







## ASSIGNMENT 1 FRONT SHEET

<b>Qualification</b>	<b>BTEC Level 5 HND Diploma in Computing</b>		
<b>Unit number and title</b>	Unit 20: Advanced Programming		
<b>Submission date</b>		<b>Date Received 1st submission</b>	
<b>Re-submission Date</b>		<b>Date Received 2nd submission</b>	
<b>Student Name</b>	NGUYEN MINH TU	<b>Student ID</b>	GCC18076
<b>Class</b>		<b>Assessor name</b>	TRUNG-VIET NGUYEN
<b>Student declaration</b> 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.			
		<b>Student's signature</b>	

### Grading grid

 F1	 P2	M1	M2	D1	D2
					

⚙ <b>Summative Feedback:</b>		⚙ <b>Resubmission Feedback:</b>	
<div style="position: relative; height: 100%;"> <div style="position: absolute; top: 10px; left: 10px; background: yellow; border: 1px solid black; padding: 2px;">2.1</div> </div>			
<b>Grade:</b> <div style="position: relative; height: 20px;"> <div style="position: absolute; top: 0; left: 0; background: yellow; border: 1px solid black; padding: 2px;">2.3</div> </div>	<b>Assessor Signature:</b> <div style="position: relative; height: 20px;"> <div style="position: absolute; top: 0; left: 0; background: yellow; border: 1px solid black; padding: 2px;">2.2</div> </div>	<b>Date:</b> <div style="position: relative; height: 20px;"> <div style="position: absolute; top: 0; left: 0; background: yellow; border: 1px solid black; padding: 2px;">2.4</div> </div>	
<b>Lecturer Signature:</b>			

### ASSIGNMENT 1 BRIEF

Unit Number and Title	20: Advance Programming
Academic Year	2018
Unit Tutor	Hoàng Đức Quang
Assignment Title	Assignment 1

Issue Date	
Submission Date	
IV Name & Date	

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

<p><b>Specific requirements</b> <i>(see Appendix for assessment criteria and grade descriptors)</i></p>	<p><b>Scenario</b></p> <p>You've just made a contract with FPT Academy International, and are about to be appointed as a Project Leader for a group of programmers to develop its Student Management System.</p> <p>In this Student Management System, you are required to create an application to store list of students and list of lecturers. Following information are to be stored for each student:</p> <ul style="list-style-type: none"> <li>- stdId: The student ID of the form like GTxxxxx or GCxxxxx (x: is a digit)</li> <li>- stdName: Student name</li> <li>- stdDoB: Student date of birth</li> <li>- stdEmail: Student email</li> <li>- stdAddress: Student address</li> <li>- stdBatch: The batch (class) of the student</li> </ul> <p>Following information are to be stored for each lecturer:</p> <ul style="list-style-type: none"> <li>- lecId: Lecturer ID with 8 digits (fixed)</li> <li>- lecName: Lecturer</li> <li>- lecDoB: Lecturer date of birth</li> <li>- lecEmail: Lecturer email</li> <li>- lecAddress: Lecturer address</li> <li>- lecDept: Lecturer department (e.g., Computing, Business, etc)</li> </ul> <p>This application will need to provide following functionalities via a menu</p> <p>=====</p> <ol style="list-style-type: none"> <li>1. Manage Students</li> <li>2. Manage Lecturers</li> </ol>
---	---

### 3. Exit

=====

Please choose:

When user selects 3, the program will exit.

When user selects 1, the program will display submenu for managing students:

=====

1. Add new student
2. View all students
3. Search students
4. Delete students
5. Update student
6. Back to main menu

=====

Please choose:

When user selects 2, the program will display submenu for managing lecturers:

=====

1. Add new lecturer
2. View all lecturers
3. Search lecturers
4. Delete lecturers

5. Update lecturer
6. Back to main menu

=====

Please choose:

For the submenu:

When user chooses 1, program will prompt user to input student's/lecturer's information (specified previously). After that, program will validate the input data and if they are all valid, program will add a new student/lecturer to the current list of students/lecturers. Program should inform to the user corresponding messages.

When user chooses 2, the program will list all the students/lecturers to the screen, each student/lecturer in a row and student's/lecturer's data fields are separated by '|'.

When user chooses 3, the program will ask user to input student's/lecturer's name to search for, the user can just type part of the name in order to search for complete student/lecturer information.

When user chooses 4, program will ask user to input student/lecturer id to delete the student/lecturer with the specified id if it exists, otherwise, it will display a message to inform users that the student/lecturer with such id doesn't exist.

When user chooses 5, program will first ask user to input student/lecturer id to update, once inserted and a student/lecturer with the inserted id exists, it will display current data for each field of the

student/lecturer and user can type in new data to update or just press enter to keep the current data for the field.

When user chooses 6, program will back to the main menu.

### **Task 1 part1**

Produce a written, self-learning course for managers/senior developers that explain the principles and features of OOP and that show how OOP is good for code re-use. You can include appropriate sample Java code where it aids your points and/or provides further clarification. Ensure that any diagrams that are included have captions and are referenced in the text.

