ASSIGNMENT 1 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 20: Advanced Programming		
Submission date	29/06/2021	Date Received 1st submission	
Re-submission Date		Date Received 2nd submission	
Student Name	PHAM CAO NGUYEN	Student ID	GCC18074
Class	GCC0801	Assessor name	TRUNG-VIET NGUYEN

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	CAONGUYEN

Grading grid

P1	P2	M1	M2	D1	D2

✿ **Summative Feedback:**   ✿ **Resubmission Feedback:**

Grade:	Assessor Signature:	Date:
Lecturer Signature:		

ASSIGNMENT 1 BRIEF

Qualification	**BTEC Level 5 HND Diploma in Computing**
Unit number	Unit 20: Advanced Programming
Assignment title	Examine and design solutions with OOP and Design Patterns
Academic Year	2018-2019
Unit Tutor	Doan Trung Tung
Issue date	25 April 2019	**Submission date**	7 May 2019

Submission Format:

Format: The submission is in the form of a **group written report**. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system.

Submission Students are compulsory to submit the assignment in due date and in a way requested by the Tutors. The form of submission will be a soft copy in PDF posted on corresponding course of http://cms.greenwich.edu.vn/

Note: The Assignment *must* be your own work, and not copied by or from another student or from books etc. If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. Make sure that you know how to reference properly, and that understand the guidelines on plagiarism. *If you do not, you definitely get fail*

Assignment Brief and Guidance:

Scenario: You have recently joined a software development company to help improve their documentation of their in-houses software libraries which were developed with very poor documentation. As a result, it has been very difficult for the company to utilise their code in multiple projects due to poor documentation. Your role is to alleviate this situation by showing the efficient of UML diagrams in OOAD and Design Patterns in usages.

Tasks

You and your team need to explain characteristics of Object-oriented programming paradigm by applying Object-oriented analysis and design on a given (assumed) scenario. The scenario can be small but should be able to presents various characteristics of OOP (such as: encapsulation, inheritance, polymorphism, override, overload, etc.).

The second task is to introduce some design patterns (including 3 types: creational, structural and behavioral) to audience by giving real case scenarios, corresponding patterns illustrated by UML class diagrams.

To summarize, you should analyze the relationship between the object-orientated paradigm and design patterns.

The presentation should be about approximately 20-30 minutes and it should be summarized of the team report.

Learning Outcomes and Assessment Criteria			
Pass	**Merit**	**Distinction**	
LO1 Examine the key components related to the object-orientated programming paradigm, analysing design pattern types			
P1 Examine the characteristics of the object-orientated paradigm as well as the various class relationships.	**M1** Determine a design pattern from each of the creational, structural and behavioural pattern types.	**D1** Analyse the relationship between the object-orientated paradigm and design patterns.	
LO2 Design a series of UML class diagrams			
P2 Design and build class diagrams using a UML tool.	**M2** Define class diagrams for specific design patterns using a UML tool.	**D2** Define/refine class diagrams derived from a given code scenario using a UML tool.	

Contents

Part 1. Examine the key components related to the object-orientated programming paradigm, analysing design pattern types

I. Examine the characteristics of the object-orientated paradigm as well as the various class relationships.

1. What is object-oriented programming?

- Object-Oriented Programming refers to the programming paradigm based on the concept of objects. It can also contain data in the form of different fields and these fields are known as properties or attributes. It also includes code in the form of procedures, which are known as methods. (Pecinovsky, 2013)
- The purpose of Object-Oriented Programming is to implement real-world entities such as polymorphism, inheritance, hiding, etc. It binds functions and data that operate over them in order to ensure that no code can access the particular data instead of function. OOPs refers to the languages that utilize the objects in programming. (Pecinovsky, 2013)

1.1. Advantages

- We have the advantages of object-oriented programming is.
  - OOP provides ease of operation due to its modularity and encapsulation.
  - OOP mimics the real world, which makes things easy to comprehend.
  - Since objects are whole within themselves, they can be reused in other programs.

1.2. Disadvantages

- The disadvantages when we use object-oriented programming.
  - Object-oriented programs appear to be sluggish and have a large memory consumption.
  - Overexploitation.
  - Programs constructed using this model can take more time to build.

1.3. Why is object-oriented programming needed?

- OOP - object-oriented programming - was created to address all of the drawbacks of earlier programming approaches, which had far too many flaws. Specifically:
- Object-oriented programming is extremely close to real-world, practical use. Once the objects have been visualized with any property methods. Then programmers can develop the program naturally, close to natural language
  - OOP also provides very quick error correction due to its proximity to natural language.
  - Easily manage code when there are changes to the program.
  - Very high security, easy to scale projects.
  - OOP allows using source code to save resources
  - Object-oriented programming is also very intuitive when moving from a real analysis model to a software implementation model.
  - Ability to maintain, change programs more efficiently and quickly.
  - Easily divide the system into small pieces to hand over to development teams.
  - Ability to reuse code when building other programs.

- Very well integrated with existing computers, suitable for modern operating systems. Ability to create an intuitive user interface.
- OOP boosts productivity while also simplifying program maintenance and expansion. Reduce the amount of code that the programmer has to do. As a result, OOP is extensively utilized; programmers may develop programs in which external components, such as actual objects, interact with the software.

2. Procedural-oriented Programming

- Procedural programming languages are also imperative languages because they make explicit references to the state of the execution environment. This could be anything from variables (which may correspond to processor registers) to something like the position of the "turtle" in the Logo programming language. (Pecinovsky, 2013)
- Often, the terms "procedural programming" and "imperative programming" are used synonymously. However, procedural programming relies heavily on blocks and scope, whereas imperative programming as a whole may or may not have such features. As such, procedural languages generally use reserved words that act on blocks, such as if, while, and for, to implement control flow, whereas non-structured imperative languages use goto statements and branch tables for the same purpose. (Pecinovsky, 2013)

2.1. Advantages

- Advantages of using Procedural-oriented Programming are.
  - Procedural-oriented programming is great for strategic planning.
  - Written flexibility together with ease of compiler and interpreter implementation.
  - A broad range of books and online training content based on validated algorithms to make learning simpler along the way.
  - The source code is adaptive, since a certain Processor may also be exploited.
  - The algorithm can be reused in certain areas of the program, without the need to copy it.
  - The memory level also slashes through the technique of Procedural Programming.
  - Effective analysis of the program flow.

2.2. Disadvantages

- Some disadvantages of using Procedural-oriented Programming.
  - The program code is harder to write when Procedural Programming is employed.
  - The Code of Procedure is also not reusable, which might necessitate the reconstruction of the code if it is used in another application.
  - Difficult to relate to real-world objects.
  - The importance is given to the operation rather than the data, which might pose issues in some data-sensitive cases.
  - The data is revealed to the whole software and does not render it secure.

3. Difference between Procedural programming and Object-Oriented Programming

- Data security is jeopardized in POP because data travels freely inside the software, and code reusability is not achieved, making development time-consuming and complex. Larger programs result in more problems and take longer to debug. (Pecinovsky, 2013)

- Both of these drawbacks need the adoption of a new methodology, namely object-oriented programming. Data protection, which tightly links data with pre-built functions, is a major problem in object-oriented programming.
- OOP also solves the issue of application reusability, because once a class is established, many instances of it (objects) may be used to reuse the class's members and member methods.

