**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** | 19/6/2025 | **Date Received 1st Submission** | 6/8/2025 |
| **Re-submission Date** |  | **Date Received 2nd Submission** |  |
| **Student Name** | TRAN NGUYEN VAN MEN | **Student ID** | BS00545 |
| **Class** | SE07101 | **Assessor name** | NGUYEN VAN QUANG |
| **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 wer responsibility to ensure that we understand correct referencing practices. As a university level student, we are expected to use appropriate references throughout and keep carefully detailed notes of all wer sources of materials for material we have used in wer work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or wer course tutor if we 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** | Men |

**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** |
| Tran Nguyen Van Men |  |  |
| Doan Thach Que Duong |  |  |
| Vo Nguyet Nhi |  |  |
|  |  |  |
|  |  |  |
| **Class** | SE07101 | **Assessor name** | NGUYEN VAN QUANG |
| **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 wer responsibility to ensure that we understand correct referencing practices. As a university level student, we are expected to use appropriate references throughout and keep carefully detailed notes of all wer sources of materials for material we have used in wer work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or wer course tutor if we 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:** |  | | | | | |
| **r Summative Feedback: r Resubmission Feedback:** | | | | | | |
| **Grade:** | | | **Assessor Signature:** | | | **Date:** |
| **Internal Verifier’s Comments:** | | | | | | |
| **Signature & Date:** | | | | | | |

# Table of Content

# Table of Figures

# Table of Table

1. ****INTRODUCTION****
2. ****MAIN BODY****
3. ****P1 Investigate the characteristics of the object- orientated paradigm, including class relationships and SOLID principles.****
   1. ****Definition OOP****

* **Object-oriented programming (OOP) is a programming approach based on objects – entities that contain data (attributes) and behavior (methods). Instead of writing a program as a single string of instructions, OOP divides the program into many objects like in real life. Each object can represent people, things, events**
  + ****Class**** is a template that describes the properties and behaviors used to create an object.
  + ****The object**** is a concrete instance of a class. The actual data stored in the object.
  + ****Attributes/ properties**– Properties is a variable inside a class, describing the characteristics of an object.**
  + ****Constructor**** is a special function, with the same name as the class, called when initializing an object to assign an initial value.
  + **Method:** Object behavior – functions that describe what the object can do **.**

****A computer screen shot of a program

AI-generated content may be incorrect.****

Figure 1\_Explain about Method, Properties, Class, Constructor for OOP

* **OOP has 4 main features:**
  + **Encapsulation: Data and data processing functions are encapsulated in an object. Helps protect data, allowing access only through predefined methods.**

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 2\_Example Encapsulation of OOP

* + **Inheritance: A child class can inherit properties and methods from the parent class. Help reuse code and extend the system easily.**

A computer screen with text

AI-generated content may be incorrect.

Figure 3\_Example Inheritance of OOP

* + **Polymorphism: Polymorphism allows the same method to behave differently depending on the specific object type, but we still handle it through the same type.**

****A screenshot of a computer program

AI-generated content may be incorrect.****

Figure 4\_Example Polymorphism and overriding of OOP

* + - Overriding is the act of overriding a parent class method in a child class, using the keyword override. The parent class method must be marked virtual or abstract.
    - Overloading is defining multiple methods with the same name in the same class, but with different parameter lists (number, type, or order of parameters).

A computer screen shot of a black screen

AI-generated content may be incorrect.

Figure 5\_Overloading Code example

* + **Abstraction: Show only what is necessary, hide the details inside, it like a contract, defines methods that every subclass must have (no need to know the specifics).**

****A screen shot of a computer program

AI-generated content may be incorrect.****

Figure 6\_Abstraction of OOP

* An interface is a data type that declares methods (and possibly properties, events) without a body, like abstract methods. When a class implements this interface, it must fully implement all the members declared in the interface. C# allows multiple inheritance of interfaces, but not multiple inheritance of classes.

A computer screen shot of a black screen

AI-generated content may be incorrect.