Hint: You must include the following terms

- Object/Class,
- Abstraction,
- Encapsulation,
- Inheritance,
- Polymorphism (Overloading and Overriding),
- Abstract classes,
- Interfaces

### **Task 2**

Problem Analysis, for the scenario above.

You need to produce a full design for the requirements given. The design must include

- Use-case diagrams for the most important features;
- Class diagrams for all objects identified as well as class relationships.

	<ul style="list-style-type: none"><li>- Pseudo-code for the Algorithms for the main functionalities (3 flowcharts for most complex functions).</li></ul> <p>Example code can be used to help clarify OOP features.</p>
<b>Student guidelines</b>	<p>For the assignment assessments, you are required to:</p> <ol style="list-style-type: none"><li>1. Produce a design in UML that fully utilizes OOP principles and features. (Use case, class diagrams, collaboration diagrams etc.).</li></ol>
<b>Submission requirements</b>	<p>Students are expected to submit hard copy of assignment</p>



## Grade Descriptor

### PASS criteria

LO	Learning outcome (LO)	AC	In this assessment you will have the opportunity to present evidence that shows you are able to:	Task no.
LO1	Examine the key components related to the object-orientated programming paradigm, analysing design pattern types	1	Examine the characteristics of the object-orientated paradigm as well as the various class relationships.	1
LO2	Design a series of UML class diagrams		Design and build class diagrams using a UML tool.	

In addition to the above PASS criteria, this assignment gives you the opportunity to submit evidence in order to achieve the following MERIT and DISTINCTION grades		
Grade Descriptor	Indicative characteristic/s	Contextualization
M1	Determine a design pattern from each of the creational, structural and behavioural pattern types.	

M2	Define class diagrams for specific design patterns using a UML tool.	
D1	Analyse the relationship between the object-orientated paradigm and design patterns.	
D2	Define/refine class diagrams derived from a given code scenario using a UML tool.	

This brief has been verified as being fit for purpose					
<b>Internal Verifier 1</b>		Signature		Date	
<b>Internal Verifier 2</b>		Signature		Date	

## Table of Contents

<b>ASSIGNMENT 1 BRIEF .....</b>	<b>2</b>
<b>Grade Descriptor .....</b>	<b>9</b>
<b>PASS criteria .....</b>	<b>9</b>
<b>I. Produce a written, self-learning course for managers/senior developers that explain the principles and features of OOP and that show how OOP is good for code re-use. You can include appropriate sample Java code where it aids your points and/or provides further clarification. Ensure that any diagrams that are included have captions and are referenced in the text. Hint: You must include the following terms .....</b>	<b>12</b>
1. What is object oriented programming? .....	12
2. Object/Class .....	12
3. Abstraction .....	13
4. Encapsulation .....	14
5. Inheritance .....	16
6. Polymorphism (Overloading and Overriding) .....	18
7. Abstract classes .....	18
8. Interfaces .....	19
<b>II. Problem Analysis, for the scenario above. You need to produce a full design for the requirements given. The design must include</b>	<b>20</b>
1. Use-case diagram .....	20
1. Class diagram .....	23
2. The Algorithms for the main functionalities .....	27
<b>REFERENCES .....</b>	<b>31</b>

- I. Produce a written, self-learning course for managers/senior developers that explain the principles and features of OOP and that show how OOP is good for code re-use. You can include appropriate sample Java code where it aids your points and/or provides further clarification. Ensure that any diagrams that are included have captions and are referenced in the text. Hint: You must include the following terms**

**1. What is object oriented programming?**

Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of "objects" that can hold both data and code: data in the form of fields (also known as attributes or properties) and code in the form of procedures (often referred to as methods)[1].

One of the characteristics of objects is that their proper procedures will enter and modify data fields on their own (objects have a notion of "this" or "self"). OOP is a programming language that allows you to create computer programs out of objects that interact with one another. The most popular OOP languages are class-based, which means that objects are class instances that often describe their types.

Many of the most widely used programming languages ( such as C++, Java , Python, etc.) are multi-paradigm and to a greater or lesser extent they support object-oriented programming, typically in combination with imperative, procedural programming. Big object-oriented languages include Java, C++, C#, Python, R, PHP, Visual Basic.NET, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Kotlin, Common Lisp, MATLAB, and Smalltalk.

**2. Object/Class**

**What is a Object?**

A class instance is referred to as an object. In OOPS, an object is a self-contained item that contains methods and properties that make a certain class of data useful. For instance, a color name, a table, a bag, or barking. When you send a message to an object, you're telling it to call or execute one of the object's methods, which are specified in the class.

An object in OOPS may be a data structure, a vector, or a function from a programming standpoint. It has a memory location set aside for it. Java Objects are organized into hierarchical classes.

**What is a Class?**

A blueprint or collection of instructions for building a particular type of object is referred to as a class. It is an Object-Oriented Programming philosophy that revolves around real-life entities. In Java, a class defines how an object will behave and what it will contain.