4. Features of object-oriented programming

4.1. Object

- An object, in object-oriented programming (OOP), is an abstract data type created by a developer. It can include multiple properties and methods and may even contain other objects. In most programming languages, objects are defined as classes. (Pecinovsky, 2013)
- An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
    char name[20];
    int id;
public:
    void getdetails(){}
};

int main()
{
    person p1; // p1 is a object
}
```

4.2. Class

- In object-oriented programming, a class is a template definition of the method s and variable s in a particular kind of object. Thus, an object is a specific instance of a class; it contains real values instead of variables.
- Class and object are two essential ideas of Object-Oriented Programming that center around real-life things. A class is a pre-defined blueprint or template that may be used to create objects. A class is a single entity that combines fields and methods (member functions that specify activities). Objects in C# support polymorphism, inheritance, and the concepts of derived classes and base classes.

For example:

```
class Car
{
    string color = "red";

    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.color);
    }
}
```

```
red
```

4.3. Relationship:

No.	Object	Class
1.	The object is an instance of a class	Classes are a template or design for creating objects
2.	The object is a real-world entity like a pencil or a bicycle	A class is a group of similar objects
3.	The object is a physical entity	Class is a logical entity
4.	The object is created mostly from the new keyword.	Class is declared using the class keyword
5.	Objects can be created many times	Class is declared only once
6.	The object is allocated memory when it is created	The class is not allocated memory when it is created

Table 1: Relationship between Object and Classes

4.4. Encapsulation

- "The process of arranging one or more things into a physical or logical container" is how encapsulation is defined. In object-oriented programming theory, encapsulation restricts access to implementation knowledge. (Geiger, 2004)
- Encapsulation aids a programmer in imposing his or her preferred abstraction level.
- Another important feature of OOP is encapsulation. Encapsulation is the process of hiding data such that it cannot be accessed directly. If you wish to see the data, you'll need to contact the data entity in charge.
- Consider the last time you mailed a letter. You ask for the letter to be delivered by the post office. You have no idea how the post office achieves this. It has no effect on how you start sending the letter if it affects the path it takes to mail it. You don't need to be familiar with the post office's internal processes for delivering the letter.

Advantages of Encapsulation:

- The most important advantage of encapsulation is data security. Some of the advantages of encapsulation are as follows:
- Clients are prevented from accessing an entity if it is encapsulated.

- Encapsulation helps you to access a dimension without exposing the intricate information under it.
- It lowers the number of human errors.
- The application's updating is made easier.
- It makes the application more understandable.

For example:

```java
public class Coat {
    private double price;
    private String customer;

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getCustomer() {
        return customer;
    }

    public void setCustomer(String customer) {
        this.customer = customer;
    }
}
```

Or:

```csharp
using System;

public class DemoEncap {

    // private variables declared
    // these can only be accessed by
    // public methods of class
    private String studentName;
    private int studentAge;

    // using accessors to get and
    // set the value of studentName
    public String Name
    {
        get
        {
            return studentName;
        }

        set
        {
            studentName = value;
        }
    }

    // using accessors to get and
    // set the value of studentAge
    public int Age
    {
        get
        {
            return studentAge;
        }
    }
}
```

```
        }
        set
        {
            studentAge = value;
        }
    }
}
// Driver Class
class GFG {
    // Main Method
    static public void Main()
    {
        // creating object
        DemoEncap obj = new DemoEncap();

        // calls set accessor of the property Name,
        // and pass "Ankita" as value of the
        // standard field 'value'
        obj.Name = "Ankita";

        // calls set accessor of the property Age,
        // and pass "21" as value of the
        // standard field 'value'
        obj.Age = 21;

        // Displaying values of the variables
        Console.WriteLine("Name: " + obj.Name);
        Console.WriteLine("Age: " + obj.Age);
```

```
Name: Ankita
Age: 21
```

4.5. Abstraction

- Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. (Pecinovsky, 2013)

- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

- Abstraction is an object-oriented programming language (OOP) principle used in C# to cover design specifics and view only basic features of the object. A class is the best representation of abstraction in object-oriented programming. In c#, we can create a class with required methods, properties and we can expose only necessary methods and properties using access modifiers based on our requirements.

- Abstraction using Classes: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

- Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header

file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Advantages of Abstraction:

- It simplifies the process of seeing objects.
- Reduces the reuse of coding and improves reusability.
- Only essential data are given to the user, which helps to improve the security of an application or service.
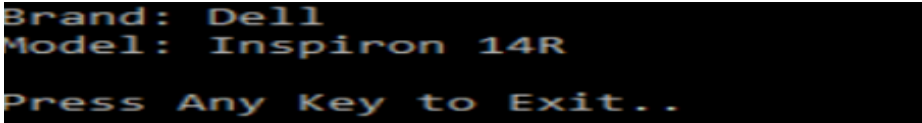
For example:

```csharp
using System;
using System.Text;

namespace Tutlane
{
    public class Laptop
    {
        private string brand;
        private string model;
        public string Brand
        {
            get { return brand; }
            set { brand = value; }
        }
        public string Model
        {
            get { return model; }
            set { model = value; }
        }
        public void LaptopDetails()
        {
            Console.WriteLine("Brand: " + Brand);
            Console.WriteLine("Model: " + Model);
        }
        public void LaptopKeyboard()
        {
            Console.WriteLine("Type using Keyword");
        }
        private void MotherBoardInfo()
        {
            Console.WriteLine("MotheBoard Information");
        }
```

```csharp
        {
            Console.WriteLine("Brand: " + Brand);
            Console.WriteLine("Model: " + Model);
        }
        public void LaptopKeyboard()
        {
            Console.WriteLine("Type using Keyword");
        }
        private void MotherBoardInfo()
        {
            Console.WriteLine("MotheBoard Information");
        }
```

```
        private void InternalProcessor()
        {
            Console.WriteLine("Processor Information");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Laptop l = new Laptop();
            l.Brand = "Dell";
            l.Model = "Inspiron 14R";
            l.LaptopDetails();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

```
Brand: Dell
Model: Inspiron 14R

Press Any Key to Exit..
```

4.6. Interface

- An abstract base class with just abstract members is generally used for the interface. All members of an interface must be enforced by each class or structure that implements it. The interface can optionally define default implementations for any or all of its elements. (Hookway, 2014)
- The code cannot be directly invoked. Any class or framework that implements the interface enforces its elements. A class or structure can implement several interfaces.
- A class can inherit a base class and can also implement one or more interfaces.

For example:

- To access the interface methods, another class must "implement" (as inherited) the interface. To enforce the code, use the: sign (as with inheritance). The "implement" class defines the body of the interface method. It's worth noting that you don't have to use the override keyword when introducing an interface:

```
Program.cs
using System;

namespace MyApplication
{
  // Interface
  interface IAnimal
  {
    void animalSound(); // interface method (does not have a body)
  }

  // Pig "implements" the IAnimal interface
  class Pig : IAnimal
  {
    public void animalSound()
    {
      // The body of animalSound() is provided here
      Console.WriteLine("The pig says: wee wee");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Pig myPig = new Pig();  // Create a Pig object
      myPig.animalSound();
    }
  }
}
```

```
The pig says: wee wee
```

4.7. Polymorphism

- Polymorphism is the capacity to act in a variety of ways depending on the situation. It is most commonly encountered in applications where many methods with the same name but distinct parameters and behavior are specified. (Wang, 1995)
- You may inherit the properties and methods of another class via inheritance. Polymorphism uses these methods to carry out a variety of tasks. It allows you to do a single activity in a variety of ways.

For example:

```
using System;

namespace MyApplication
{
  class Animal  // Base class (parent)
  {
    public void animalSound()
    {
      Console.WriteLine("The animal makes a sound");
    }
  }
```

```
class Pig : Animal   // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal   // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

4.8. Inheritance

- The act of generating a new class based on the attributes and methods of an existing class is known as inheritance. The current class is referred to as the base class, while the newly generated class is referred to as the derived class. This is a crucial notion in object-oriented programming because it allows inherited properties and functions to be reused.
- You may use inheritance to create new classes that reuse, expand, and alter the behaviors defined in previous classes. The base class is the one whose members are inherited, and the derived class is the one that inherits those members. There can only be one direct base class for a derived class. Nonetheless, the succession is just temporary.

For example:

```csharp
using System;

namespace MyApplication
{
  class Vehicle  // Base class
  {
    public string brand = "Ford";  // Vehicle field
    public void honk()             // Vehicle method
    {
      Console.WriteLine("Tuut, tuut!");
    }
  }
}
```

```csharp
using System;

namespace MyApplication
{
  class Car : Vehicle  // Derived class
  {
    public string modelName = "Mustang";  // Car field
  }
}
```

```csharp
using System;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      // Create a myCar object
      Car myCar = new Car();

      // Call the honk() method (from the Vehicle class) on the myCar object
      myCar.honk();

      // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
      Console.WriteLine(myCar.brand + " " + myCar.modelName);
    }
  }
}
```

