test Explorer, no additional plugins required.
  - ✓ Supports high-level testing, including: Unit Test, Integration Test, UI Test (when combined with WinAppDriver), Easy connection to Azure DevOps/Team Foundation Server.
- **Benefits**
  - ✓ Easiest to use for beginners thanks to built-in Visual Studio support.
  - ✓ No external libraries required → stable and suitable for enterprise environments.
  - ✓ Supports data-driven testing, allowing testing of various datasets.
  - ✓ Continuously maintained and updated by Microsoft.
  - ✓ Highly compatible with the Azure ecosystem.

11.2.Analyze the benefits and limitations of automated testing methods deployed in applications.

In the development process of the SIMS_FPT student information management system, automated testing methods are being applied, utilizing the NUnit framework and the Moq library, which are essential to ensure the system's stability and reliability. We will use a combination of unit testing (to isolate logic) and interaction testing (to verify communication between components). Below is an analysis of these methods based on the actual implementation in the project.

a, For Unit Test

- **Benefits:** Early error detection: For each unit test, it allows for immediate detection of logical errors during the Controller implementation process, without needing to run the entire web application. This helps to pinpoint the problem to a specific method.

- For example, in LoginController, we will implement a test case to verify that the system correctly rejects invalid login credentials. As shown in the code snippet below, the test confirms that when

Login(email, password) fails, the View must return a specific error message: "Invalid email or password!".

```
62          // TC01: Verify Account Authentication (Invalid Login)
63          [Test]
            0 references
64          public async Task TC01_Login_WithInvalidCredentials_ReturnsViewWithError()
65          {
66              // Arrange
67              string email = "wrong@test.com";
68              string password = "wrong";
69              _mockUserRepo.Setup(r => r.Login(email, password)).Returns((Users)null);
70              var result = await _controller.Login(email, password) as ViewResult;
71              // Assert
72              Assert.That(result, Is.Not.Null, "Result should be a ViewResult");
73              Assert.That(result.ViewData["Error"], Is.EqualTo("Invalid email or password!"));
74          }
75          // TC07: User Role Authorization Check (Redirect Logic)
76          [Test]
```

Figure 11.3 Unit test ensuring invalid login attempts are handled correctly.

- **Improved code quality and modularity:** Writing unit tests forces the code to adhere to the Dependency Injection (DI) model. To test LoginController, we cannot use a direct database connection; instead, we must inject interfaces such as IUserRepository and IAuthenticationService. This makes the source code more loosely coupled and easier to maintain.

```
21          public void Setup()
22          {
23              // 1. Mock Repository
24              _mockUserRepo = new Mock<IUserRepository>();
25              _controller = new LoginController(_mockUserRepo.Object);
26
27              // 2. Mock HttpContext and ServiceProvider
28              var mockHttpContext = new Mock<HttpContext>();
29              var serviceProvider = new Mock<IServiceProvider>();
30
31              // --- A. Mock AuthenticationService (Required for Login/Logout) ---
32              var authService = new Mock<IAuthenticationService>();
33              serviceProvider.Setup(s => s.GetService(typeof(IAuthenticationService)))
34                             .Returns(authService.Object);
35
36              // --- B. Mock ITempDataDictionaryFactory (Required for Views/TempData) ---
37              var tempDataFactory = new Mock<ITempDataDictionaryFactory>();
38              var tempData = new Mock<ITempDataDictionary>();
39              tempDataFactory.Setup(f => f.GetTempData(It.IsAny<HttpContext>()))
40                             .Returns(tempData.Object);
41              serviceProvider.Setup(s => s.GetService(typeof(ITempDataDictionaryFactory)))
42                             .Returns(tempDataFactory.Object);
43
```

Figure 11.4 Dependency Injection and Mock Setup in Unit Tests.

- **Disadvantages**: Setup complexity: As can be seen in the figure, setting up test environments for a Controller that depends on HttpContext, TempData, and AuthenticationService is very verbose and complex. It requires significant effort to model all framework-level dependencies before writing the actual test logic.

b, For Integration and Interaction Test

In the context of SIMS_FPT, we use Interaction Testing (via MOQ) to verify that the different layers of the system (Controller, Service, Repository) communicate correctly with each other.

