UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

Higher Nationals in Computing

Unit 20: Advanced Programming

ASSIGNMENT 1

Learner's name: NGUYEN TIEN THANH

ID: GCS190601

Class: GCS0805A_ppt

Subject code: 1651

Assessor name: **PHAN MINH TAM**

Assignment due: Assignment submitted:

ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	**Unit 20: Advanced Programming**		
Submission date	June 12, 2021	**Date Received 1st submission**	
Re-submission Date	June 15, 2021	**Date Received 2nd submission**	
Student Name	NGUYEN TIEN THANH	**Student ID**	GCS190601
Class	GCS0805A_ppt	**Assessor name**	Phan Minh Tam
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	thanh

Grading grid

P1	P2	M1	M2	D1	D2

☼ Summative Feedback:	☼ Resubmission Feedback:	
Grade:	**Assessor Signature:**	**Date:**
Signature & Date:		

ASSIGNMENT 1 BRIEF

Qualification	**BTEC Level 5 HND Diploma in Business**		
Unit number	Unit 20: Advanced Programming		
Assignment title	Examine and design solutions with OOP and Design Patterns		
Academic Year			
Unit Tutor			
Issue date		**Submission date**	June 12, 2021
IV name and date			

Submission Format:

Format: The submission is in the form of a **group written report and presentation**. 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.

UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

Table of Contents

Table of Figures

ASSIGNMENT 1 ANSWERS

P1 Examine the characteristics of the object-orientated paradigm as well as the various class relationships.

1. Object-orientated paradigm definition

Object-Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages:

- Modularisation: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

An object-oriented programming language generally supports five main features:

- Classes and Objects
- Encapsulation
- Abstraction
- Polymorphism
- Inheritance

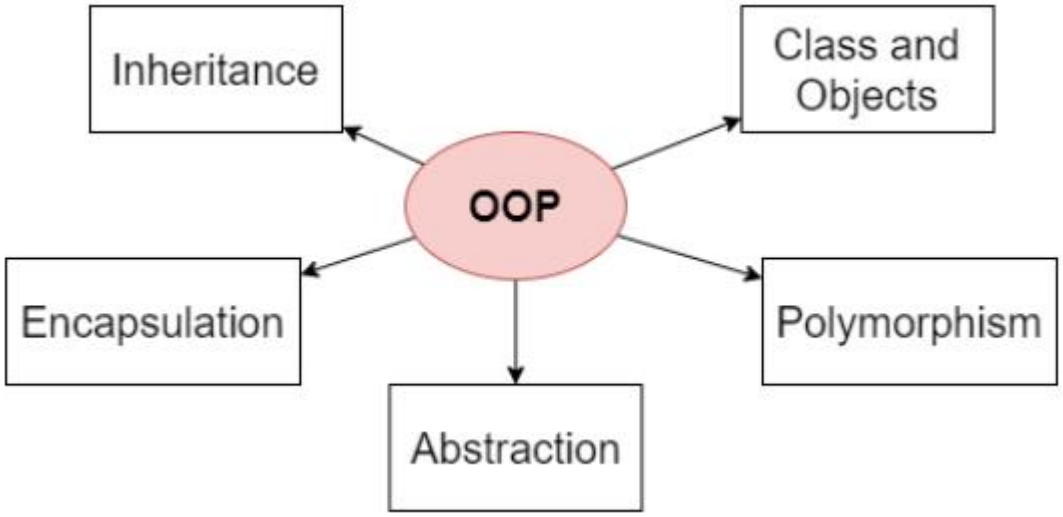

Figure 1 Object-orientated paradigm definition

2. The characteristics of the object-orientated paradigm

2.1. Class and Object

2.1.1. Class

With a functional programming language (like C) we would have the component parts of the television scattered everywhere and we would be responsible for making them work correctly - there would be no case surrounding the electronic components.

Classes allow us a way to represent complex structures within a programming language. They have two components:

- **States** - (or data) are the values that the object has.
- **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

2.1.2. Objects

An object is an instance of a class. You could think of a class as the description of a concept, and an object as the realisation of this description to create an independent distinguishable entity. For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realisation of these plans into a real-world physical television. So there would be one set of plans (the class), but there could be

thousands of real-world televisions (objects).