```
Tuut, tuut!
Ford Mustang
```

4.9. Abstract class

- The abstract adjustment indicates that the item being modified is absent or incompletely implemented. Classes, processes, properties, indexers, and events can all benefit from abstract modifiers. Use an abstract changer in a class declaration to indicate that a class is intended to be a base class for other classes rather than a standalone object. Members marked as abstract must perform non-abstract classes derived from abstract classes.

Abstract Class Features

- An abstract class may inherit one or more interfaces from the same class.
- An abstract class can use non-Abstract methods to implement code.
- Constants and fields may be in an Abstract class.
- Under abstract class, a property can be enforced.
- Constructors or destructors that have an abstract class.
- Unable to inherit an abstract entity from the structures.
- Can does not accept multiple inheritances by an abstract class.

For example:

```
//Abstract class can have constant and fields
public abstract class ConstantFields
{
    public int no;
    private const int id = 10;
}
```

Or:

```
1 reference
abstract class Tutorial
{
    0 references
    public virtual void Set()
    {

    }
}
2 references
class Guru99Tutorial : Tutorial
{
    protected int TutorialID;
    protected string TutorialName;

    1 reference
    public void SetTutorial(int pID, string pName)
    {
        TutorialID = pID;
        TutorialName = pName;
    }

    1 reference
    public String GetTutorial()
    {
        return TutorialName;
    }

    0 references
    static void Main(string[] args)
    {
        Guru99Tutorial pTutor = new Guru99Tutorial();

        pTutor.SetTutorial(1, ".Net");

        Console.WriteLine(pTutor.GetTutorial());

        Console.ReadKey();
    }
}
```

Result

C:\Users

.Net

Part 2: Design a series of UML class diagrams

II. Design and build class diagrams using a UML tool.

❖ **Project Specification:**

FPT is an international academy and now they want to create an application to store the list of students, faculty, employees, customers, and show the main menu. I perform the segment of managing to in student management system. My team and I will program and develop the student management system for FPT International Academy, and I am the team leader. An application should be designed to show all the students of the school. And display details about student information such as phone number, address, email, DoB...with functions such as search, adding, editing, and deleting customer and employee information as well as students and lecturers. May view all as ID, Name, Phone, Mail, Address.

1. Use-case diagram.

- **Use case diagram includes:** Manage Students, Manage Lecturers.
- **Manage Students:** Administrators may conduct student management roles to handle students in an overall manner. Student management functions include basic functions such as: adding, deleting, upgrading students, administrators can scan for and display a list of all the students entered while active, and a spin feature. Back to main menu.
- **Manage Lecturers:** Administrators may conduct lecturers' management roles to handle Lecturers in an overall manner. Lecturer management functions include basic functions such as: adding, deleting, upgrading Lecturers, administrators can scan for and display a list of all the Lecturers entered while active, and a spin feature. Back to main menu.
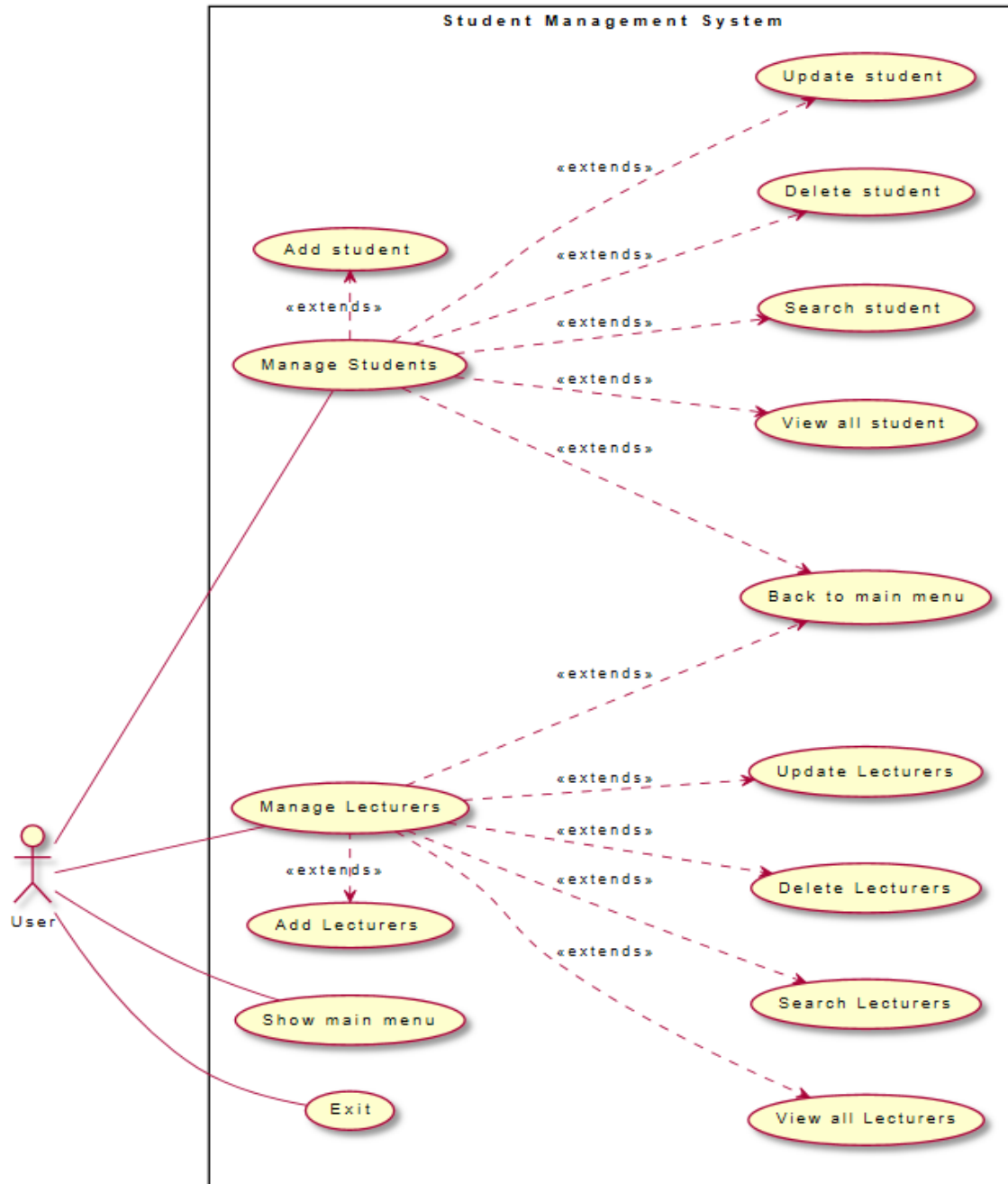
Figure 1: Use case diagram

- **Manage Student Menu:**
  - Add student
  - Update student
  - Delete student
  - View all student

- Back to main menu
- Exit

- **Manage Lecture Menu:**
  - Add lecture
  - Update lecture
  - Delete lecture
  - View all lecture
  - Back to main menu
  - Exit

- **Use-case diagram's notation.**
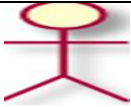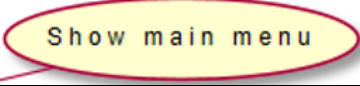
No.	Symbols	Description
1. Actor		The actor is used to referring to a user or an external object interacting with the system we are looking at.
2. Use Case	Show main menu	Use Case is the function that Actors will use.
3. Relationship	«extends»	The term "extends" was used to denote the connection between the two Use Cases. When a Use Case is built to add functionality to an existing Use Case and is utilized under a certain condition, the Extend relation is employed.

Table 2: Use-case diagram's notation.

- **Use-case code:**

```
@startuml
'' plantumlfile1
left to right direction
skinparam packageStyle rectangle
actor User
rectangle "Student Management System" {
User --- (Manage Students)
(Manage Students) -right-.> (Add student) : <<extends>>
(Manage Students) --.> (View all student) : <<extends>>
(Manage Students) --.> (Search student) : <<extends>>
(Manage Students) --.> (Delete student) : <<extends>>
(Manage Students) --.> (Update student) : <<extends>>
(Manage Students) --.> (Back to main menu) : <<extends>>
User --- (Show main menu)
User --- (Manage Lecturers)
(Manage Lecturers) -left-.> (Add Lecturers) : <<extends>>
(Manage Lecturers) --.> (View all Lecturers) : <<extends>>
(Manage Lecturers) --.> (Search Lecturers) : <<extends>>
(Manage Lecturers) --.> (Delete Lecturers) : <<extends>>
(Manage Lecturers) --.> (Update Lecturers) : <<extends>>
(Manage Lecturers) --.> (Back to main menu) : <<extends>>
```