Figure 7\_Interface

* 1. ****Advantages and disadvantages of OOP****

Table 1\_Advantages and disadvantages of OOP

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Advantages** | **Disadvantages** |
| Source Code Organization (Modularity) | Break the program into layers, easy to manage, develop and extend. | It is easy to create too many layers, causing confusion if not managed well. |
| Code Reusability | Easy to reuse classes, inheritance saves time and effort. | Can lead to tight coupling if inheritance is not done properly. |
| Encapsulation | Keep data secure, limit access with access modifiers (private, protected). | Can hide useful information if misused. |
| Abstraction | Hide unnecessary details, helping to focus on the main function. | It is difficult to design at the right level of abstraction, which can easily lead to confusion. |
| Polymorphism | A generic method can have different behavior depending on the specific object. | New learners can get confused when many classes have the same methods. |
| Maintainability | Code is easy to edit, update, and does not affect other parts much. | If the initial design is poor, expansion becomes very difficult. |
| Testability | We can test each layer separately, easy to find errors in small blocks. | Testing can be complicated if classes depend on each other too much. |
| Performance | — | Performance may be reduced due to features like virtual, dynamic binding,… |
| Level of complexity | — | Requires object design thinking, inheritance, abstraction, etc., which is difficult for beginners. |
| System size | Suitable for large, complex systems that need long-term expansion. | Not suitable for small, simple applications – causes structural excess. |

* 1. ****UML and Class Diagram****
* **UML (Unified Modeling Language) is a standard modeling language used to design, describe, visualize, and document software systems.**
  + **UML is not a programming language, but a visual language (using diagrams) to describe the architecture and behavior of a system.**
  + **UML is used in requirements analysis, system design, and communication between teams in the software development process.**
  + **UML includes many types of diagrams, divided into 2 main groups:**

Table 2\_Type UML diagram

|  |  |  |
| --- | --- | --- |
| **Diagram group** | **Purpose** | **For example** |
| **Structural diagrams** | System structure description | Class Diagram, Object Diagram, Component Diagram |
| **Behavioral diagrams** | Describe behavior, data flow | Use Case, Sequence Diagram, Activity Diagram |

* **Class Diagram is a type of UML diagram, used to describe the structure of an OOP system through classes, attributes, methods, and relationships between classes.**

**A screen shot of a black and white screen

AI-generated content may be incorrect.**

Figure 8\_Class Diagram

* 1. ****Class Relationships****
* **Association: A class it knows about others but does not own or manage the lifecycle of that object. It means this class is deleted, another still exists, and vice versa. The relationship can be one-way or two-way, and is flexible, discrete, and not constrained.**
  + **Simple Association: It is a relationship in which only one class knows about the other class, it means one-way.**

**A computer screen shot of a black screen

AI-generated content may be incorrect.**

Figure 9\_Simple Association

* + **Bidirectional Association: It is a relationship in which both classes know about each other and can access each other. It is often used when both objects need to interact and reference each other.**

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 10\_Bidirectional Association

* Dependency is a weak, temporary relationship in which one class uses another class as a temporary object, parameter, or method call.

A computer screen shot of a computer

AI-generated content may be incorrect.

Figure 11\_Dependency

* Inheritance is a mechanism in object-oriented programming that allows a subclass to inherit the attributes and methods of a superclass, thereby reusing code and extending functionality.

A black grid with white text

AI-generated content may be incorrect.

Figure 12\_Inherit

* Aggregation is a special form of association, representing a "whole part" relationship, in which the parent object loosely owns the child object. The child object can exist independently of the parent object; **the child objects can still exist independently** if the parent object **is destroyed.**

A screenshot of a computer

AI-generated content may be incorrect.

Figure 13\_Aggregation

* Composition is a stronger form of aggregation, where the parent object fully owns and is responsible for the lifecycle of the child object. If the **parent object is destroyed**, the child object is **also destroyed.**

A screen shot of a computer

AI-generated content may be incorrect.

Figure 14\_Composition

* 1. **\*\*SOLID Principles:**
     1. **Open/Closed Principle**
* A class should be open for extension but closed for modification. It means we can extend the functionality of a class without changing its original source code.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 15\_This class is not following the Open/Closed Principle. When extend class, must fix original code

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 16\_Apply OCP in this example

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 17\_Extend from Interface. Add Attributes of Class Student

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 18\_Run Program OCP in Main and Result

* + 1. **Single responsibility principle (SRP):**
* A module should be responsible to one, and only one, actor. **[Martin, R.C. (2017).]**, meaning each class should have only one responsibility in the system.

A computer screen shot of a computer code

AI-generated content may be incorrect.