For example, if the channel is changed on one television it will not change on the other televisions.

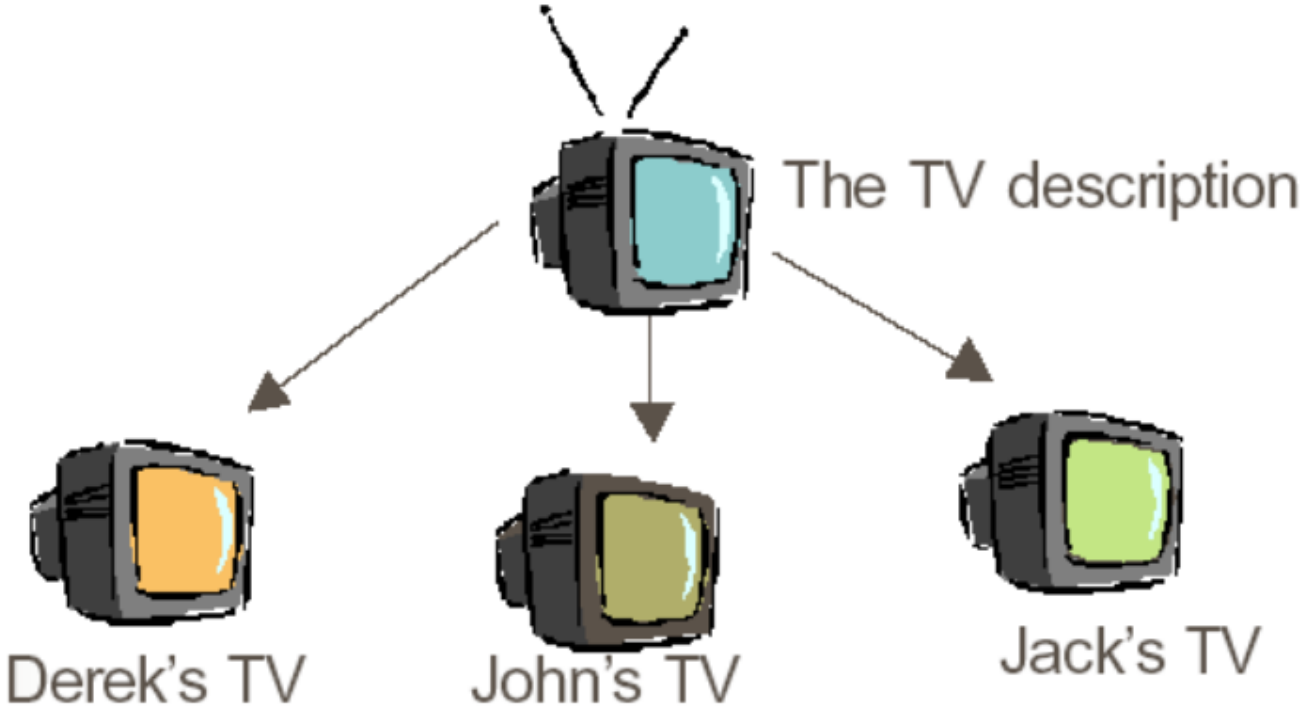

Figure 2 Example of Class and Objects

2.2. Inheritance

If we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions. The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.

Humans use this concept in categorising objects and descriptions. So we can describe this relationship as a child/parent relationship, where Figure illustrates the relationship between a base class and a derived class. A derived class inherits from a base class, so in Figure the Car class is a child of the Vehicle class, so Car inherits from Vehicle.

