Advanced Programming

Assessor name: TRUNG-VIET NGUYEN

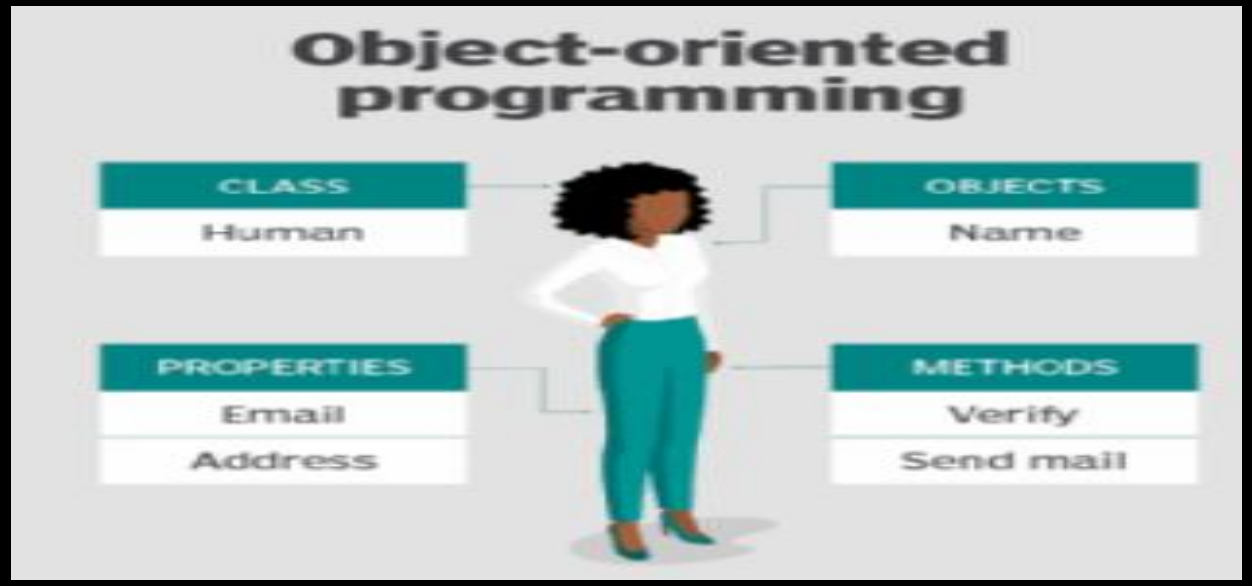
Student Name: PHAM CAO NGUYEN

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.

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.**

```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;
    }
}
```

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..
```

```
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");
        }
```

```
        {
            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.

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
```

```
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.

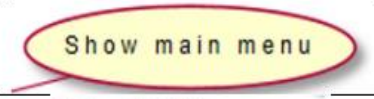
- **Project Specification: FPT is an international academy and now they want to create an application to store the list of students, faculty, employees, customers, and show the main menu. I perform the segment of managing to in student management system. My team and I will program and develop the student 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.**