- **Benefits**: Validates business logic workflows these tests ensure that a sequence of operations performs as expected.
- **Example:** The GradingService involves complex logic: verifying a submission exists, saving the new grade, and updating the assignment status. The test TC05 verifies this entire workflow by checking if SaveSubmission is called with the specific grade (8.5) and feedback.

```
33          // TC05: Grade Calculation (Processing/Saving)
34          [Test]
            0 references
35          public void TC05_ProcessGrades_UpdatesSubmissionRepository()
36          {
37              // Arrange
38              var model = new BulkGradeViewModel
39              {
40                  AssignmentId = "ASM01",
41                  IsPublished = true,
42                  StudentGrades = new List<StudentGradeItem>
43                  {
44                      new StudentGradeItem { StudentId = "ST01", Grade = 8.5, Feedback = "Good job" }
45                  }
46              };
47              _assignRepo.Setup(r => r.GetById("ASM01")).Returns(new AssignmentModel());
48              _subRepo.Setup(r => r.GetByStudentAndAssignment("ST01", "ASM01"))
49                      .Returns(new SubmissionModel { StudentId = "ST01", AssignmentId = "ASM01" });
50              // Act
51              _service.ProcessGrades(model);
52              // Assert
53              // Verify SaveSubmission was called with grade 8.5
```

```
53              // Verify SaveSubmission was called with grade 8.5
54              _subRepo.Verify(r => r.SaveSubmission(It.Is<SubmissionModel>(
55                  s => s.StudentId == "ST01" &&
56                       s.Grade == 8.5 &&
57                       s.TeacherComments == "Good job"
58              )), Times.Once);
59
60              _assignRepo.Verify(r => r.Update(It.Is<AssignmentModel>(a => a.AreGradesPublished == true)), Times.
61          }
62      }
63  }
```

Figure 11.5 Verifying the Grading Business Logic Workflow.

- **Detects Interface and Data Flow Issues:** It ensures that data passed from the Controller reaches the Repository correctly.

⇨ **Example:** In ClassController, when adding students to a class, the test ensures that the StudentClassRepository.Add() method is triggered exactly once for each selected student. This confirms the loop logic in the controller is functioning correctly.

```
35      // TC04: Course Enrollment Verification
36      [Test]
        0 references
37      public void TC04_AddStudentsToClass_CreatesEnrollmentRecords()
38      {
39          // Arrange
40          string classId = "SE1601";
41          var selectedStudents = new List<string> { "ST01", "ST02" };
42
43          // Setup mocks
44          _studentClassRepo.Setup(r => r.IsEnrolled(classId, It.IsAny<string>())).Returns(false);
45          _classRepo.Setup(r => r.GetById(classId)).Returns(new ClassModel { ClassId = classId });
46          _studentClassRepo.Setup(r => r.GetByClassId(classId)).Returns(new List<StudentClassModel>());
47
48          // Act
49          var result = _controller.AddStudentsToClass(classId, selectedStudents) as RedirectToActionResult;
50
51          // Assert
52          Assert.That(result.ActionName, Is.EqualTo("ManageStudents"));
53
54          _studentClassRepo.Verify(r => r.Add(It.Is<StudentClassModel>(s => s.ClassId == classId && s.Student
55          _studentClassRepo.Verify(r => r.Add(It.Is<StudentClassModel>(s => s.ClassId == classId && s.Student
56      }
57  }
```

Figure 11.6 Verifying Data Flow between Controller and Repository.