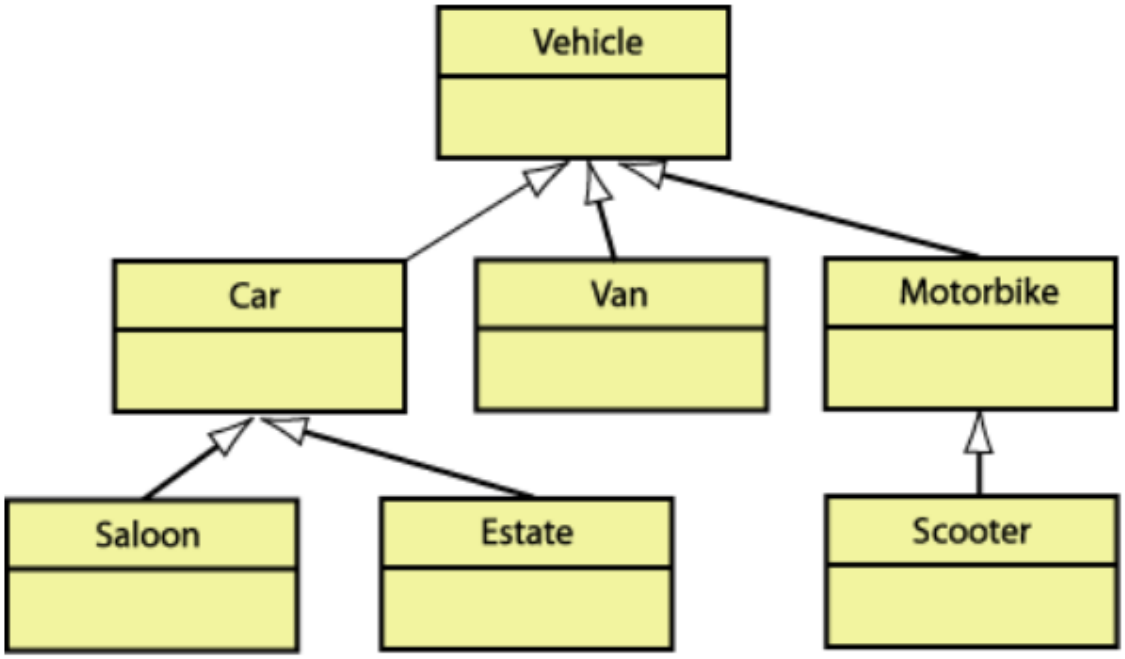

Figure 3 Example 1 of Inheritance

One way to determine that you have organised your classes correctly is to check them using the "IS-A" and "IS-A-PART-OF" relationship checks. It is easy to confuse objects within a class and children of classes when you first begin programming with an OOP methodology. So, to check the previous relationship between Car and Vehicle, we can see this in Figure.