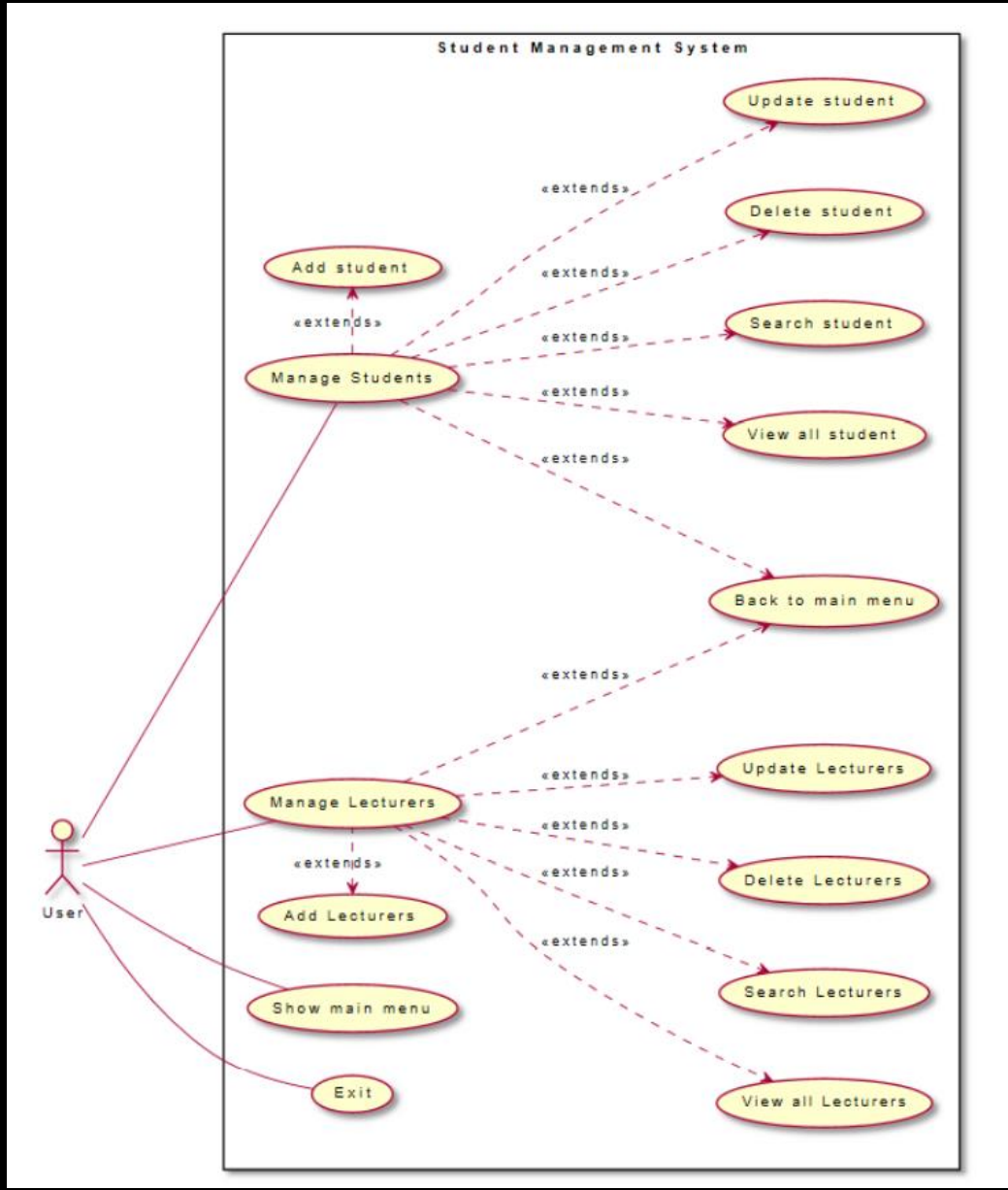
1. Use-case diagram.

- ❑Use case diagram includes:

Manage Students, Manage Lecturers.

- ❑Use-case diagram's notation table.