Figure 19\_Class Dog combines multiple responsibilities, so it violates the SRP

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 20\_Applying the Single Responsibility Principle

* Conclusion: SRP helps Separate the responsibilities of each class. This helps clear code, easy-to-understand, easy-to-maintain code, easy to extend when we want to add new behaviour, reuse and testing are simpler.
  + 1. **Dependency Inversion Principle**
* **Definition:**
  + "High-level modules should not depend on low-level modules. Both should depend on abstractions."
  + "Abstractions should not depend on details. Details should depend on abstractions." **[Robert C. Martin]**
* **In Figure 14,** In the Main original function, the code is OCP compliant because it is possible to extend new bonus calculator classes without changing the old logic. The bonus calculator classes are also SRP compliant; each class only does one thing. **However**, Main it violates SRP because it does too much: it creates data, handles bonus logic, and prints results. In addition, it violates DIP because it depends directly on concrete classes instead of just using abstractions. This makes the code difficult to extend or test later.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 21\_Apply DIP in this example. Create BonusManagerCalculator\_DIP class.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 22\_Main function after apply DIP

1. ****P2 Explain how clean coding techniques can impact on the use of data structures and operations when writing algorithms.****
   1. ****Clean Coding Techniques:****

* **Meaningful naming: Variable, function, and class names should clearly state their purpose. Don't skimp on words, and avoid generic names like x, data, temp, good names help us understand what they do without reading the entire block of code.**

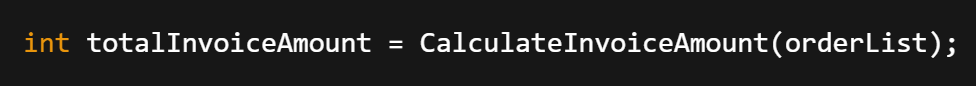
****

Figure 23\_Clear example: Knowing what this variable contains, and what that function does.

**A black background with white text

AI-generated content may be incorrect.**

Figure 24\_Unclear example: We must read inside the function Getto understand what it does.

* **Modularity: Don't cram everything into one big function. Instead, split it into several small functions, each doing just one thing. This helps: Easy to read, easy to reuse, easy to test, and easy to fix.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure 25\_Modularity

* Comment: Comment only, when necessary, especially when explaining why we wrote something, not just reiterating the code. Code should "speak for itself", while comments help us provide reason or context.

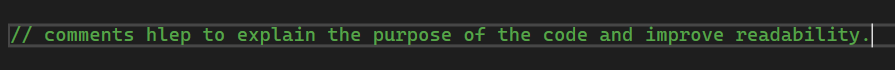


Figure 26\_Comment

* Consistency: Maintain the same coding style throughout the entire program. If code is consistency, readers won’t be "shocked" because they don't know which writing style to follow, it feels confusing, difficult to debug, and difficult to fix errors. Consistency includes:
  + Set to name variables, functions, classes
  + Set to use brackets and indentation
  + Logical writing style (naming, casing, comment...)
  + Error handling, conditional logic, code structure

A computer screen with white text

AI-generated content may be incorrect.

Figure 27\_Consistency

* 1. ****Impact on Data Structures and Operations:****
* **Clean code not only helps others understand the logic of the program, but also helps programmers manage data structures and operations (add, delete, search...) accurately, easily expandable and maintainable.** 
  + **Example: Dirty Code**

**A computer screen shot of code

AI-generated content may be incorrect.**

Figure 28\_Example Dirty Code

Table 3\_Expalain why That code is Dirty Code

|  |  |  |
| --- | --- | --- |
| **STT** | **Unclean problem** | **Explain** |
| 1 | **Non-compliance with packaging principles (Encapsulation)** | Variables name and grades let public, anyone can change without control. This violates OOP. |
| 2 | **Unclear variable name (s)** | Names “s” is meaningless. Reading the code, we have no idea what it contains unless we look closely. |
| 3 | **Class name “Main Class” is ambiguous** | Class name does not reflect role. Does not properly describe the task as the entry point to the program. |
| 4 | **Unnecessary nested classes** | “Main Class” located inside Dirty Code but not using anything from the outer class. Makes the code hard to read, not well organized. |
| 5 | **Lack of Separation of Responsibilities (SRP - Single Responsibility Principle)** | A class “Main Class” that handles both logic and information display, instead of being clearly divided (e.g., separated into “Student Printer”, “Student Analyzer”,…) |
| 6 | **Print logic directly in loop** | If we need to print HTML instead of Console in the future, we will have to change the logic. Clean code should separate data processing and display. |
| 7 | **No input data check** | Suppose the score is negative or exceeds 10, no validation controls. |
| 8 | **No comment or logical description** | Even if the code is short, if it expands to many lines, it will be difficult to understand without description. |

* + Clean Code:

A screen shot of a computer

AI-generated content may be incorrect.