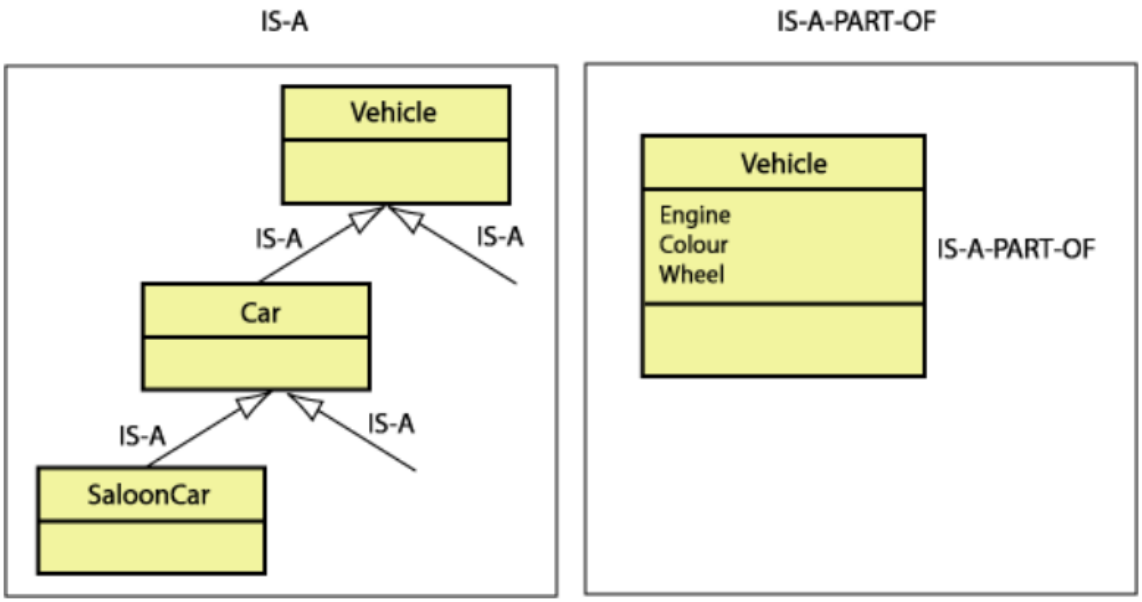

Figure 4 Example 2 of Inheritance

The IS-A relationship describes the inheritance in the figure, where we can say, "A Car IS-A Vehicle" and "A SaloonCar IS-A Car", so all relationships are correct. The IS-A-PART-OF relationship describes the composition (or aggregation) of a class. So in the same figure (Figure 1.9) we can say "An Engine IS-A-PART-OF a Vehicle", or "An Engine, Colour and Wheels IS-A-PART-OF a Vehicle". This is the case even though an Engine is also a class! where there could be many different descriptions of an Engine - petrol, diesel, 1.4, 2.0, 16 valve etc.

So, using inheritance the programmer can:

- Inherit a behaviour and add further specialised behaviour - for example a Car IS A Vehicle with the addition of four Wheel objects, Seats etc.
- Inherit a behaviour and replace it - for example the SaloonCar class will inherit from Car and provide a new "boot" implementation.
- Cut down on the amount of code that needs to be written and debugged - for example in this case only the differences are detailed, a SaloonCar is essentially identical to the Car, with only the differences requiring description.

2.3. Polyporphism

Polymorphism means "multiple forms". In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in different classes, or the same method name can be used in the same class with slightly different paramaters. There are two forms of polymorphism, **over-riding** and **over-loading**.

2.2.1. Over-Riding

As discussed, a derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation. So, over-riding is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.

Over-riding allows different kinds of objects that share a common behaviour to be used in

code that only requires that common behaviour. So, Over-Riding allows:

- A more straightforward API where we can call methods the same name, even thought these methods have slightly different functionality.
- A better level of abstraction, in that the implementation mechanics remain hidden.

2.2.2. Over-Loading

Over-Loading is the second form of polymorphism. The same method name can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler. There are two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality. When we add two int values we expect an intresult - for example 6 + 7 = 13. However, if we passed two String objects we would expect a result of "6" + "7" = "67". In other words the strings should be concatenated.

The number of arguments can also determine which method should be run which will provide different functionality where the first method may simply display the current channel number, but the second method will set the channel number to the number passed.

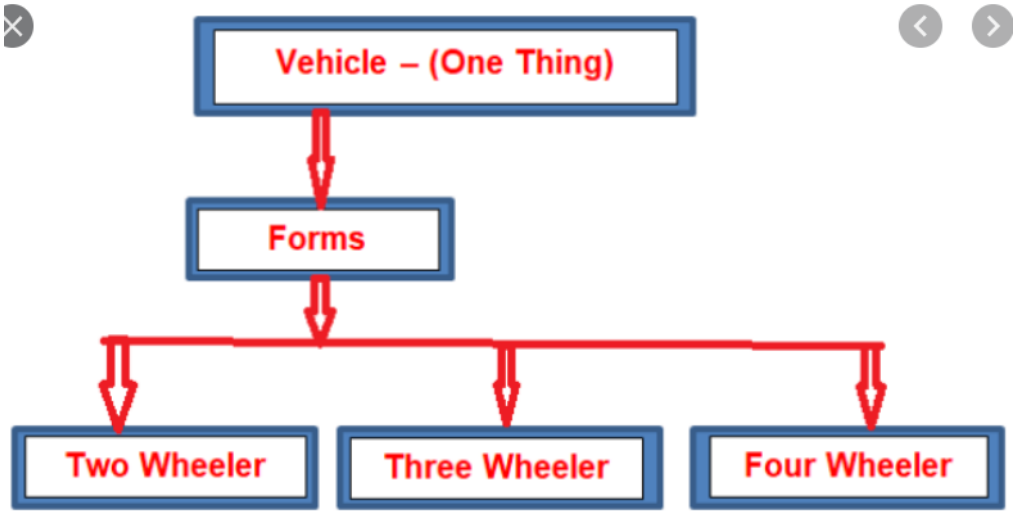

Figure 5 Example of Polyporphism

2.4. Encapsulation

The object-oriented paradigm encourages encapsulation. Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works. All we need to understand is the interface that is provided for us.

There is a sub-set of functionality that the user is allowed to call, termed the interface. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television.

The full implemenation of a class is the sum of the public interface plus the private implementation.