- Disadvantages: Complexity in state simulation to test interactions, we must meticulously simulate the state of related objects. For instance, testing the grading service required mocking both the AssignmentRepository and SubmissionRepository to return specific data objects before the test could run.

c, Application Strategy for SIMS_FPT

To apply automated testing effectively in the SIMS_FPT project, we adopted the following structured approach:

- **Isolate and Test Validation Logic (Unit Testing):** We focus on testing input validation and Model State verification. This ensures that invalid data never reaches the service layer.
  - ✓ Strategy: Check ModelState.IsValid within the Controller tests.
  - ✓ Implementation: In the StudentController, we test that adding a student with an invalid email format prevents the service call and returns the View with errors.

```
29          // TC02: Add New Student with Data Validation
30          [Test]
            0 references
31          public async Task TC02_Add_WithInvalidModel_ReturnsViewAndDoesNotCallService()
32          {
33              // Arrange
34              var invalidModel = new StudentCSVModel { StudentId = "ST001", Email = "invalid-email" };
35
36              // Simulate Validation Error
37              _controller.ModelState.AddModelError("Email", "Invalid email format");
38
39              // Act
40              var result = await _controller.Add(invalidModel) as ViewResult;
41
42              // Assert
43              Assert.That(result, Is.Not.Null);
44              Assert.That(result.Model, Is.InstanceOf<StudentCSVModel>());
45              Assert.That(_controller.ModelState.IsValid, Is.False);
46          }
47
```

Figure 11.7 Testing Data Validation Strategy in StudentController.

- **Verify Business Rules and Side Effects (Interaction Testing):** For critical operations like deleting data or grading, we focus on verifying that the correct repository methods are invoked.
  - ✓ Strategy: Use Moq.Verify to count the number of times a dependency method is called.
  - ✓ Implementation: As shown in the SubjectController, we verify that the Delete() action calls the repository's Delete() method exactly once, ensuring data integrity.

```
24          // TC03: Delete Course by ID
25          [Test]
            0 references
26          public void TC03_DeleteSubject_CallsRepositoryDelete()
27          {
28              // Arrange
29              string subjectId = "SUB001";
30
31              // Act
32              var result = _controller.Delete(subjectId) as RedirectToActionResult;
33
34              // Assert
35              Assert.That(result.ActionName, Is.EqualTo("List"));
36              _mockRepo.Verify(r => r.Delete(subjectId), Times.Once);
37          }
38      }
39  }
```

Figure 11.8 Verifying Critical Business Actions (Deletion).

d, Test Results and Evaluation

After implementing the unit and interaction tests described above, the entire test suite was executed using the Visual Studio Test Explorer to validate the system's stability.

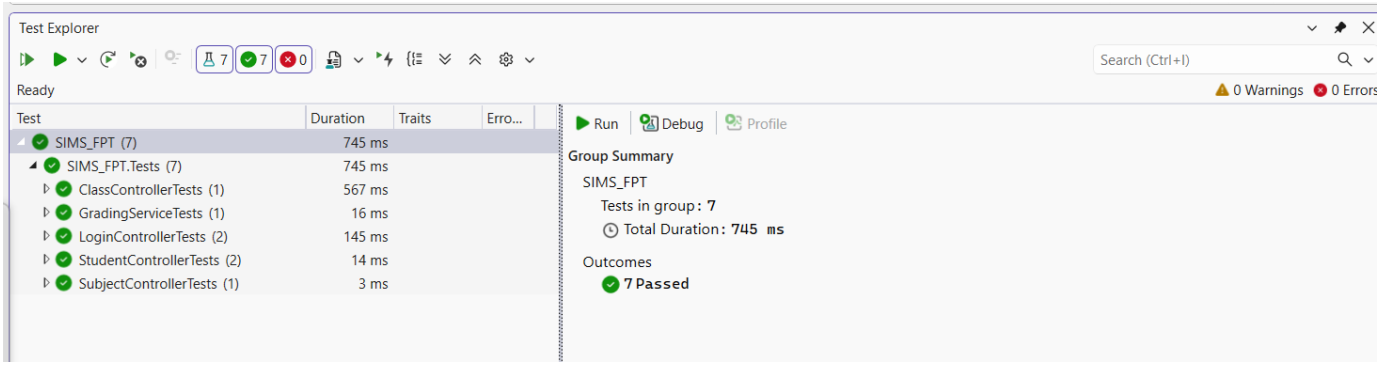


Figure 11.9 Test execution results showing 100% pass rate.