```
User --- (Exit)
@enduml
```

2. Class diagram.

- The class diagram includes the following properties: Person, student, program, lecturer. in the Person section include: ID, Name, DoB, Email, Address. Properties in Person will be set in lecturer and student.
- IPerson interface has InputForm (), Validate () methods used to check ID, DisplayInfor () The abstract Person class extends the IPerson interface to implement methods. It is also a class inherited by the Student and Instructor classes to add information such as ID, Name, Email, Address, DoB. Teacher class has a Division property and a student class has a Batch property, they work similarly
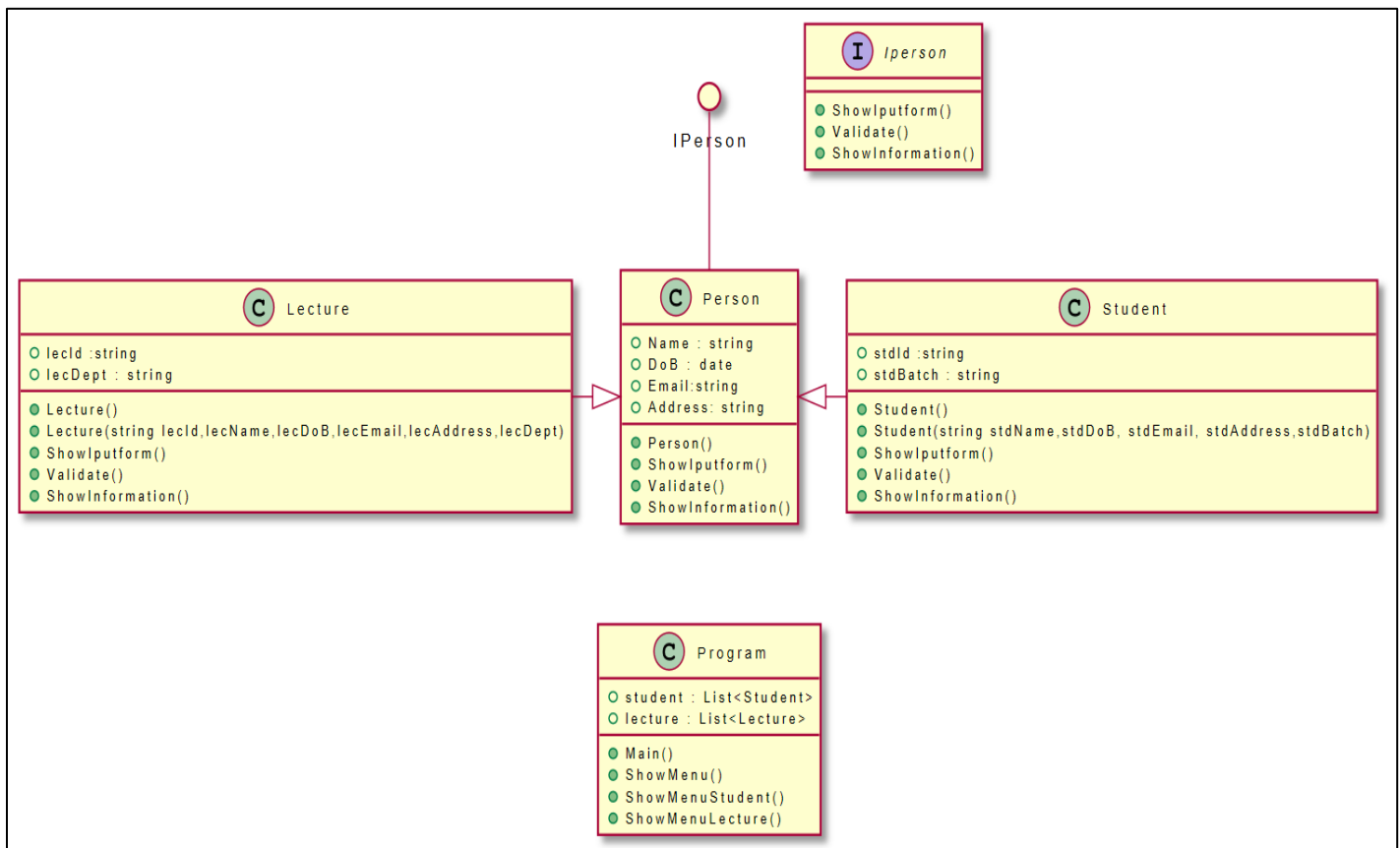
Figure 3: Class diagram

- **Class diagram's notation**

No.	Symbols	Description
1. Extension		Relationship between classes

2. Public		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
3. Private		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
4. Private		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
5. Object		Center circle
6. Interface	I interface	Show interface of the diagram.
7. Class	C class	Show class of the diagram.
8. Classes	Class name / Attributes / Methods	The fundamental component of the Class Diagram drawing is the class. In the system, a class refers to a set of objects that share the same attributes and behaviors. The class "Customer" is used to describe a customer, for example. Class described includes Class name, properties, and methods.

Table 2: Class diagram's notation

- Relations between classes are defined using the following symbols:
- It is possible to replace -- by .. to have a dotted line.

Type	Symbol	Drawing
Extension	`<|--`	◁—
Composition	`*--`	◆—
Aggregation	`o--`	◇—

Figure 4: Relations between classes

- **The code of the Class diagram:**

```
@startuml
circle IPerson
class Person {
    +Name : string
```

```
        +DoB : date
        +Email:string
        +Address: string
        + Person()
        + ShowIputform()
        + Validate()
        + ShowInformation()
}
IPerson -- Person
Student -left-^ Person
Lecture -right-^ Person
Person -[hidden]-> Program
class Student {
        + stdId :string
        + stdBatch : string
        + Student()
        + Student(string stdName,stdDoB, stdEmail, stdAddress,stdBatch)
        + ShowIputform()
        + Validate()
        + ShowInformation()
}
class Lecture {
        + lecId :string
        + lecDept  : string
        + Lecture()
        + Lecture(string lecId,lecName,lecDoB,lecEmail,lecAddress,lecDept)
        + ShowIputform()
        + Validate()
        + ShowInformation()
}
class Program {
        +student : List<Student>
        +lecture : List<Lecture>
        + Main()
        + ShowMenu()
        + ShowMenuStudent()
        + ShowMenuLecture()
}
interface Iperson {
        + ShowIputform()
        + Validate()
        + ShowInformation()
}
@enduml
```

3. What are Design Patterns?

3.1. Definition:

-       Design Patterns are models of code that solve classic problems. They are solutions to software design problems that you can find in a real-world application. A Design Pattern is not a code that is ready to be used in your application, but it is a model that you can use to solve a problem. (Johnson, 1995)

-       Design patterns are language-neutral, so they can be applied to any language that supports object orientation.

3.2. We have the pros and cons of using design patterns.

-   Advantages: The option to use tried and tested solutions can save you time and money. Development teams do not have to constantly reinvent the wheel to solve subtasks in a program procedure that has already been solved multiple times before. The individual patterns are usually named using common design vocabulary. This simplifies discussions between developers as well as communication with the user of the future solution. The software's documentation is also simplified if you use building blocks that have already been documented. These advantages also hold true for the maintenance and further development of a program. (Johnson, 1995)

-   Disadvantages: Using design patterns requires extensive knowledge. Having design patterns available can also lead to people believing that apparently all problems can be solved using existing design patterns. In short, this can limit creativity and the desire to find new (better) solutions. (Johnson, 1995)

3.3. Here are some examples of design patterns.

-   The following code snippet is an example of low cohesion:

```
class AccountService{

  public Account createAccount(){
  // business logic
  }

  public Opportunity createOpportunity(){
  // business logic
  }

  public Contact createContact(){
  // business logic
  }
}
```

Figure 5: Example of low cohesion

-   The following code snippet is an example of low cohesion:

```
class AccountService{

  public Account createAccount(){
  // business logic
  }
}

class OpportunityService

  public Opportunity createOpportunity(){
  // business logic
  }
}

class ContactService

  public Contact createContact(){
  // business logic
  }
```

Figure 6: Example of low cohesion

-   The following diagram shows how we converted low cohesion to high cohesion.
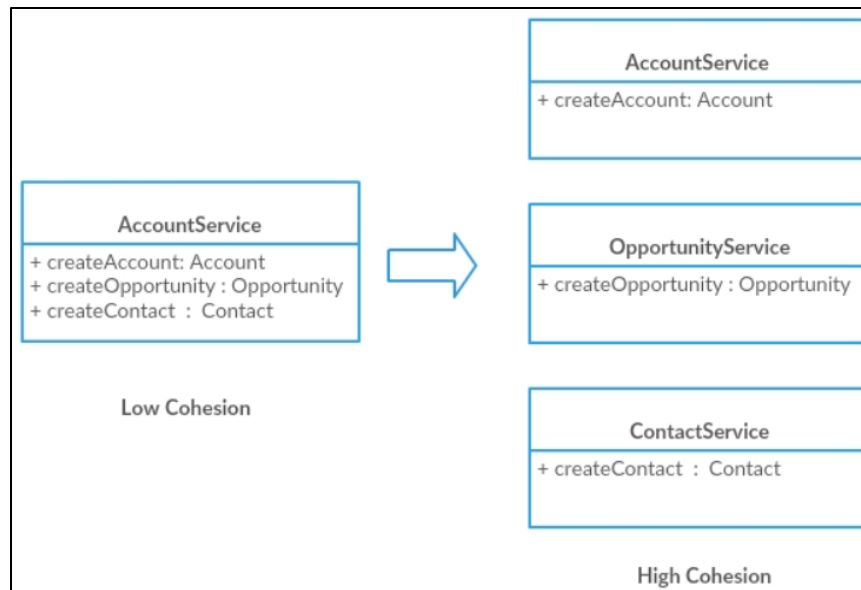
Figure 6: Converted low cohesion to high cohesion.

- Another advantage of using design patterns is that if testers know that a specific design pattern is used in an implementation, they can quickly relate to it. According to their past experience with design patterns, they can easily identify possible failure scenarios.
- Next in the list of advantages are communication and support. Design patterns are well-known in the developer community and forums. Also, they can be easily discussed with your technical lead, project manager, test lead, or architects. When someone new joins your development team, usage of design patterns can help describe the code base to the new team member and aid in a developer's ramp up and acclimation.

4. Introduction to Creational Design Patterns.
- **Introduce**
  - In software engineering, Design Patterns describe building solutions to the most common problems in software design. It represents best practices developed over a long period of time through trial and error by experienced developers. (Johnson, 1995)

  - In this article, we will learn about creational design patterns and their types. We'll also look at a few examples and discuss the conditions under which design patterns are suitable for design**.**

- **Creational Design Patterns**
  - Creational Design Patterns are concerned with how objects are initialized. They reduce the complexity and instability of creating objects in a controlled manner.
  - The new operator is generally considered harmful because it scatters objects across the application. Over time, changing implementations can become difficult as classes become tightly coupled.

- Creational Design Patterns solve this problem by completely decoupling the client from the actual initialization process
- **In this article, we discuss four types of Creational Design Patterns:**
  - Singleton – Ensures that only one instance of the object exists throughout the application.
  - Factory Method – Creates objects of several related classes without specifying the exact object to be created.
  - Abstract Factory – Creates the same related dependent objects.
  - Builder – Build complex objects using a step-by-step approach.

5. Activity Diagram:
- **Function table and legend of the flowchart:**

No.	Symbols
1. Process: A box indicates some particular operation.	
2.The terminator symbol represents the starting or ending point of the system.	
3. Decision: A diamond represents a decision or branching point. Lines coming out from the diamond indicates different possible situations, leading to different sub-processes.	
4. Arrows: Used to connect symbols and indicate the flow of logic	
5. Input/Output: Used to input and output operations, such as reading and printing. The data to be read or printed are described inside	
6. Connector: Used to join different flowlines.	

Table 3: Function table and legend of the flowchart:

- **Add Information Student/ Lecture:**
  - If the user needs to assign students to a specific class, use this case. To add students to a particular list, the system allows users to select classes and personal details of students.
  - Use this case starting when a user wants to add a lecture of a certain class. The system requires the user to select the class and personal information of the lecture for the user to add the lecture to a specific list.
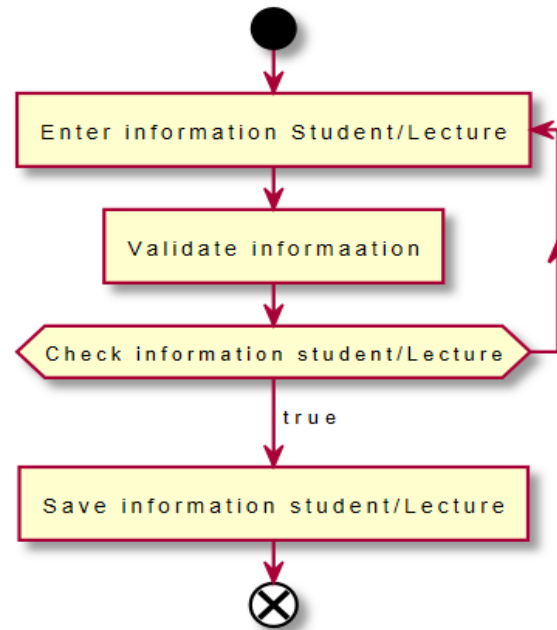
Figure 7: Add Information Student/Lecture

Figure 7: Add Information Student/Lecture

- **Code:**

```
@startuml
start
repeat:Enter information Student/Lecture]
  :Validate informaation]
repeat while (Check information student/Lecture)
->true;
  :Save information student/Lecture]
end
@enduml
```

o **Update information Student/Lecture:**
   - When a student wishes to make changes to their information, they can go to the update student page and choose to save.
   - When a lecture wishes to make changes to their information, they can go to the update lecture page and choose to save.
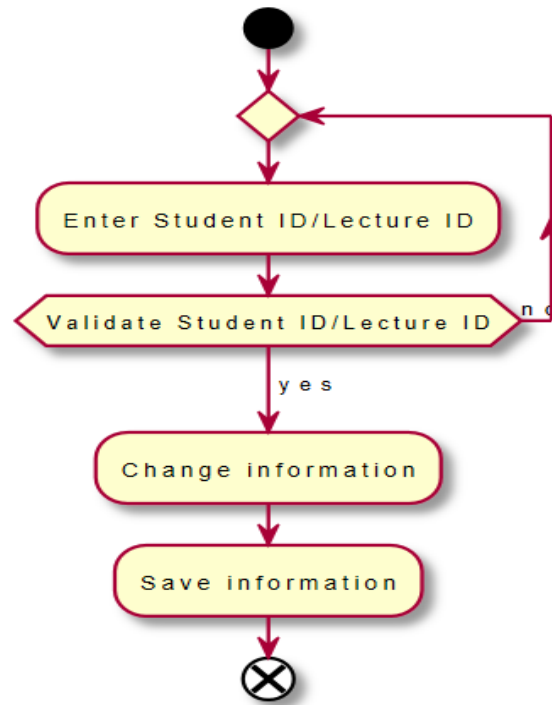
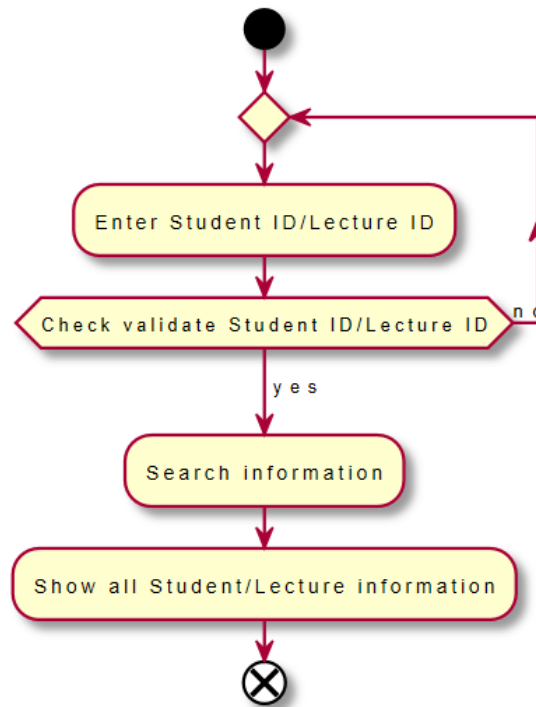Figure 8: Update Information Student/Lecture

- **Code:**