Figure 29\_Clean code

* Conclusion: Impact on Data Structures and Operations is how code redesign makes data structures safer, clearer, easier to use, and improves program performance, scalability, and maintainability.

1. ****P3 Design a large data set processing application, utilizing SOLID principles, clean coding techniques and a design pattern.****
2. ****Usecase, Class Diagram, Package Diagram****

****A screenshot of a computer screen

AI-generated content may be incorrect.****

Figure 30\_Use case Diagram

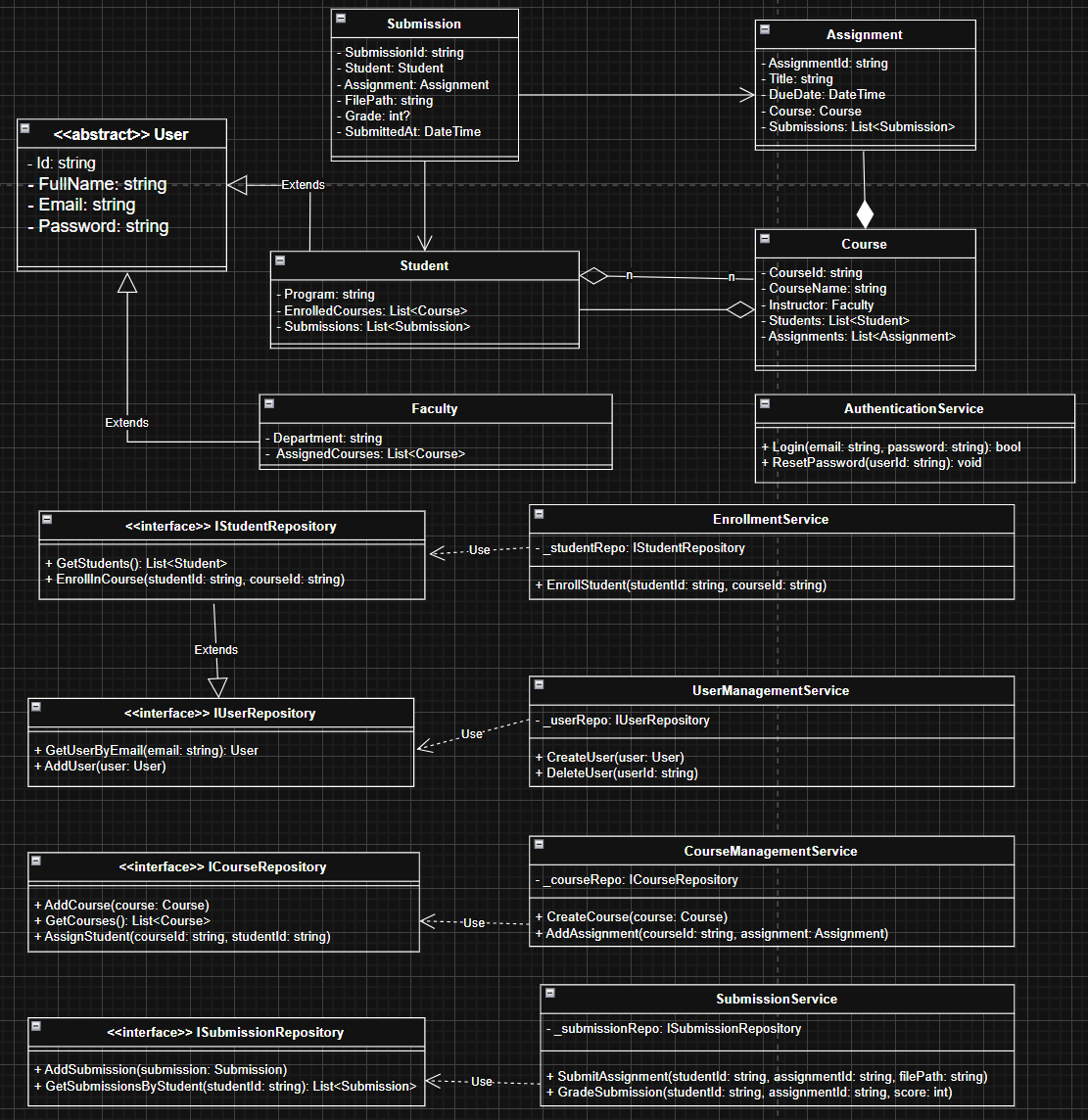


Figure 31\_Class Diagram

* This Class Diagram clearly shows how apply object orientation and 3 SOLID principles: SRP, OCP and DIP. The classes are neatly divided: entity manages data, service processes business, repository uses interface to reduce dependencies and easy to extend. The relationships between classes such as inheritance, composition, and dependency are all shown according to UML standards.

1. **An explanation of how SOLID principles have been applied in our team design.**

* In designing the Student Information Management System (SIMS), our team carefully applied key SOLID principles—SRP (Single Responsibility), OCP (Open/Closed), and DIP (Dependency Inversion)—throughout both the Use Case and Class Diagrams to ensure a maintainable, scalable, and clean object-oriented architecture.
  + SRP (Single Responsibility Principle) was applied by clearly separating functionalities across different classes and services. In our Use Case Diagram, each actor (Student, Faculty, Admin) was assigned distinct roles. This translated into the Class Diagram with service classes like StudentService, CourseService, and SubmissionService, each handling only one specific responsibility without overlapping logic.
  + OCP (Open/Closed Principle) guided our use of interfaces such as IStudentRepository and ICourseRepository. This allowed our system to be open for extension (e.g., switching from SQL to Firebase) without modifying the existing services. New implementations could be added without breaking existing logic, making the system more adaptable to future requirements.
  + DIP (Dependency Inversion Principle) was applied by having our services depend on abstractions rather than concrete classes. For example, StudentService communicates through IStudentRepository, not directly with any database class. This approach enhanced the system’s testability and reduced coupling between layers.