Example:

```
01. class Employee
02. {
03.
04. }
```

Figure 1:Class

### 3. Abstraction

One of the four major characteristics of object-oriented programming is abstraction (OOP). Its key aim is to simplify things by hiding information that isn't immediately important to the customer (the user is not the end user, but the programmer). This helps the user to complete necessary tasks against an arbitrary object without having to comprehend or even consider any of the secret complexities.

Really, the term "subtraction" refers to a common principle that is applied in all aspects of existence, not just programming.

Advantages of Abstraction:

- + It simplifies the process of seeing objects.
- + Reduces 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.

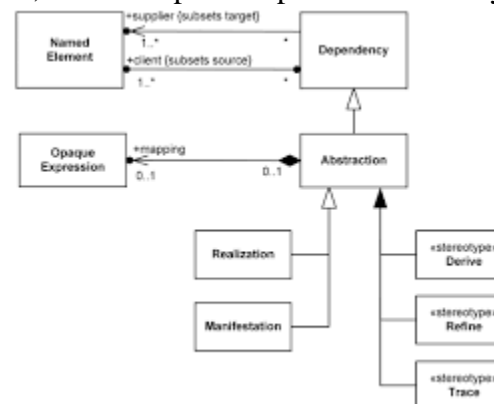


Figure 2: Abstraction

Example :

Real-world Example of Abstraction

Suppose you have Mobile Phone object.

Assume you've got 3 cell phones, as follows:

Nokia 1280 (Features: Calling, SMS)

Nokia 2700 (Features: Calling, SMS, FM Radio, MP3, Camera)

Black Berry (Features: Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)

For the object "Cell Phone," abstract information (necessary and common knowledge) is that it makes a call to any number and is

able to send text messages.

So that you will have the abstract class for a cell phone object as in the following,

```
01. abstract class MobilePhone {  
02.     public void Calling();  
03.     public void SendSMS();  
04. }  
05. public class Nokia1400: MobilePhone {}  
06. public class Nokia2700: MobilePhone {  
07.     public void FMRadio();  
08.     public void MP3();  
09.     public void Camera();  
10. }  
11. public class BlackBerry: MobilePhone {  
12.     public void FMRadio();  
13.     public void MP3();  
14.     public void Camera();  
15.     public void Recording();  
16.     public void ReadAndSendEmails();  
17. }
```

Figure 3: Example Abstraction

Abstraction means bringing all the necessary variables and methods into a class.

#### 4. Encapsulation

One of the key principles of object-oriented programming is encapsulation (OOP). It refers to the concept of combining data and methods that operate on that data into a single entity, such as a Java class.

Advantages of Encapsulation:

The most significant benefit of encapsulation is data protection. The following are some of the advantages of encapsulation:

- + 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.

Object data can almost always be limited to private or covered for the best encapsulation. If you want to set the access level to public, make sure you're aware of the consequences.

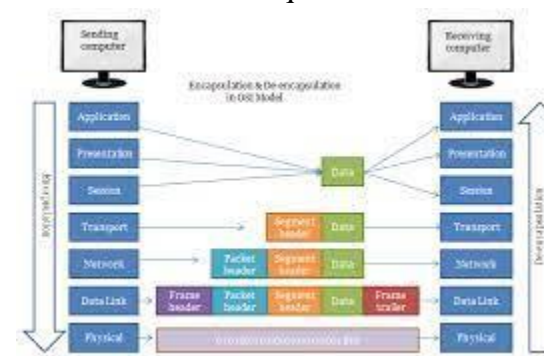


Figure 4: Encapsulation

Example Encapsulation:

```

1 public class Coat {
2     private double price;
3     private String customer;
4
5     public double getPrice() {
6         return price;
7     }
8
9     public void setPrice(double price) {
10        this.price = price;
11    }
12
13    public String getCustomer() {
14        return customer;
15    }
16
17    public void setCustomer(String customer) {
18        this.customer = customer;
19    }
20 }

```

Abstraction	Encapsulation
Abstraction addresses the issue in terms of architecture.	Encapsulation solves the Implementation Level problem.
Abstraction conceals undesirable data and gives important data.	Encapsulation involves separating the code and data from the outside world into a single unit to encrypt the data
Abstraction helps you to concentrate on what the object does, rather than how it does it	Encapsulation means concealing the internal information or mechanisms of how such an entity does
Abstraction: Outer layout, used in terms of design. For example: A Cell Phone external as it has a display screen and buttons for dialing a call	Encapsulation- Inner layout, used in terms of implementation. For example: Internal Cell Phone information, how to attach the keypad button and the display screen to each other using circuits.

## 5. Inheritance

An inheritance is a financial concept that refers to the properties that are handed on to people when they die. The majority of inheritances are cash, but they can also include securities, shares, houses, jewels, automobiles, paintings, antiques, real estate, and other tangible assets.