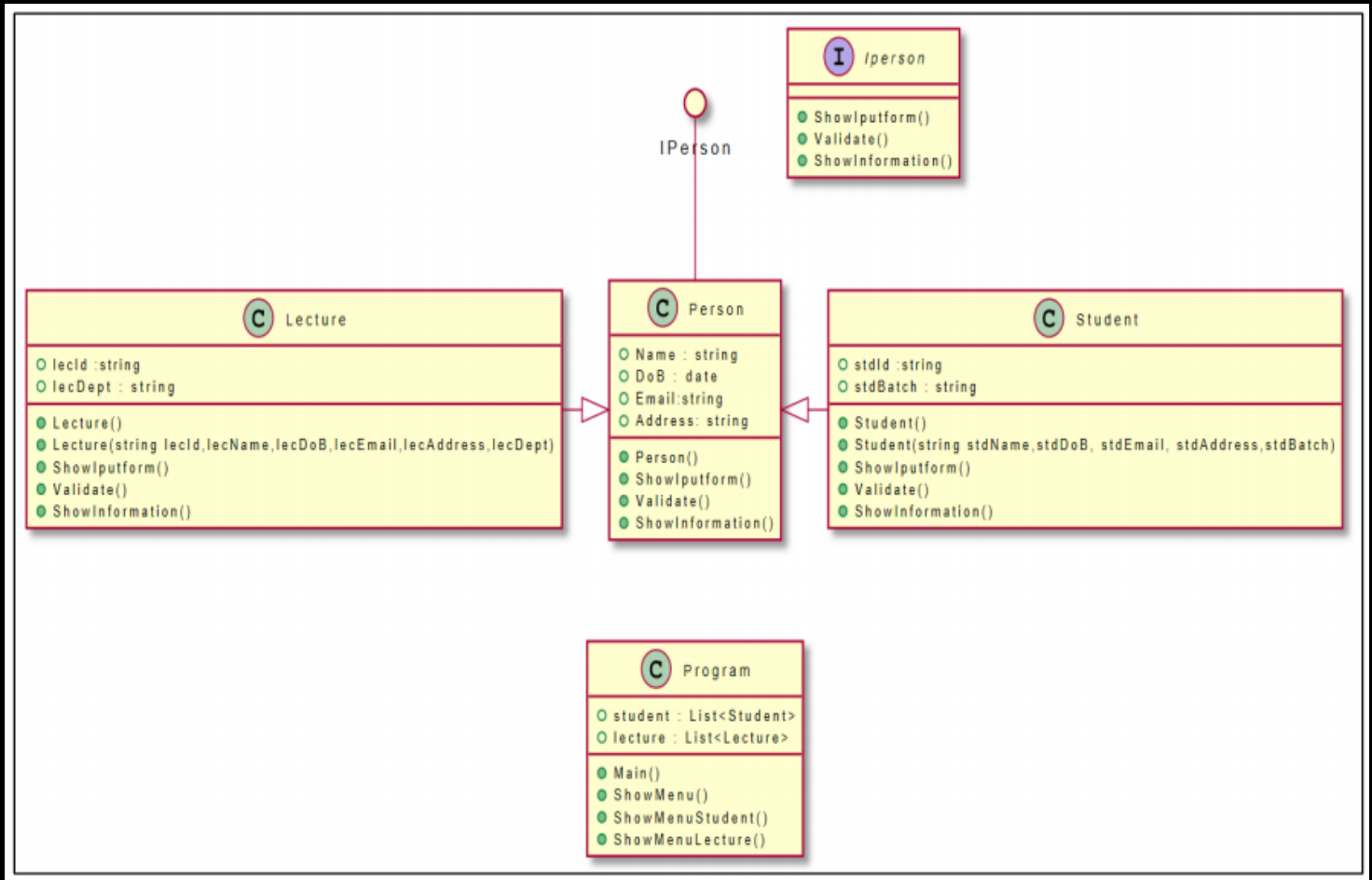
No.	Symbols	Description
1. Actor		The actor is used to referring to a user or an external object interacting with the system we are looking at.
2. Use Case		Use Case is the function that Actors will use.
3. Relationship		The term "extends" was used to denote the connection between the two Use Cases. 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.



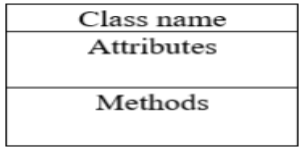
2. Class diagram.