1. ****P4 Design a suitable testing regime for the application, including provision for automated testing.****

****4.1** Unit Testing**

* **Unit testing is testing each smallest function, method, or class of a system to ensure that each component functions as expected, independently of the rest.**

Table 4\_Applicable for SIMS

|  |  |  |
| --- | --- | --- |
| **Class to test** | **Method** | **Test Objective** |
| AuthenticationService | Login(), ResetPassword() | Correct information → login successful |
| EnrollmentService | EnrollStudent() | Register true/false |
| SubmissionService | SubmitAssignment(),GradeSubmission() | Processing submitted files and valid scoring |
| UserManagementService | CreateUser(), DeleteUser() | Create users, handle duplicate errors |

* **Automated testing tools :**
  + **xUnit/NUnit (C#)**
  + **Mocking framework : Mockito, Moq to mock repository**

****4.2 Integration Testing****

* **Integration testing is the testing of interactions between modules or classes to ensure that the components communicate properly when combined.**
* **Purpose:**
  + **Ensure data is passed correctly between layers/services.**
  + **Check the overall logic between sections.**

Table 5\_Applicable for SIMS

|  |  |  |
| --- | --- | --- |
| **Class** | **Method** | **Test content** |
| AuthenticationService | Login(email, password) | Login true/false |
| EnrollmentService | EnrollStudent(studentId, courseId) | Correct registration, error handling |
| SubmissionService | SubmitAssignment(...),GradeSubmission(...) | Submit and grade |
| UserManagementService | CreateUser(),DeleteUser() | Add & remove users |

****4.3 System Testing****

* **Definition: Test the entire system as a complete whole, from input to output.**
* **Target:**
  + **Ensure that the system correctly executes all functional requirements.**
  + **Includes interaction from UI/API to backend.**
* **Applicable for SIMS:**
* **Sample system testing flow:**
  + **Create student account**
  + **Log in to the system**
  + **Enrolled on the course**
  + **Submit your assignment**
  + **Instructor grading**

****4.4 User Acceptance Testing (UAT)****

* **Definition: UAT is the phase where end users (students, lecturers, admins) test to ensure the system meets actual requirements.**
* **Target:**
  + **Confirm that the system is easy to use and fully functional.**
  + **Increase the possibility of practical implementation.**