Advantages:

- + One of the many advantages of inheritance is that it reduces the amount of redundant code in an application by allowing subclasses to share similar code. When two similar classes have the same code, the hierarchy will normally be refactored to transfer the shared code to a joint superclass. This also leads to greater code organization and fewer, more straightforward compilation modules.
- + Since classes that inherit from a similar superclass can be used interchangeably, inheritance can allow program code more resilient to modify. If a method's return form is superclass.
- + Reusability refers to the ability to call base case public methods without having to rewrite them. Extensibility refers to the ability to expand the logic of the base class to meet the needs of the derived class. Data hiding — a base class may attempt to keep any data secret so that the derived class cannot change it.
- + Overriding—We will use inheritance to override the methods of the base class so that the derived class can design a meaningful application of the base class method.



# Inheritance

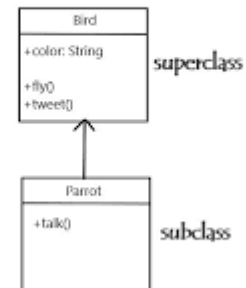


Figure 5: Inheritance

Example: Subclass: Cat

```
package com.journaldev.inheritance;

public class Cat extends Animal{

    private String color;

    public Cat(boolean veg, String food, int legs) {
        super(veg, food, legs);
        this.color="White";
    }

    public Cat(boolean veg, String food, int legs, String
color){
        super(veg, food, legs);
        this.color=color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

}
```

Figure 6: Example Inheritance

## 6. Polymorphism (Overloading and Overriding)

Polymorphism refers to the use of a single interface to multiple types of entities, or the use of a single symbol to represent multiple types of entities. I will train my dog to respond to a command bark and my bird to respond to command calls, for example. In the other hand, I will teach them so that they both obey the order. Polymorphism has taught me that the dog will bark and the bird will yell in response.

The following are the most well-known primary polymorphism groups:

- + Ad hoc polymorphism: Sets a standard interface for an arbitrary set of specified types.
- + Parametric polymorphism: If one or more types are not defined by name but by abstract symbols of some kind.
- + Subtyping (also called subtype polymorphism or inclusion polymorphism): When a name denotes instances related to some common superclass of many different classes.

Example:

```
public class Demo
{
    public static void main(String[] args) {
        Animal a1 = new Cat();
        a1.makeNoise(); //Prints Meowoo

        Animal a2 = new Dog();
        a2.makeNoise(); //Prints Bark
    }
}
```

## 7. Abstract classes

A entity that is declared with the abstract keyword is known as an abstract class. An abstract class is a partly implemented class that is used to develop some of the operations of an object that are shared by all sub-classes at the next level. As a result, it encompasses both abstract and explicit approaches, such as variables, properties, and indexers.

It is often generated as a super class next to an interface in the object inheritance hierarchy to execute standard operations from an interface. An abstract class may or may not have abstract methods. However, if a class contains an abstract method, it must be called abstract.

An abstract object cannot be directly instantiated. It is required to construct / derive a subclass from the abstract class in order to provide functionality to their abstract functions.

### 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 not accept multiple inheritance by an abstract class.

Example:

```
01. #region
02. //Abstract class can have constant and fields
03. public abstract class ConstantFields
04. {
05.     public int no;
06.     private const int id = 10;
07. }
08. #endregion
```

Figure 7:Example Abstract class

## 8. Interfaces

An interface is a programming structure/syntax that enables a machine to impose specific properties on an object (class).

When to use an interface: To the new developer this is a very critical issue. We do have a core banking system , for example. We all know the banking data is highly sensitive. A little carelessness is risky. So if the bank decides to develop a third party developer mobile banking web application then we can't provide full access to our core banking application. So in our core banking application, we have to design the interface. Also you could use a DLL.

The other developers in the main application will use that DLL and submit data for transformation. Third-party developers access only the core banking application with the restricted rights in the interface that we have provided them. And under thesesituations, the interface is very useful.

Why use an interface:

- +Create loosely coupled software
- +Support design by contract (the entire interface must be provided by an implementer)
- +Enable plug-in apps

- +Have artifacts easy to communicate
- +Hide implementation knowledge of each other's classes
- +Facilitating software reuse

Example Interface

```
class Books extends Periodicals  
implements Periodicals.getPeriodicals {  
    public void getPublisher() {  
        //get the publisher  
    }  
    public void getPublishDate() {  
        //get the publish date  
    }  
}
```

## II. Problem Analysis, for the scenario above. You need to produce a full design for the requirements given. The design must include

### 1. Use-case diagram

Use case diagram includes: student section and lecture section, student section including delete students, add new student, view all students, update student, search students and back to main menu will supplement management student. section for lecturers including delete lecturers, add new lecturer, view all lecturer, update lecturer, search lecturers and back to main menu will support management lecturers. exit information will be d login to device, main menu