```
@startuml
start
repeat
        :Enter Student ID/Lecture ID;
repeat while (Validate Student ID/Lecture ID) is (no)
-> yes;
        :Change information;
        :Save information;
end
@enduml
```

o **Delete Student/Lecture:**
- To delete a student, need to enter the student ID to be deleted, then confirm the student ID, if it is correct, the user selects delete.
- To delete a lecture, need to enter the lecture ID to be deleted, then confirm the lecture ID, if it is correct, the user selects delete.
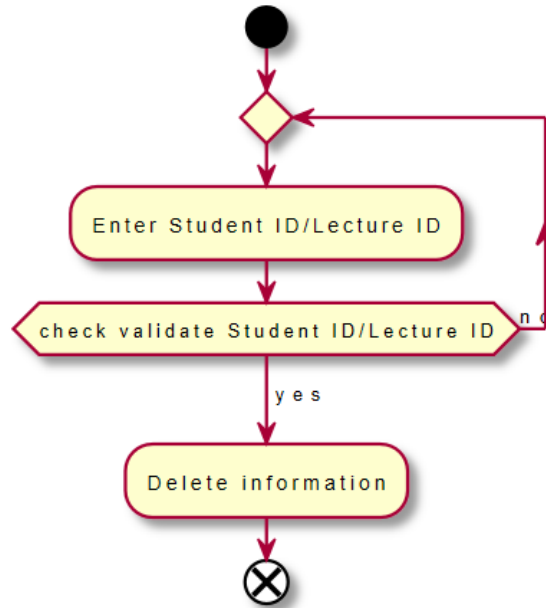
Figure 9: Delete Student/Lecture.

- **Code:**

```
@startuml
start
repeat
      :Enter Student ID/Lecture ID;
repeat while (check validate Student ID/Lecture ID) is (no)
-> yes;
      :Delete information;
end
@enduml
```

- o **Search information Student/Lecture:**
- To search for students, the user enters the Student ID, the system will display the student name that needs to be searched.
- As the user types in the lecture number, the device displays the name of the lecture that needs to be found.

Figure 10: Search information Student/Lecture.

- **Code:**

```
@startuml
start
repeat
      :Enter Student ID/Lecture ID;
repeat while (Check validate Student ID/Lecture ID) is (no)
-> yes;
      :Search information;
      :Show all Student/Lecture information;
end
@enduml
```

References

Pecinovsky, R., 2013. *OOP-Learn Object Oriented Thinking & Programming*. Tomáš Bruckner.

Hookway, B., 2014. *Interface*. MIT Press.

Wang, G. and Ambler, A., 1995, September. Invocation polymorphism. In *Proceedings of Symposium on Visual Languages* (pp. 83-90). IEEE.

Geiger, S.K., 2004. *Legacy Computing Markup Language (LCML) and LEGEND--LEGacy Encapsulation for Network Distribution* (Doctoral dissertation, Massachusetts Institute of Technology).

Johnson, R. and Vlissides, J., 1995. Design patterns. *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading.*

Advanced Programming

Assessor name: **TRUNG-VIET NGUYEN**

Student Name: **PHAM CAO NGUYEN**

Class: **GCC0801**

Student ID: **GCC18074**

Part 1. Examine the key components related to the object-orientated programming paradigm, analyzing design pattern types.

I. Examine the characteristics of the object-orientated paradigm as well as the various class relationships.
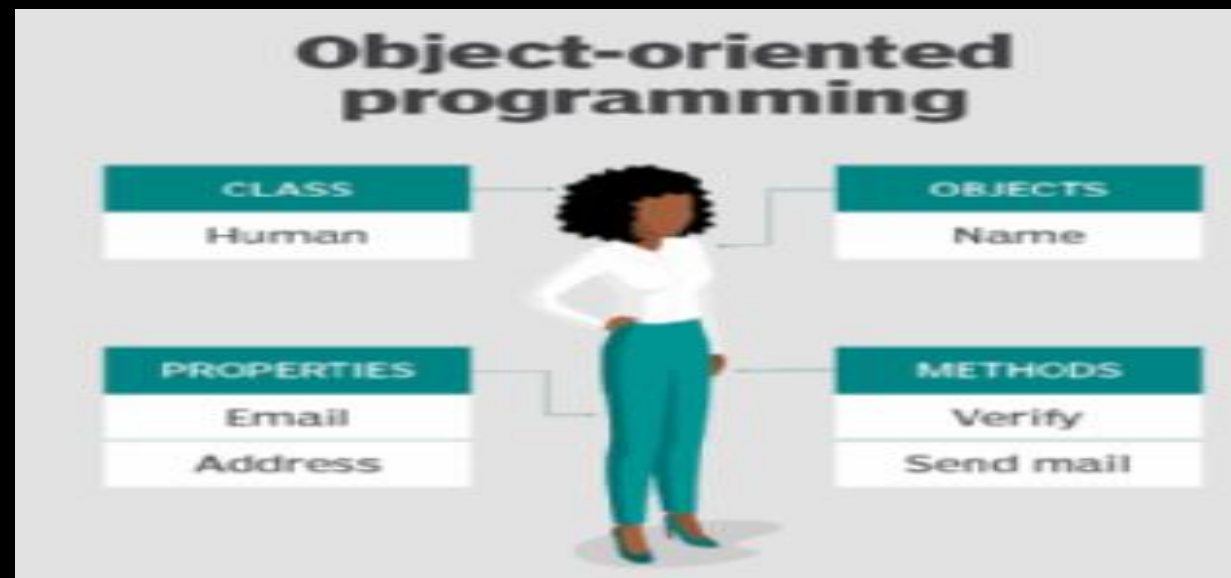
1. What is object-oriented programming?

- **Object-Oriented Programming refers to the programming paradigm based on the concept of objects. It can also contain data in the form of different fields and these fields are known as properties or attributes. It also includes code in the form of procedures, which are known as methods.**

<What is OOP?>

2. Advantages and Disadvantages

❑**Advantages**

▪ **OOP provides ease of operation due to its modularity and encapsulation.**

▪ **OOP mimics the real world, which makes things easy to comprehend.**

▪ **Since objects are whole within themselves, they can be reused in other programs.**

❑ **Disadvantages**

▪ **Object-oriented programs appear to be sluggish and have a large memory consumption.**

▪ **Overexploitation.**

▪ **Programs constructed using this model can take more time to build.**

3. Procedural-oriented Programming

Procedural programming languages are also imperative languages because they make explicit references to the state of the execution environment. This could be anything from variables (which may correspond to processor registers) to something like the position of the "turtle" in the Logo programming language.

4. Advantages and Disadvantages

❑**Advantages**

▪ **Procedural-oriented programming is great for strategic planning.**

▪ **Written flexibility together with ease of compiler and interpreter implementation.**

▪ **A broad range of books and online training content based on validated algorithms to make learning simpler along the way.**

▪ **The source code is adaptive, since a certain Processor may also be exploited.**

▪ **The algorithm can be reused in certain areas of the program, without the need to copy it.**

▪ **The memory level also slashes through the technique of Procedural Programming.**

▪ **Effective analysis of the program flow.**

❑ **Disadvantages**

▪ **The program code is harder to write when Procedural Programming is employed.**

▪ **The Code of Procedure is also not reusable, which might necessitate the reconstruction of the code if it is used in another application.**

▪ **Difficult to relate to real-world objects.**

▪ **The importance is given to the operation rather than the data, which might pose issues in some data-sensitive cases.**

▪ **The data is revealed to the whole software and does not render it secure.**

5. Relationship of Objects and Classes

❑ **Objects: An object is an instance of a class that collects data and procedures for manipulating data.**

❑ **Classes: A class defines the properties of objects linked to it.**

Object	Class
The object is an instance of a class	Classes are a template or design for creating objects
The object is a real-world entity like a pencil or a bicycle	A class is a group of similar objects
The object is a physical entity	Class is a logical entity
The object is created mostly from the new keyword.	Class is declared using the class keyword
Objects can be created many times	Class is declared only once
The object is allocated memory when it is created	The class is not allocated memory when it is created

6. Encapsulation

❑ **"The process of arranging one or more things into a physical or logical container" is how encapsulation is defined. In object-oriented programming theory, encapsulation restricts access to implementation knowledge.**