****4.5 Automated Testing****

* **Definition: Automate the execution of test cases using tools to ensure efficiency, speed and accuracy in software testing.**
* **Target:**
  + **Minimize human error in manual testing.**
  + **CI/CD support – automatically test every code update.**

1. ****P5 Build a large dataset processing application based on the design produced.****
2. ****P6 Examine the different methods of implementing automatic testing as designed in the test plan.****
3. ****P7 Implement automatic testing of the developed application.****
4. ****M1 Analyze, with examples, each of the creational, structural and behavioral design pattern types.****
   1. ****Definition****

* **A design pattern is a general, proven solution to recurring problems in software design. It is not a specific piece of code, but a way to organize classes, objects, and their relationships in a way that is flexible, extensible, reusable, and maintainable.**
  + **Simple understand, like when building a house, people have many sample designs to build a 1-storey house, 2-storey house, villa, apartment... depending on the situation, choose the appropriate model. In programming, each design pattern is a "blueprint" to handle a familiar situation.**
  + **In programming, each design pattern is a "blueprint" to handle a familiar situation.**
* **Main groups:**
  + **Creational – Relating to how objects are created.**
  + **Structural – How classes and objects are organized to create larger structures.**
  + **Behavioral – How subjects communicate and assign responsibilities.**

Table 6\_Different types of patterns

|  |  |  |
| --- | --- | --- |
| **Type** | **Focus on what?** | **The question it addresses** |
| Creational | How to **create** an object | How to create objects flexibly and effectively? |
| Structural | How to **connect** classes together | How to connect classes to make them easy to change and extend? |
| Behavioral | How to **behave** or interact | How do objects work together best? |

* 1. ****Creational Patterns****
* **Creational Patterns help create objects in the program in a flexible, optimal way, hiding the complexity of initialization.**
* **Instead of using it “new” to create objects directly in many places, we use patterns to manage how to create objects so that they are easy to control, extend, and reuse.**

Table 7\_List of Creational Patterns-Creational Patterns

|  |  |
| --- | --- |
| **Pattern Name** | **Main objective** |
| 1. **Singleton** | Ensure there is only **one object** in the entire system |
| 2. **Factory Method** | Create an object **through a method**, hiding the logic that creates the object |
| 3. **Abstract Factory** | Create **groups of related objects** without specifying a specific class |
| 4. **Builder** | Creating complex objects **step by step** |
| 5. **Prototype** | **Clone** an existing object instead of creating a new one from scratch |

* + 1. ****Singleton****
* **Singleton Pattern is a design pattern that ensures that:**
  + **Only one instance of a class is created in the entire program.**
  + **And everywhere in the program uses that same object.**
* **Example:**

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 32\_Example Singleton

* Instance is a static variable, so it exists only in memory GetInstance() no matter how many times we call it.
  + Each time we call GetInstance() if instance == null, it will create a new object only once
  + Next time → only return instance old
  + Id3 and Id4 are True, it means they refer to the same instance.
* The object is unique, but the Log(...)messages are different, it helps "unique in state" and "unique in control".
  + Write all Ids to a file
  + Record same time format
  + Record same display type
  + Log level definition: info, warning, error, ...
    1. **Factory Method**
* Factory Method is a design pattern that helps:
  + Create an object, but don't need to know what the specific class of that object is
  + We only know we need a type of object, and let Factory take care of "new what"
  1. ****Structural design patterns****
  2. ****Behavioral design patterns****

1. ****M2 Refine the design to include multiple design patterns.****
2. ****M3 Assess the effectiveness of using SOLID principles, clean coding techniques and programming patterns on the application developed.****
3. ****M4 Discuss the differences between developer- produced and vendor- provided automatic testing tools for applications and software systems.****
4. ****D1 Evaluate the impact of SOLID development principles on object- orientated application development.****
5. ****Apply SRP****

* **In the Use Case Diagram, each actor only performs the functions that are appropriate to their role:**
  + **Students only register for courses and submit assignments.**
  + **Lecturers only grade and download documents.**
  + **Admin creates courses and assigns students.**
* **When switching to Class Diagram, we separate StudentService, CourseService, SubmissionService, each service handles exactly 1 task.**
* **If we don't use SRP : A service can handle too much logic (for example, UserServicehandling registration, submission, grading...), which can be confusing, difficult to maintain, and error-prone when expanding new functionality.**
* **Impact when using : The system is clear, easy to understand, easy to fix, easy to test each part independently.**