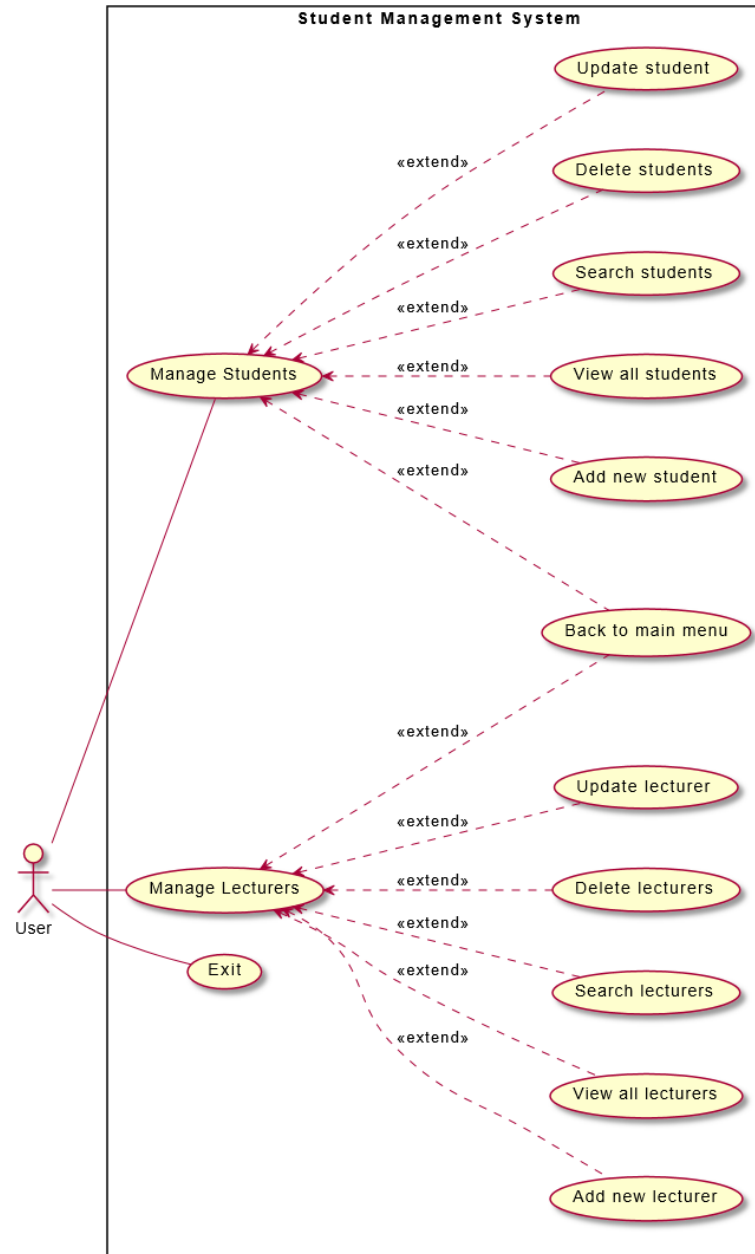


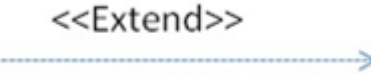


Figure 8: Use case diagram

## Use case diagram's notation

No.	Symbols	Description
1. Actor		Actor is used to refer 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		Extend used to describe the relationship between 2 Use Cases. The Extend relation is used when a Use Case is created to add functionality to an existing Use Case and is used under a certain condition.

## The code of the Use Case diagram

```
@startuml
left to right direction
skinparam packageStyle rectangle
actor User

rectangle "Student Management System" {
    User -- (Manage Students)
    User -- (Manage Lecturers)
    User -- (Exit)
    (Manage Students) <.. (Add new student) : <<include>>
    (Manage Students) <.. (View all students) : <<include>>
    (Manage Students) <.. (Search students) : <<include>>
    (Manage Students) <.. (Delete students) : <<include>>
    (Manage Students) <.. (Update student) : <<include>>
    (Manage Students) <.. (Back to main menu) : <<include>>

    (Manage Lecturers) <.. (Add new lecturer) : <<include>>
    (Manage Lecturers) <.. (View all lecturers) : <<include>>
}
```

```
(Manage Lecturers) <!-- (Search lecturers) : <<include>>
(Manage Lecturers) <!-- (Delete lecturers) : <<include>>
(Manage Lecturers) <!-- (Update lecturer) : <<include>>
(Manage Lecturers) <!-- (Back to main menu) : <<include>>

}
```

@endum1

Actor operates with main functions as Management Students, Management Lecturers, and Exit.