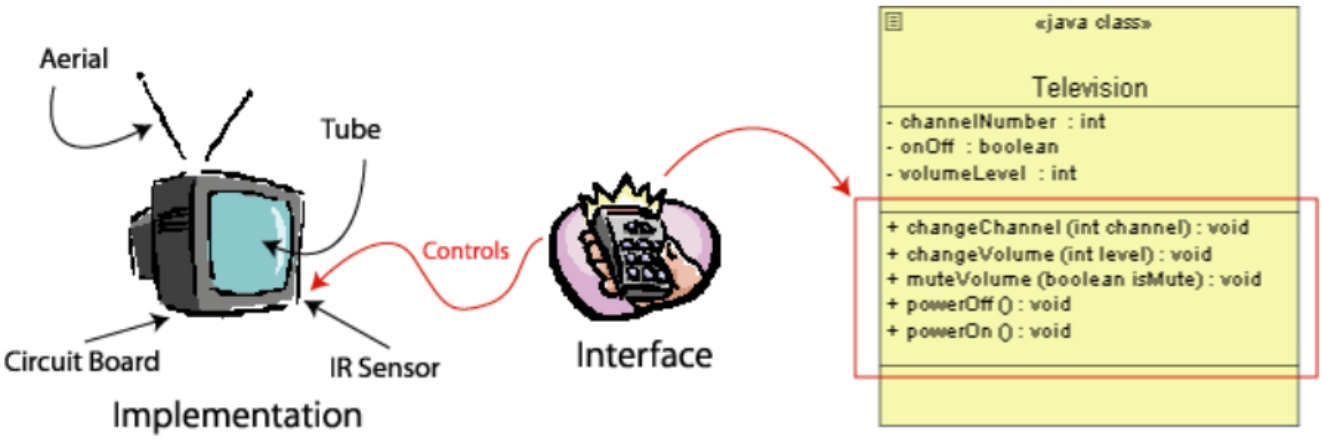

Figure 6 Example of Encapsulation

Encapsulation is the term used to describe the way that the interface is separated from the implementation. You can think of encapsulation as "data-hiding", allowing certain parts of an object to be visible, while other parts remain hidden. This has advantages for both the user and the programmer.

For the user (who could be another programmer):

- The user need only understand the interface.
- The user need not understand how the implementation works or was created.

For the programmer:

- The programmer can change the implementation, but need not notify the user.

So, providing the programmer does not change the interface in any way, the user will be unaware of any changes, except maybe a minor change in the actual functionality of the application.

We can identify a level of 'hiding' of particular methods or states within a class using the public, private and protected keywords:

- public methods - describe the interface.
- private methods - describe the implementation

2.5. Abstraction

An abstract class is a class that is incomplete, in that it describes a set of operations, but is missing the actual implementation of these operations. Abstract classes:

Cannot be instantiated.

So, can only be used through inheritance.

As discussed previously, a class is like a set of plans from which you can create objects. In relation to this analogy, an abstract class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.