- **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

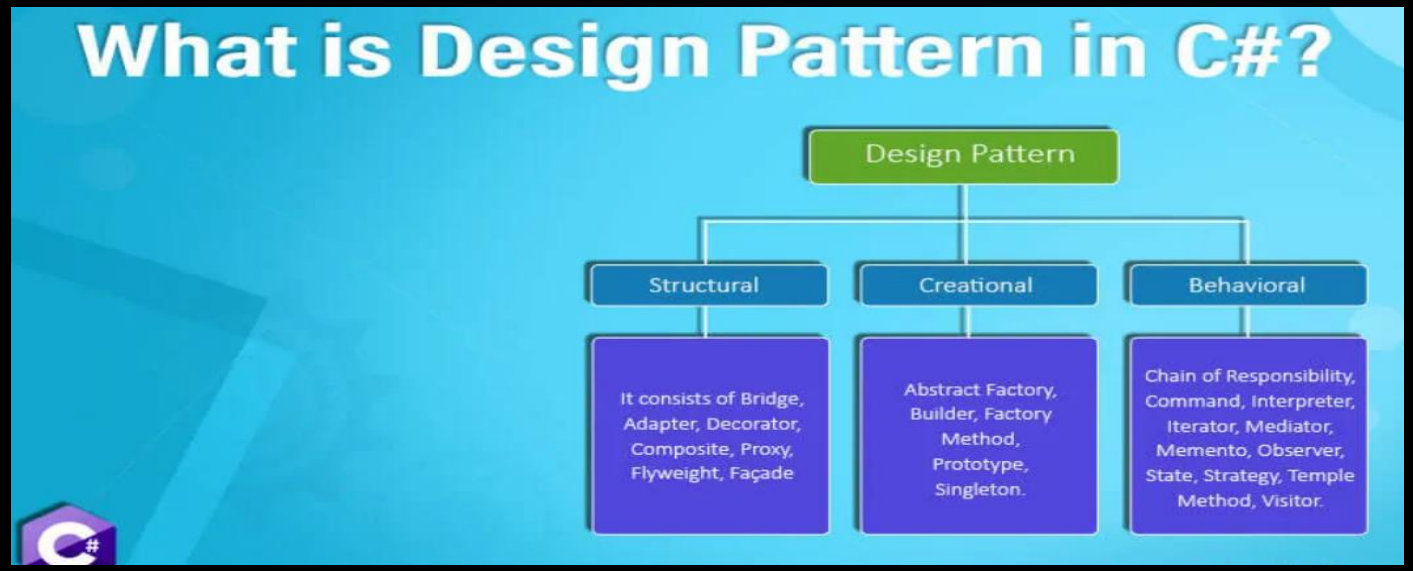


❑ Class diagram's notation table.

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		Show interface of the diagram.
7. Class		Show class of the diagram.
8. Classes		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.

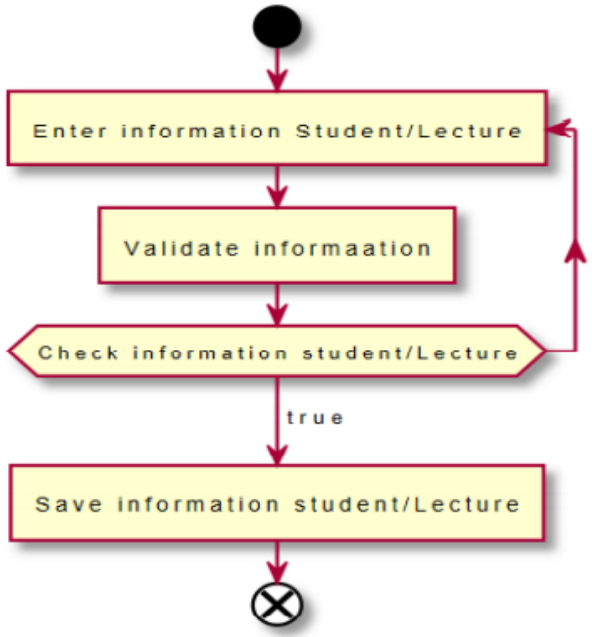
❑ Introduce

- 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

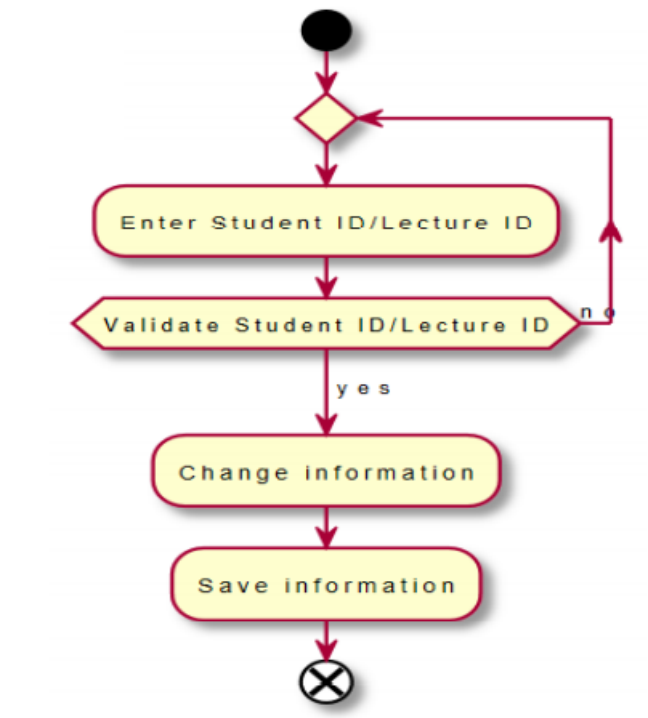
❑Add Information Student/ Lecture

```
@startuml
'' plantumlfile1
start
repeat:Enter information Student/Lecture]
  :Validate informaation]
repeat while (Check information student/Lecture)
->true;
  :Save information student/Lecture]
end
@enduml
```