Management Student has the following functions:

- + Add new student
- + Update student
- + Delete students
- + Search students
- + View all students
- + Back to main menu

The Management Lecturer also has similar functions:

- + Add new lecturer
- + Update lecturer
- + Delete lecturers
- + Search lecturer
- + View all lecturers
- + Back to main menu

## 1. 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.

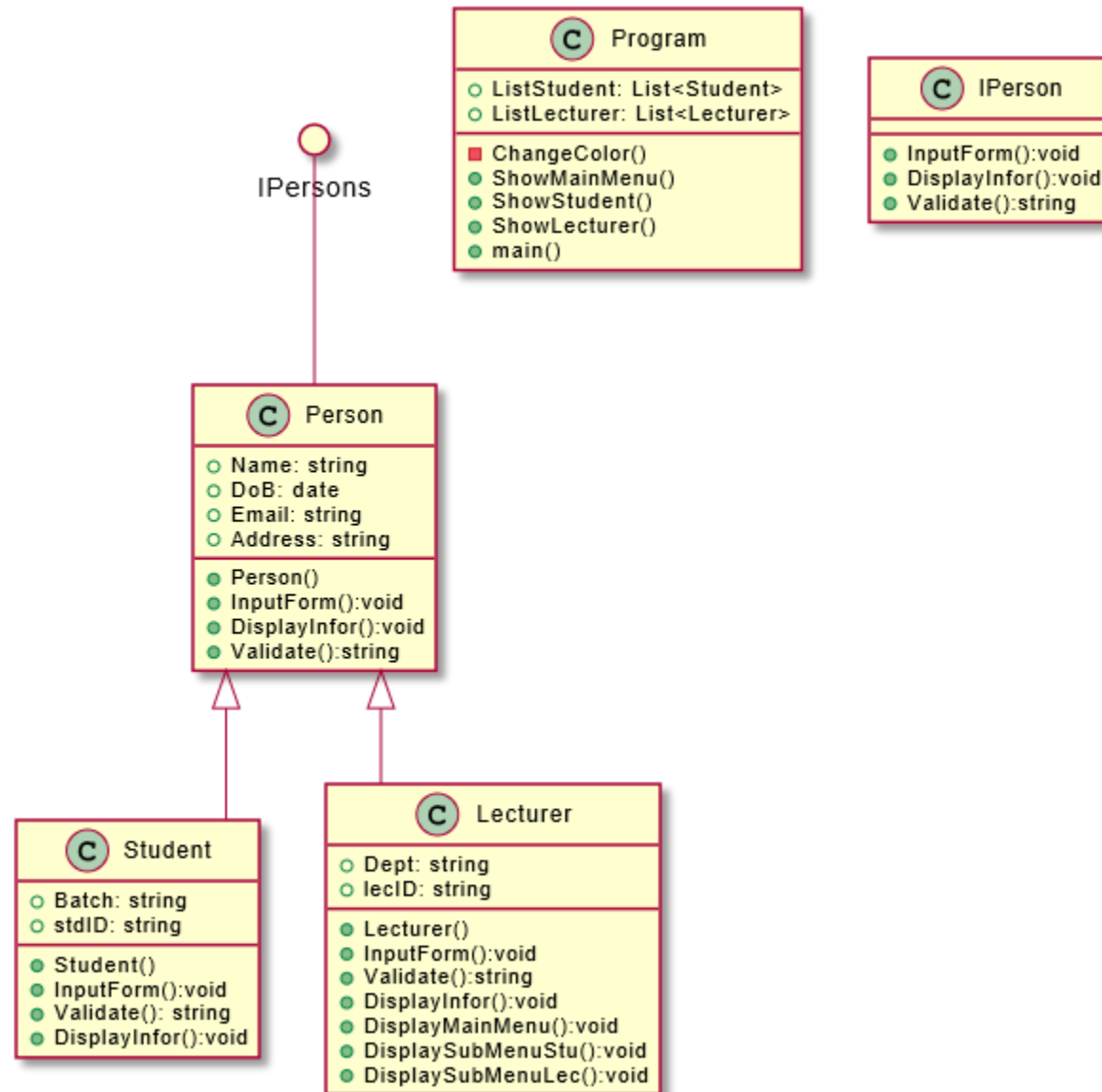



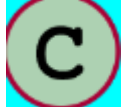


Figure 9: Class diagram



### Class diagram's notation

No.	Symbols	Description
1. Extension		Relationship between classes
2. Public	Icon for method ● , Icon for field ○	When you define methods or fields, you can use characters to define the visibility of the corresponding item
3. Private	Icon for method ■ , Icon for field □	
4. Classes	<div> <div>Class Name</div> <div>Attributes</div> <div>Methods</div> </div>	Class is the main component of the Class Diagram drawing. Class describes a group of objects with the same properties and actions in the system. For an example to describe the customer we use the class "Customer". Class described includes Class name, properties and methods.
5. Object		center circle
6. Interface		show interface of diagram
7. Class		show class of diagram