- ✓ Total Tests: 7 critical test cases covering Controllers (Class, Login, Student, Subject) and Services (Grading).
- ✓ Outcome: All 7 tests passed (indicated by the green checkmarks), confirming that the core functionalities such as Login validation, Student enrollment, and Grading logic are functioning according to specifications.
- ✓ Performance: The total execution time was approximately 745ms, demonstrating that the automated tests act as a rapid feedback loop for development.

⇨ **Conclusion**: The successful execution of these automated tests creates high confidence in the quality of the SIMS_FPT application. This confirms that the integration of NUnit and Moq has effectively protected the system from logic errors and regression issues during development.

12.Evaluate the impact of SOLID development principles on object-orientated application. (D1)

Each SOLID principle is a fundamental guide in object-oriented software engineering, which in turn helps improve code quality, flexibility, and maintainability. Each assessment of their impact on object-oriented application development processes provides insight into how to build modular, testable, and extensible systems. In the context of Student Information Management System (SIMS), each application of SOLID principles has played a crucial role in shaping a clean, maintainable, and extensible architecture.

- **Evaluation of SOLID Principles' Impact**

- **Single Responsibility Principle (SRP)**

✓ With the single responsibility principle, it is strongly suggested that each class should have only one reason to change from here on out, meaning it should be focused on a very single function. In traditional OOP systems that do not follow SRP, classes often handle multiple responsibilities—such as data management, business logic, and presentation—which increases coupling and complicates maintenance.

- ✓ In SIMS, SRP is applied to each through separation of concerns for classes like StudentService, AuthService, and CourseRepository, where each class handles a very separate concern. For example, StudentService would manage business logic related to students, while CsvDataProvider would only be responsible for data access operations. This separation also helps to increase maintainability and makes debugging or updating more efficient. It also supports parallel development, as multiple developers can work independently on different modules without affecting each other's code.
- ✓ Finally, SRP improves cohesion and reduces code duplication, making the SIMS system more adaptable to future changes (Martin, 2003).

- **Open/Closed Principle (OCP)**
- The open/closed principle states that each software entity should be open for extension but closed for modification. This ensures that new features can be added without changing existing code, thus minimizing the risk of regression errors.
- In SIMS, these principles are reflected in the design of the repository interface and the services. For example, IRepository<T> can be extended to support new entities such as faculty or admissions without modifying the logic of the existing repository. Similarly, adding a new service (such as NotificationService) does not affect the existing StudentService or AuthService implementation.
- By adhering to OCP, SIMS achieves better scalability and future adaptability, ensuring that new requirements—such as integrating SQL databases instead of CSV files—can be implemented with minimal disruption to the system's core architecture.

- **Liskov Substitution Principle (LSP)**
- With each Liskov substitution principle, it is guaranteed that subclasses can be replaced with their parent classes without affecting the functionality of the systems. This principle protects the system of each inheritance hierarchy and also ensures behavioral consistency.
- In SIMS, each entity such as Admin, Student and Lecturer inherits from the User class, but they can be used interchangeably when a User object is needed. With each design, it allows flexible polymorphism and reduces redundant code. For example, the authentication mechanism can

authenticate all types of users through the same interface without needing to have specialized methods for each subclass.

- LSP compliance will be enhanced with each role enabling code reuse and from here it will also be supported with each role polymorphic behavior, thus improving consistency in the system's authentication and access control logic.

- **Interface Segregation Principle (ISP)**

- The Interface Segregation Principle encourages the creation of small, specific interfaces rather than large, general ones. It prevents each class from being forced to implement methods that will not be used.
- In SIMS, ISP is explicitly expressed through the use of centralized interfaces such as IRepository<T>, IUserService, and ICsvProvider. Each interface defines the operations related to its responsibility. For example, ICsvProvider will only handle reading and writing CSV data, while IRepository<T> focuses on CRUD operations. This design makes the code easier to test, simulate, and extend.
- Through ISP, the system achieves greater modularity and avoids unnecessary dependencies, resulting in cleaner and more maintainable code (Martin, 2008).