```
public class Coat {
    private double price;
    private String customer;

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getCustomer() {
        return customer;
    }

    public void setCustomer(String customer) {
        this.customer = customer;
    }
}
```

7. Abstraction

❑ **Abstract classes and interfaces are used to hide the internal details and show the functionality.**

```
Brand: Dell
Model: Inspiron 14R

Press Any Key to Exit..
```

```csharp
using System;
using System.Text;

namespace Tutlane
{
    public class Laptop
    {
        private string brand;
        private string model;
        public string Brand
        {
            get { return brand; }
            set { brand = value; }
        }
        public string Model
        {
            get { return model; }
            set { model = value; }
        }
        public void LaptopDetails()
        {
            Console.WriteLine("Brand: " + Brand);
            Console.WriteLine("Model: " + Model);
        }
        public void LaptopKeyboard()
        {
            Console.WriteLine("Type using Keyword");
        }
        private void MotherBoardInfo()
        {
            Console.WriteLine("MotheBoard Information");
        }
        private void InternalProcessor()
        {
            Console.WriteLine("Processor Information");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Laptop l = new Laptop();
            l.Brand = "Dell";
            l.Model = "Inspiron 14R";
            l.LaptopDetails();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

8. Interface

❑**An abstract base class with just abstract members is generally used for the interface. All members of an interface must be enforced by each class or structure that implements it. The interface can optionally define default implementations for any or all of its elements.**

```csharp
Program.cs
using System;

namespace MyApplication
{
    // Interface
    interface IAnimal
    {
        void animalSound(); // interface method (does not have a body)
    }

    // Pig "implements" the IAnimal interface
    class Pig : IAnimal
    {
        public void animalSound()
        {
            // The body of animalSound() is provided here
            Console.WriteLine("The pig says: wee wee");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pig myPig = new Pig();  // Create a Pig object
            myPig.animalSound();
        }
    }
}
```

```
The pig says: wee wee
```

9. Polymorphism

❑Polymorphism is the capacity to act in a variety of ways depending on the situation. It is most commonly encountered in applications where many methods with the same name but distinct parameters and behavior are specified.

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

```csharp
using System;

namespace MyApplication
{
    class Animal   // Base class (parent)
    {
        public void animalSound()
        {
            Console.WriteLine("The animal makes a sound");
        }
    }

    class Pig : Animal   // Derived class (child)
    {
        public void animalSound()
        {
            Console.WriteLine("The pig says: wee wee");
        }
    }

    class Dog : Animal   // Derived class (child)
    {
        public void animalSound()
        {
            Console.WriteLine("The dog says: bow wow");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Animal myAnimal = new Animal();  // Create a Animal object
            Animal myPig = new Pig();   // Create a Pig object
            Animal myDog = new Dog();   // Create a Dog object

            myAnimal.animalSound();
            myPig.animalSound();
            myDog.animalSound();
        }
    }
}
```

10. Inheritance

❑ **The act of generating a new class based on the attributes and methods of an existing class is known as inheritance. The current class is referred to as the base class, while the newly generated class is referred to as the derived class. This is a crucial notion in object-oriented programming because it allows inherited properties and functions to be reused.**

```
using System;

namespace MyApplication
{
  class Vehicle  // Base class
  {
    public string brand = "Ford";  // Vehicle field
    public void honk()           // Vehicle method
    {
      Console.WriteLine("Tuut, tuut!");
    }
  }
}
```
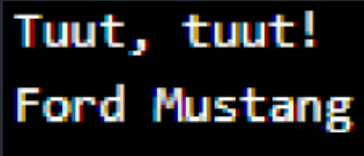
```
using System;

namespace MyApplication
{
  class Car : Vehicle  // Derived class
  {
    public string modelName = "Mustang";  // Car field
  }
}
```

```
using System;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      // Create a myCar object
      Car myCar = new Car();

      // Call the honk() method (from the Vehicle class) on the myCar object
      myCar.honk();

      // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
      Console.WriteLine(myCar.brand + " " + myCar.modelName);
    }
  }
}
```

```
Tuut, tuut!
Ford Mustang
```

11. Abstract class

❑ **The abstract adjustment indicates that the item being modified is absent or incompletely implemented. Classes, processes, properties, indexers, and events can all benefit from abstract modifiers. Use an abstract changer in a class declaration to indicate that a class is intended to be a base class for other classes rather than a standalone object. Members marked as abstract must perform non-abstract classes derived from abstract classes.**

```
//Abstract class can have constant and fields
public abstract class ConstantFields
{
    public int no;
    private const int id = 10;
}
```

Part 2: Design a series of UML class diagrams

II. Design and build class diagrams using a UML tool.

❏ Project Specification: FPT is an international academy and now they want to create an application to store the list of students, faculty, employees, customers, and show the main menu. I perform the segment of managing to in student management system. My team and I will program and develop the student management system for FPT International Academy, and I am the team leader. An application should be designed to show all the students of the school. And display details about student information such as phone number, address, email, DoB...with functions such as search, adding, editing, and deleting customer and employee information as well as students and lecturers. May view all as ID, Name, Phone, Mail, Address.

1. Use-case diagram.

□ **Use case diagram includes:**

Manage Students, Manage Lecturers.

□ **Use-case diagram's notation table.**

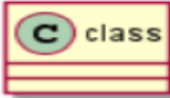| No.	Symbols	Description
1. Actor		The actor is used to referring to a user or an external object interacting with the system we are looking at.
2. Use Case	Show main menu	Use Case is the function that Actors will use.
3. Relationship	«extends»	The term "extends" was used to denote the connection between the two Use Cases. When a Use Case is built to add functionality to an existing Use Case and is utilized under a certain condition, the Extend relation is employed.

Student Management System

2. Class diagram.

❑The class diagram includes the following properties: Person, student, program, lecturer. in the Person, section include ID, Name, DoB, Email, Address. Properties in Person will be set in lecturer and student. IPerson interface has InputForm (), Validate () methods used to check ID, DisplayInfor () The abstract Person class extends the IPerson interface to implement methods. It is also a class inherited by the Student and Instructor classes to add information such as ID, Name, Email, Address, DoB. Teacher class has a Division property and a student class has a Batch property, they work similarly

❑ Class diagram

☐ Class diagram's notation table.

No.	Symbols	Description
1. Extension		Relationship between classes
2. Public		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
3. Private		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
4. Private		When you define methods or fields, you can use characters to define the visibility of the corresponding item.
5. Object		Center circle
6. Interface	interface	Show interface of the diagram.
7. Class	class	Show class of the diagram.
8. Classes	Class name / Attributes / Methods	The fundamental component of the Class Diagram drawing is the class. In the system, a class refers to a set of objects that share the same attributes and behaviors. The class "Customer" is used to describe a customer, for example. Class described includes Class name, properties, and methods.