The code of the Class diagram:

```
@startuml
skinparam linetype ortho
class Person
class Student
class Lecturer
class Program
```

```
class IPerson
```

```
circle IPersons
```

```
Person <|-- Student
```

```
Person <|-- Lecturer
```

```
IPersons -- Person
```

```
class Person{
```

```
+Name: string
```

```
+DoB: date
```

```
+Email: string
```

```
+Address: string
```

```
}
```

```
Person : +Person()
```

```
Person : +InputForm():void
```

```
Person : +DisplayInfor():void
```

```
Person : +Validate():string
```

```
class Lecturer{
```

```
    +Dept: string
```

```
    +lecID: string
```

```
}
```

```
Lecturer : +Lecturer()
```

```
Lecturer : +InputForm():void
```

```
Lecturer : +Validate():string
```

```
Lecturer : +DisplayInfor():void
```

```
class Student{
```

```
    +Batch: string
```

```
    +stdID: string
```

```
}
```

```
Student : +Student()
```

```
Student : +InputForm():void
```

```
Student : +Validate(): string
```

```
Student : +DisplayInfor():void
```

```
class Program{
```

```
    +ListStudent: List<Student>
```

```
    +ListLecturer: List<Lecturer>
```

```
    - ChangeColor()
```

```
    + ShowMainMenu()
```

```
    + ShowStudent()
```

```
    +ShowLecturer()
```

```
+ main()
}
Lecturer : +DisplayMainMenu():void
Lecturer : +DisplaySubMenuStu():void
Lecturer : +DisplaySubMenuLec():void

IPerson : +InputForm():void
IPerson : +DisplayInfor():void
IPerson : +Validate():string
@enduml
```

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.

## 2. The Algorithms for the main functionalities

### Flowchart Update Lecturer function

Flowchart diagram's notation

No.	Symbols
1. The terminator symbol represents the starting or ending point of the system.	
2. Process: A box indicates some particular operation.	

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.

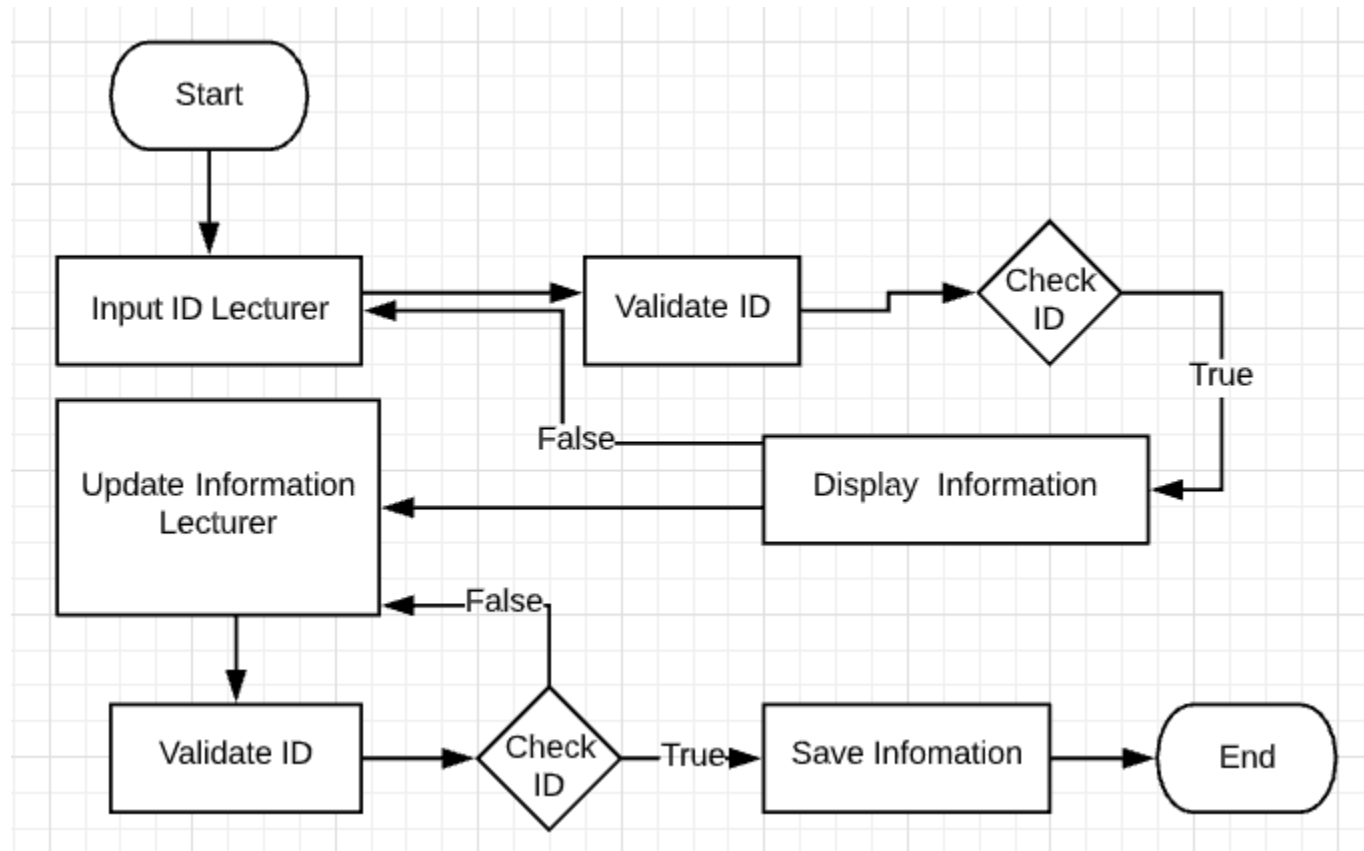


Figure 10: Flowchart Update Lecturer function

### Flowchart Add function

Input information and validate information. If the information is incorrect, it will return input information. If the information is correct, go to the information display the user can change the information. The changed information will be saved. Such execution is the end.