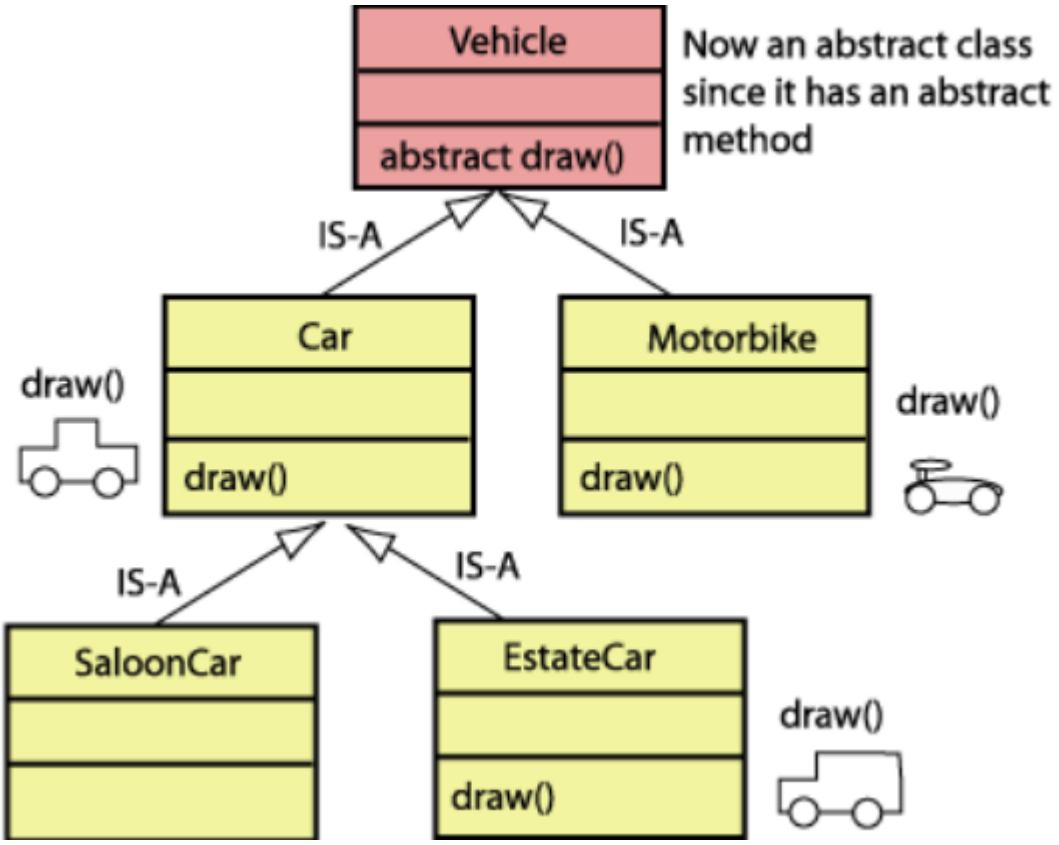

Figure 7 Example 1 of Abstraction

Figure above illustrates this example. The draw() has been written in all of the classes and has some functionality. The draw() in the Vehicle has been tagged as abstract and so this class cannot be instantiated - i.e. we cannot create an object of the Vehicle class, as it is incomplete. In Figure 1.11 the SaloonCar has no draw() method, but it does inherit a draw() method from the parent Car class. Therefore, it is possible to create objects of SaloonCar.

If we required we could also tag the draw() method as abstract in a derived class, for example we could also have tagged the draw() as abstract in the Car class. This would mean that you could not create an object of the Car class and would pass on responsibility for implementing the draw() method to its children - see Figure below.