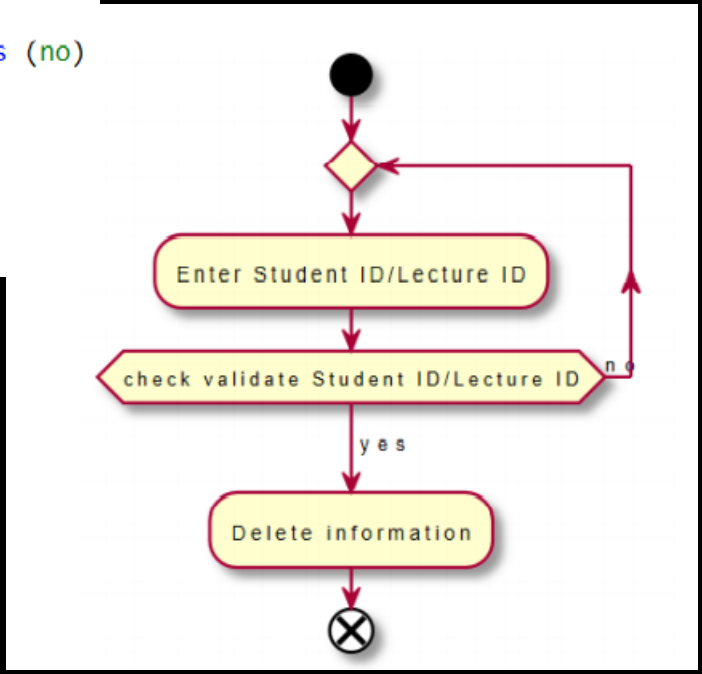
❑Update information Student/Lecture

```
@startuml
'' plantumlfile2
start
repeat
    :Enter Student ID/Lecture ID;
repeat while (Validate Student ID/Lecture ID) is (no)
-> yes;
    :Change information;
    :Save information;
end

@enduml
```

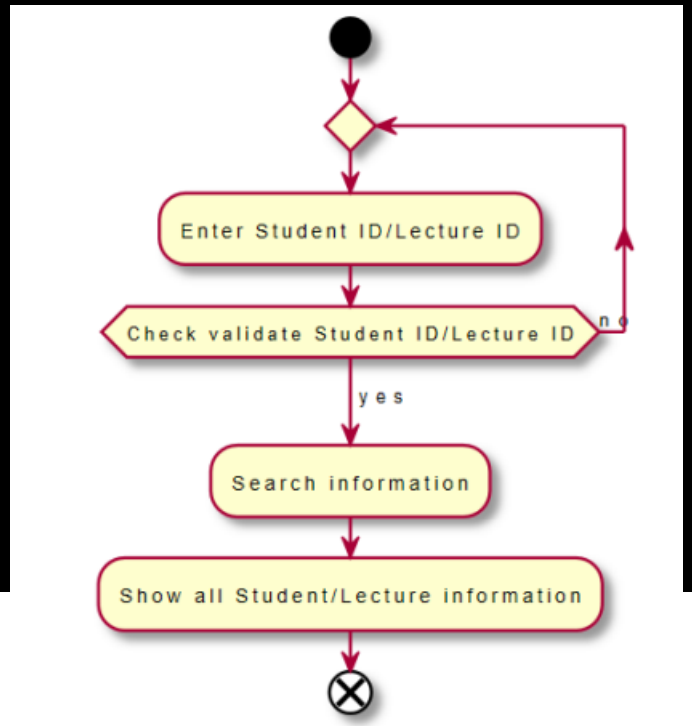


❑Delete Student/Lecture

```
@startuml
'' plantumlfile3
start
repeat
    :Enter Student ID/Lecture ID;
repeat while (check validate Student ID/Lecture ID) is (no)
-> yes;
    :Delete information;
end

@enduml
```



❑Search information Student/Lecture




```
@startuml
'' plantumlfile4
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
```

----THE END----

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