- **Comparative Evaluation**

- Before adopting SOLID, traditional object-oriented designs often suffered from tightly coupled classes, duplicated logic, and inflexible architectures. Changing one module often caused ripple effects in other areas, increasing maintenance costs.
- In contrast, SOLID principles allowed SIMS to maintain a clear separation of concerns across the presentation, business logic, and data access layers. For example, when migrating from CSV storage to SQL, only the Repository and Provider layers needed to be modified, while the upper layers remained the same. This demonstrates that SOLID promotes stability, adaptability, and scalability—essential qualities for enterprise-grade applications (Meyer, 2020).
- Furthermore, SOLID compliance allows for more seamless integration of design patterns like Repository, Factory, and Singleton, reinforcing the consistency and maintainability of the software architecture.

- **Critical Reflection**

- While SOLID principles offer significant long-term benefits, they also introduce initial complexity into project design. New developers may have difficulty grasping the concepts of abstraction and dependency injection. Excessive layering can lead to overdesign in smaller systems. However, in

large-scale systems like SIMS, this complexity is justified as it ensures long-term sustainability and reduces technical debt.

- The disciplined application of SOLID principles encourages a culture of clean architecture, modular design, and professional coding standards – key characteristics that align with industry best practices and modern software development methodologies (IBM, 2025).

13.Analyse the benefits and drawbacks of different forms of automatic testing of applications and software systems, with examples from the developed application. (D2)

In the development of the Student Information Management System (SIMS), we implemented two primary forms of automated testing: Unit Testing and Integration (Interaction) Testing. Below is a critical analysis of these methods, supported by specific code examples from the developed application.

13.1.Analysis of Unit Testing

Unit testing focuses on verifying the smallest testable parts of the application, such as individual methods in Controllers or Services, in isolation from external dependencies.

a, Benefits

- **Immediate Feedback & Logic Isolation:** Unit tests run extremely fast (milliseconds) and pinpoint the exact location of a bug.
  - ✓ Evidence in SIMS: In the StudentControllerTests, we validated the logic for adding a new student. The test TC02_Add_WithInvalidModel confirms that if the input model is invalid (e.g., wrong email format), the system immediately returns an error View without attempting to call the service layer. This isolation prevents bad data from ever reaching the core business logic.

```
29          // TC02: Add New Student with Data Validation
30          [Test]
            0 references
31          public async Task TC02_Add_WithInvalidModel_ReturnsViewAndDoesNotCallService()
32          {
33              // Arrange
34              var invalidModel = new StudentCSVModel { StudentId = "ST001", Email = "invalid-email" };
35
36              // Simulate Validation Error
37              _controller.ModelState.AddModelError("Email", "Invalid email format");
38
39              // Act
40              var result = await _controller.Add(invalidModel) as ViewResult;
41
42              // Assert
43              Assert.That(result, Is.Not.Null);
44              Assert.That(result.Model, Is.InstanceOf<StudentCSVModel>());
45              Assert.That(_controller.ModelState.IsValid, Is.False);
46          }
```

Figure 13.1 Unit test verifying validation logic for invalid inputs.

- **Enforces Loosely Coupled Architecture:** To make unit testing possible, the code must follow the Dependency Injection (DI) pattern.
- **Evidence in SIMS:** The LoginController does not instantiate UserRepository directly. Instead, it accepts an IUserRepository interface in its constructor. This allowed us to pass a "Mock" repository during testing, proving that the system is modular and maintainable.

b, Drawbacks

❖ High Setup Complexity (Mocking Overhead): Testing a simple method often requires writing significantly more lines of setup code than the actual logic being tested.