1. ****Apply OCP****

* **When building SIMS, the team applied OCP by designing interfaces like IUserRepository, ICourseRepository, IAssignmentRepository. Services only interact with interfaces, not directly dependent on specific classes.**
* **For example:**
  + **CourseService do not manipulate SQL directly but call throughICourseRepository**
  + **When we want to switch from SQL Server to Firebase, just create a new class FirebaseCourseRepository without having to edit the service.**
* **OCP allows the system to be easily extended, adding new features or changing the implementation without breaking the old logic. This is true to the spirit of OOP – the system should be designed to adapt, without having to change the core when requirements change. Without OCP, any change requires editing at multiple layers, creating risk and loss of control.**

1. ****Apply DIP****

* **DIP is most clearly demonstrated in the fact that services do not depend on specific data classes but only on abstractions (interfaces). This helps the SIMS system reduce constraints between layers, increasing flexibility.**
* **For example:**
  + **SubmissionService use IAssignmentRepository, so it can be easily tested by mocking data**
  + **No need to install real database to test logic, because service part has been separated from the storage layer**
* **DIP is consistent with the layered architecture principle in OOP: the business logic should not depend directly on the data layer. If DIP is ignored, the system will be “hard-coded”, making it difficult to write unit tests, and when changing the database, the entire business logic must be modified.**

1. ****Comparison with and without SOLID in SIMS****

Table 8\_Comparison with and without SOLID in SIMS

|  |  |  |
| --- | --- | --- |
| **Criteria** | **When SOLID is not applied** | **When applying SOLID** |
| Code structure | Messy, versatile classes | Each class has a clear role. |
| Expanded functionality | Must fix old code | Adding a new class is enough |
| Unit test | Hard to simulate, hard dependencies | Easy to test due to interface dependency |
| Change request | Break the existing logic | Easy to fix, does not affect other logic |
| Easy to read – maintain | Difficult to understand, easy to bug when fixing | Easy to understand, separate and clear |

1. ****The relationship between SOLID and Object-Oriented Programming (OOP)****

* **OOP only provides “framework” such as encapsulation, inheritance, polymorphism – but it is not enough to ensure a flexible and maintainable system. Many systems written in OOP are still overlapping and confusing if the design principles are not applied correctly.**
* **SOLID is the quality guideline for object-oriented design. In SIMS, applying SOLID helps OOP characteristics to be used properly:**
  + **Inheritance is used in a controlled manner (Student inherit from User, not rampantly)**
  + **Polymorphism is implemented through interfaces.**
  + **Clear encapsulation through each class only performing 1 responsibility**
* **Thanks to that, the SIMS system not only "has classes and interfaces" but also truly reflects a strong and sustainable object-oriented design.**

1. ****Conclusion****

* **The application of SOLID to the SIMS system has had a clear impact on design and development. From Use Case to Class Diagram, each part reflects the thinking of layering, separating tasks, and limiting dependencies - the true spirit of modern object-oriented design. Through this project, I not only understood what SOLID is, but also learned why to use it, how to use it, and how it changes design thinking.**

1. ****D2 Analyze the benefits and drawbacks of different forms of automatic testing of applications and software systems, with examples from the developed application.****
2. ****CONCLUSION****
3. ****REFERENCE****
4. **Marcotte, C.-H. (2022). An atypical ASP.NET core 6 design patterns guide: a solid adventure into architectural principles and design patterns using .NET 6 and C# 10. Birmingham: Packt Publishing.**
5. **Martin, R.C. (2017). Clean Architecture: a craftsman’s guide to software structure and design. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Prentice Hall.**
6. **Gillis, A. S. and Lewis, S. (2024) What is Object-Oriented Programming (OOP)?: Definition from TechTarget, Search App Architecture. TechTarget. Available at:** <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP> **(Accessed: 16 May 2025).**
7. **‌** **Blog, T. (2021) Design pattern LÀ GÌ? Tại Sao Nên SỬ dụng design pattern?, TopDev. Available at:** <https://topdev.vn/blog/design-pattern-la-gi/> **(Accessed: 25 June 2025).**