3. What are Design Patterns?

❑ **Design Patterns are models of code that solve classic problems. They are solutions to software design problems that you can find in a real-world application. A Design Pattern is not a code that is ready to be used in your application, but it is a model that you can use to solve a problem.**
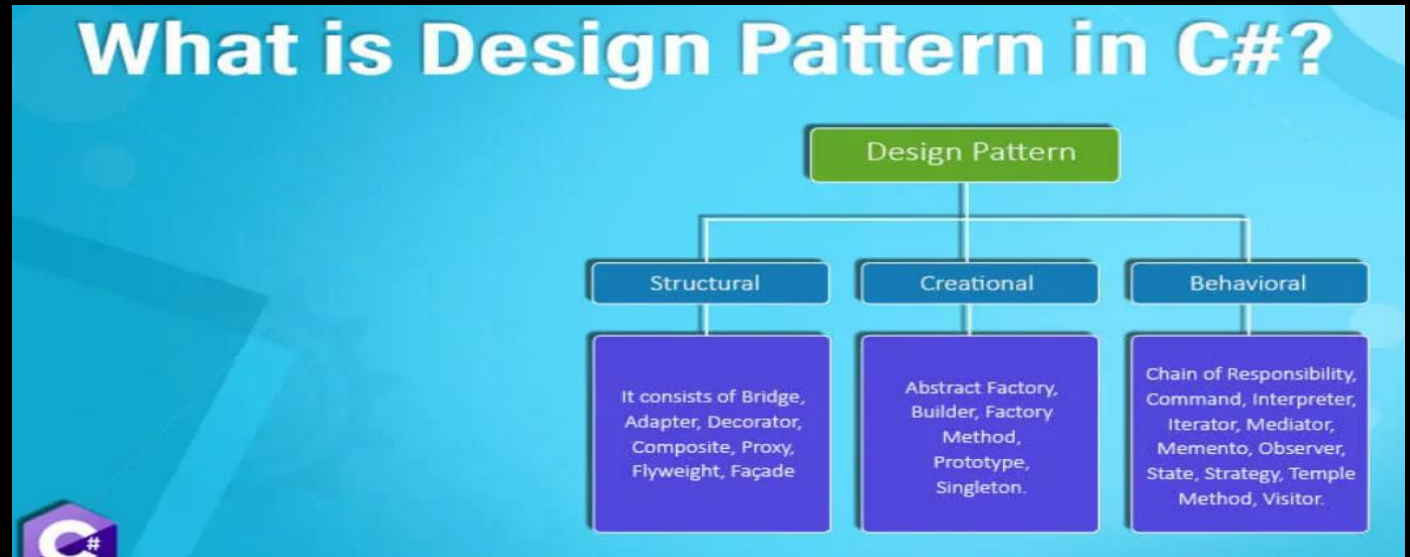
# 4. Introduction to Creational Design Patterns.

❑ **Introduce**

▪ In software engineering, Design Patterns describe building solutions to the most common problems in software design. It represents best practices developed over a long period of time through trial and error by experienced developers.

▪ In this article, we will learn about creational design patterns and their types. We'll also look at a few examples and discuss the conditions under which design patterns are suitable for design.
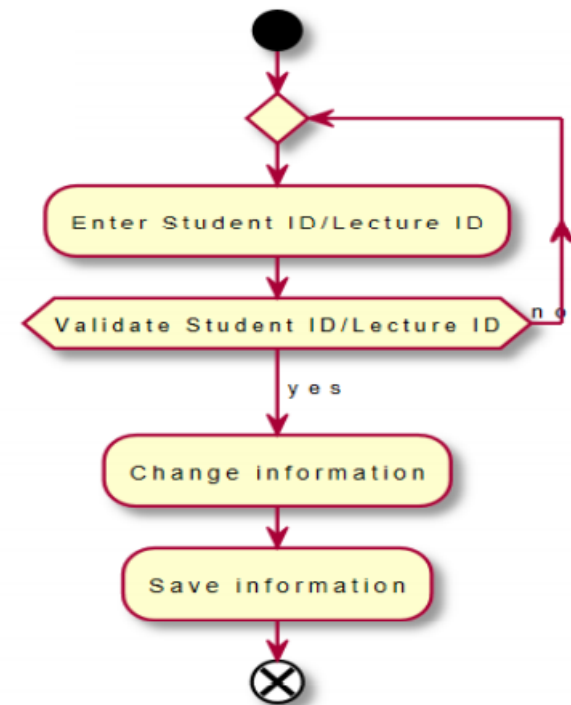
5. Activity Diagram

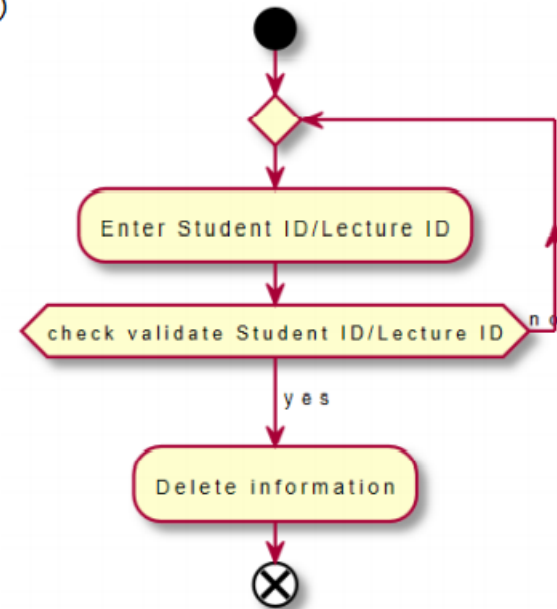## ❑Add Information Student/ Lecture
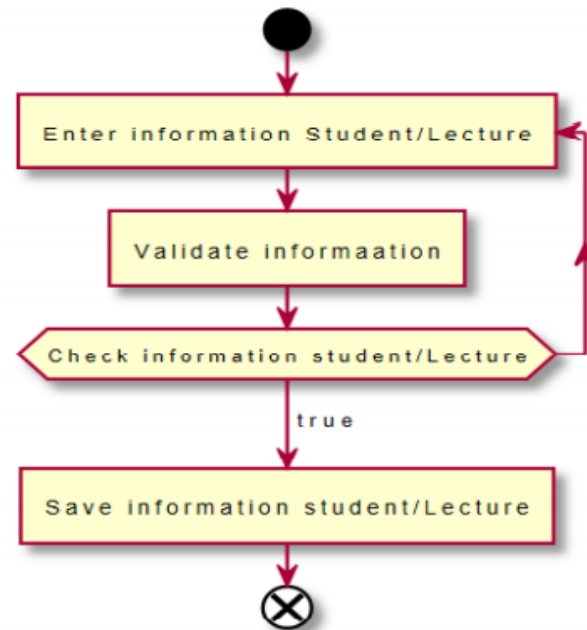
❑Update information Student/Lecture


```
1    @startuml
2    '' plantumlfile2
3    start
4    repeat
5        :Enter Student ID/Lecture ID;
6    repeat while (Validate Student ID/Lecture ID) is (no)
7    -> yes;
8        :Change information;
9        :Save information;
10   end
11
12   @enduml
```
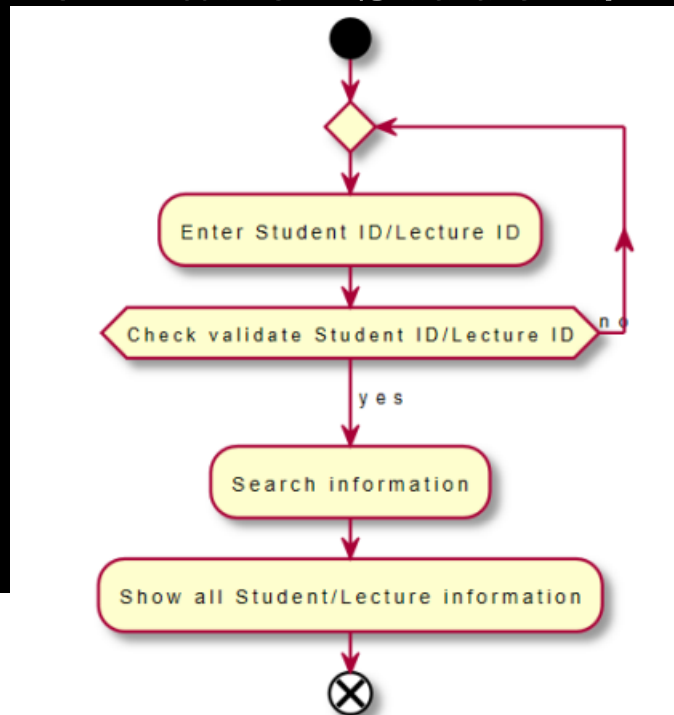
❑ Delete Student/Lecture

```
1    @startuml
2    '' plantumlfile3
3    start
4    repeat
5        :Enter Student ID/Lecture ID;
6    repeat while (check validate Student ID/Lecture ID) is (no)
7    -> yes;
8        :Delete information;
9    end
10
11   @enduml
```

❑Search information Student/Lecture


```
1    @startuml
2    '' plantumlfile4
3    start
4    repeat
5        :Enter Student ID/Lecture ID;
6    repeat while (Check validate Student ID/Lecture ID) is (no)
7    -> yes;
8        :Search information;
9        :Show all Student/Lecture information;
10       end
11
12   @enduml
```

----THE END----

--Thank you to everyone who listened to the report--