- Evidence in SIMS: As shown in Figure, to test a simple Login action, we had to mock HttpContext, TempData, UrlHelper, and AuthenticationService. This complexity increases the initial development time and learning curve for new developers.

```
21      public void Setup()
22      {
23          // 1. Mock Repository
24          _mockUserRepo = new Mock<IUserRepository>();
25          _controller = new LoginController(_mockUserRepo.Object);
26
27          // 2. Mock HttpContext and ServiceProvider
28          var mockHttpContext = new Mock<HttpContext>();
29          var serviceProvider = new Mock<IServiceProvider>();
30
31          // --- A. Mock AuthenticationService (Required for Login/Logout) ---
32          var authService = new Mock<IAuthenticationService>();
33          serviceProvider.Setup(s => s.GetService(typeof(IAuthenticationService)))
34                         .Returns(authService.Object);
35
36          // --- B. Mock ITempDataDictionaryFactory (Required for Views/TempData) ---
37          var tempDataFactory = new Mock<ITempDataDictionaryFactory>();
38          var tempData = new Mock<ITempDataDictionary>();
39          tempDataFactory.Setup(f => f.GetTempData(It.IsAny<HttpContext>()))
40                         .Returns(tempData.Object);
41          serviceProvider.Setup(s => s.GetService(typeof(ITempDataDictionaryFactory)))
42                         .Returns(tempDataFactory.Object);
43
```

Figure 13.2 High setup complexity involved in mocking external dependencies using Moq.

- **Limited Scope:** Passing a unit test does not guarantee the feature works for the end-user. For example, a unit test might pass because the Mock repository returns perfect data, but the actual SQL query in production might fail due to connection issues.

13.2.Analysis of Integration (Interaction) Testing

Integration testing (specifically Interaction Testing in this context) focuses on verifying that different layers (Controller, Service, Repository) communicate correctly to complete a business workflow.

a, Benefits

- ❖ **Validates End-to-End Workflows:** These tests ensure that data flows correctly from the business layer down to the data access layer.
  - ✓ Evidence in SIMS: The GradingServiceTests validates a critical workflow: A teacher submits grades -> The service calculates results -> The repository persists the data. The test TC05 confirms that the SaveSubmission method is actually called with the correct grade (8.5), ensuring the business rule is preserved across layers.

```
34          [Test]
            0 references
35          public void TC05_ProcessGrades_UpdatesSubmissionRepository()
36          {
37              // Arrange
38              var model = new BulkGradeViewModel
39              {
40                  AssignmentId = "ASM01",
41                  IsPublished = true,
42                  StudentGrades = new List<StudentGradeItem>
43                  {
44                      new StudentGradeItem { StudentId = "ST01", Grade = 8.5, Feedback = "Good job" }
45                  }
46              };
47              _assignRepo.Setup(r => r.GetById("ASM01")).Returns(new AssignmentModel());
48              _subRepo.Setup(r => r.GetByStudentAndAssignment("ST01", "ASM01"))
49                      .Returns(new SubmissionModel { StudentId = "ST01", AssignmentId = "ASM01" });
50              // Act
51              _service.ProcessGrades(model);
52              // Assert
53              // Verify SaveSubmission was called with grade 8.5
54              _subRepo.Verify(r => r.SaveSubmission(It.Is<SubmissionModel>(
55                  s => s.StudentId == "ST01" &&
56                       s.Grade == 8.5 &&
57                       s.TeacherComments == "Good job"
58              )), Times.Once);
59
60              _assignRepo.Verify(r => r.Update(It.Is<AssignmentModel>(a => a.AreGradesPublished == true)), Times.
61          }
62      }
63  }
```

Figure 13.3 Interaction test verifying the data flow between Service and Repository layers.

- **Data Integrity Verification:** It ensures that side effects (like updating a status or creating multiple child records) happen as expected.
  - ✓ **Evidence in SIMS:** In ClassControllerTests, the test TC04 verifies that enrolling 2 students triggers exactly 2 distinct calls to the repository's Add method. This catches logic errors (such as "off-by-one" loops) that simple unit tests might miss.

b, Drawbacks

- ❖ State Management Complexity: Simulating a realistic environment is difficult.

- ❖ Evidence in SIMS: To test the grading logic, we had to artificially construct a complex object graph where a Student exists, an Assignment exists, and a Submission exists. If any of this setup data is incorrect, the test fails even if the code logic is correct.