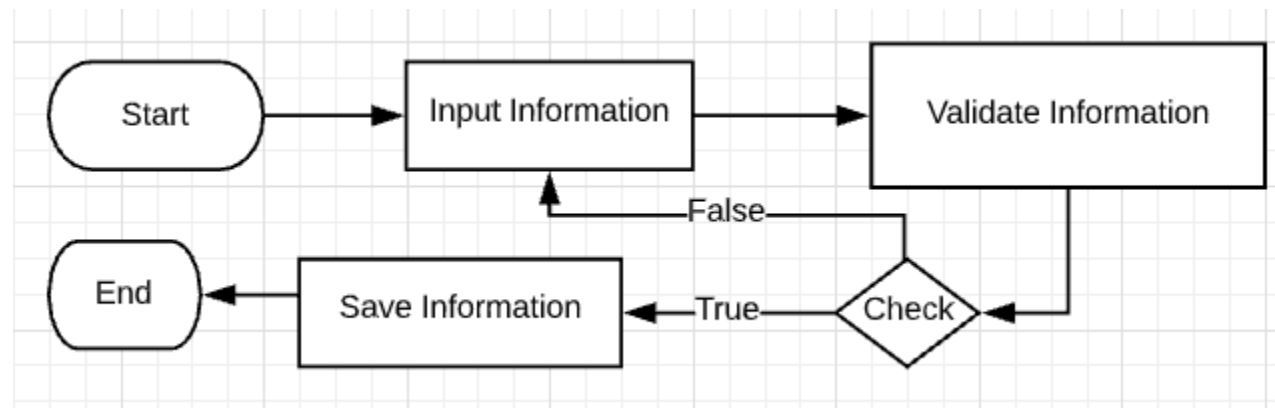


Figure 11:Flowchart Add function

### Flowchart Delete information of the student function

Input student ID and validate ID. if the ID is incorrect it will return student ID input.If the ID is correct the user can change the information and delete information .Such execution is the end.

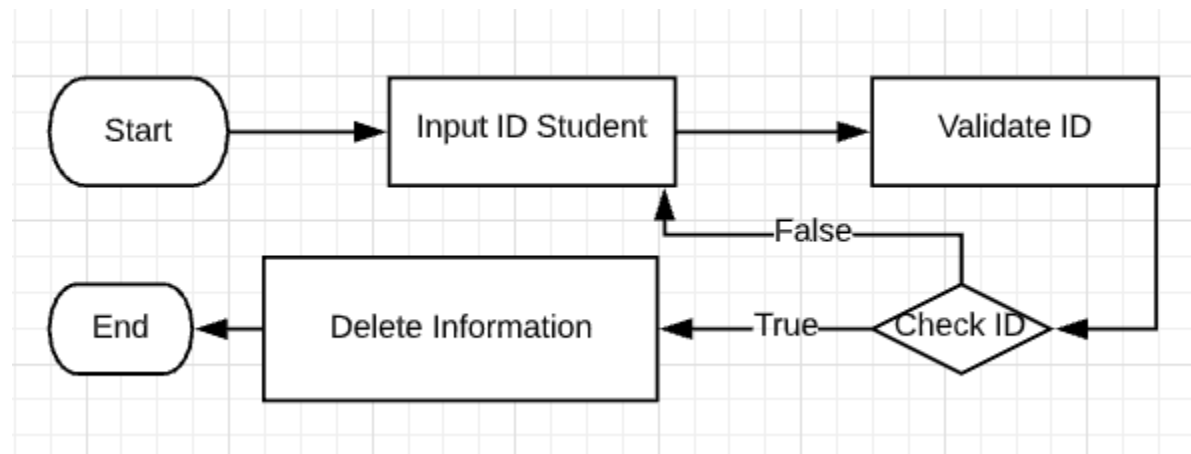


Figure 12: Flowchart Delete information

## REFERENCES

- [1 ]Madsen, O.L., Møller-Pedersen, B. and Nygaard, K., 1993. Object-oriented programming in the BETA programming language. Addison-Wesley
- Finkel, R.A. and Finkel, R.A., 1996. *Advanced programming language design* (p. 372). Reading: Addison-Wesley.
- Stevens, W.R. and Rago, S.A., 2008. *Advanced programming in the UNIX environment*. Addison-Wesley.
- Bitter, R., Mohiuddin, T. and Nawrocki, M., 2017. *LabVIEW: Advanced programming techniques*. CRC press.

## Index of comments

---

- 2.1 You have clearly and detailed description of the object-oriented model's features. You do not analyze relationship between object oriented models and design patterns.
- 2.2 Trung-Viet Nguyen
- 2.3 Pass
- 2.4 28/04/2021