13.3.Comparative Evaluation & Application Results

Based on the implementation experience in SIMS, we can draw a critical comparison between the two methods. The table below summarizes the trade-offs observed:

Criteria	Unit Testing (e.g., LoginController)	Integration/Interaction Testing (e.g., GradingService)
Execution Speed	Extremely Fast (< 10ms). Runs entirely in memory with mocked dependencies.	Slower. Requires setting up complex object graphs and simulating data flow.
Isolation Level	High. If the test fails, we know exactly which method is broken.	Low. If the test fails, the bug could be in the Service, the Repository, or the mapping between them.
Maintenance Cost	High. Tightly coupled to implementation details. Changing a constructor requires updating all test setups.	Medium. Focuses on behavior/outcome. Refactoring internal code usually doesn't break these tests.
Reliability	Theoretical. Proves logic is correct assuming external systems work perfectly.	Practical. Proves that the business process completes successfully.
Best Used For	Validation logic, Calculation rules, Edge cases.	Critical business transactions (Grading, Enrollment), Data persistence.

Table 13.1 Critical Comparison of Testing Methods in SIMS

Strategic Conclusion: In the SIMS application, relying solely on one method would be insufficient.

- ✓ Reliance on Unit Tests alone would lead to a "false sense of security," where components work individually but fail when connected.
- ✓ Reliance on Integration Tests alone would make debugging extremely difficult and slow down the development cycle.

Therefore, the SIMS project adopted a Hybrid Strategy: strict Unit Tests were used to guard against invalid inputs (as seen in StudentController), while Interaction Tests were reserved for high-value business transactions (as seen in GradingService).

CONCLUSION

This paper presented a comprehensive analysis of the core principles of object-oriented software development (OOP) and applied them to the design of a large-scale Student Information Management System (SIMS).

The key points are summarized as follows:

- OOP and SOLID Principles: The paper investigated the characteristics of OOP, including the relationships between classes and the four main pillars (Encapsulation, Abstraction, Inheritance, Polymorphism). In particular, the strict application of SOLID principles was shown to be the foundation for the architecture of SIMS.
- SOLID principles ensure the system is highly modular, scalable (OCP) and easy to maintain, with clear separation of concerns (SRP) between the Service and Repository layers.
- Clean Code and Impact: Clean Code techniques, such as meaningful naming and high modularity, were explained as key to improving readability, testability, and team efficiency in writing algorithms and handling data structures.
- SIMS Architecture Design: The SIMS system was designed based on a Three-Tier Architecture (Presentation, Business Logic, Data Access), using ASP.NET Core.
- This design uses Use Case Diagram, Class Diagram, and Package Diagram to clearly illustrate the structure, ensuring separation of responsibilities and compliance with SOLID principles (DIP) through interfaces.
- Design Patterns and Refinement: The report analyzed the Creational, Structural, and Behavioral design patterns. Furthermore, the SIMS architecture was refined to incorporate multiple patterns, including Repository, Singleton, Factory Method, Observer, and Facade. This integration further strengthened the system's flexibility, reusability, and testability.
- Testing Process: A proper testing regime was designed, covering Unit Test, Integration Test, and System Test levels, with a strong focus on Automated Testing using tools such as xUnit and Moq. This ensured the functional correctness, performance, and reliability of the SIMS application.
- In short, the SIMS project demonstrates that disciplined application of SOLID principles and proven design patterns is critical to building a robust, scalable, and maintainable enterprise-grade system that effectively meets functional and non-functional requirements.

REFERENCES

GeeksforGeeks (2020). Introduction of Object Oriented Programming. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/dsa/introduction-of-object-oriented-programming/.

Codacy (2023). What Is Clean Code? A Guide to Principles and Best Practices. [online] blog.codacy.com. Available at: https://blog.codacy.com/what-is-clean-code.

IBM (2025). Defining use cases. [online] Available at: https://www.ibm.com/docs/en/product-master/12.0.0?topic=processes-defining-use-cases.

Service, I. (2020). Class Diagrams. [online] Improvement Service. Available at: https://www.improvementservice.org.uk/business-analysis-framework/define-requirements/class-diagrams.

SmartBear (2019). *What is Automated Testing | Software Testing Basics*. [online] Smartbear.com. Available at: https://smartbear.com/learn/automated-testing/what-is-automated-testing/.