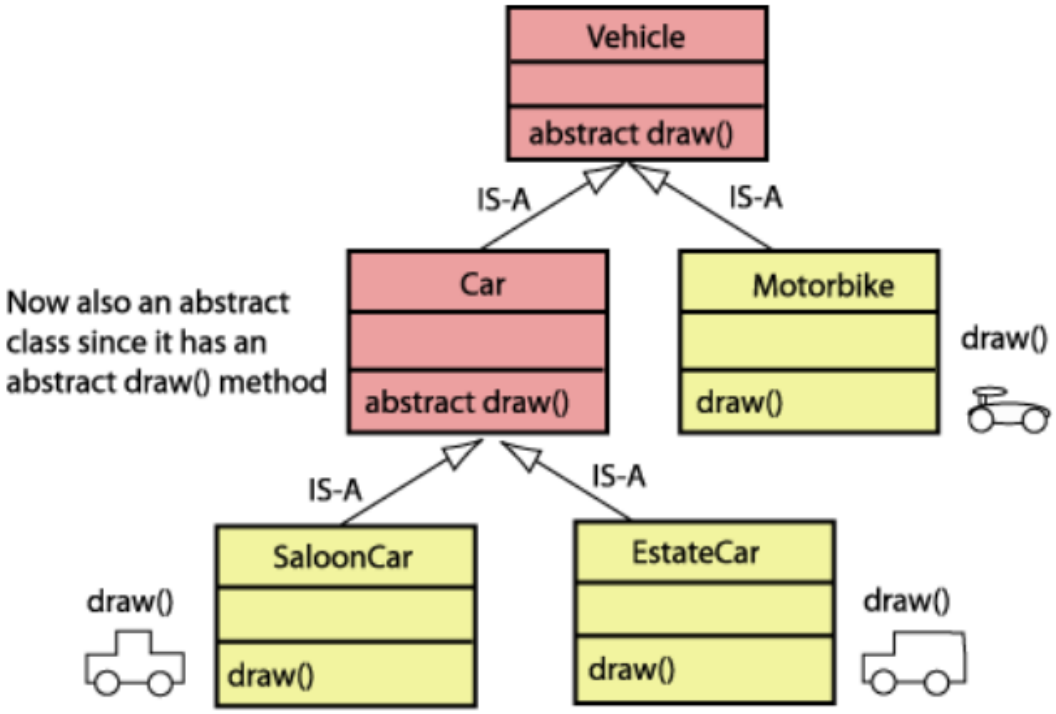

Figure 8 Example 2 of Abstraction

3. Object-Orientated Paradigm class Relationships

3.1. Aggregation

We talk about aggregation between two object when one of them is an owner of the other one.

Looking at the sentence “A manager has many workers under him”, we can see that manager is the owner of a group of employees in his organization, and those employees are not going to report to another entity in the organization. The relationship between the manager and the employees are an aggregation relationship. Let see how this works with a code.

Figure 9 UML of Aggregation

The worker class contains only a method returning the worker name for simplicity.

In this relationship, the Manager class is an owner of employees. That ownership is shown in the manager class by the list of workers. This kind of relationship is an aggregation relationship.

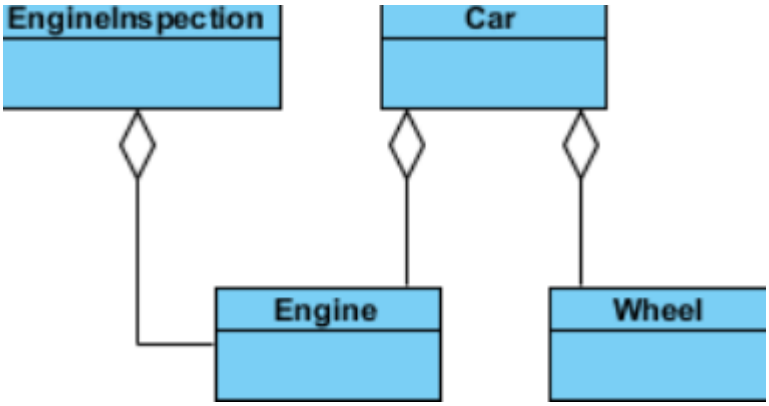

Figure 10 - Aggregation Example

3.2. Composition

We talk about composition between two objects when they are dependent each other during they life time. Let take a look at these sentences:

- The salary of a manager depends on project success.
- A project success depends on a manager.

This kind of relationship is a composition relationship since each one of the two objects depends on the other object. Let see how it works with a code.

Figure 11 UML of Composition

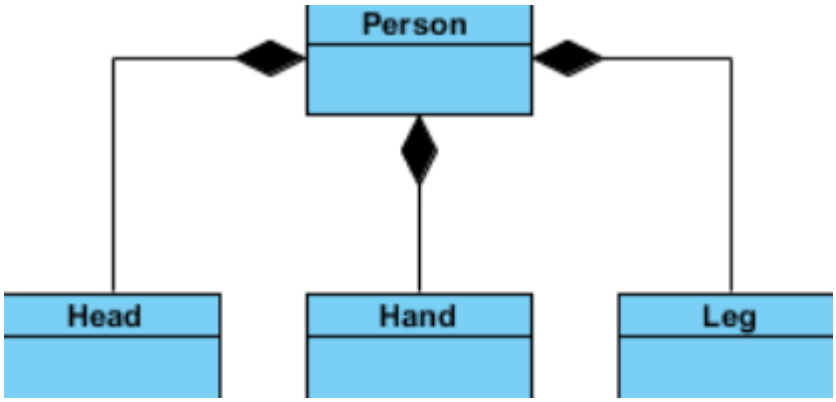

Figure 12 Composition Example

3.3. Association

We talk about association between two objects when each one of them can use the other one, but also each one of them can exist without the other one. There is no dependency between them.

If we take a look at the sentence “A manager has a swipe card to enter the company premises”, we can see the relationship “has a”. That means a manager can exist without his swipe card, and his swipe card can also be assigned to another employee. The manager also can have another swipe cared. The relationship between the manager and the swipe card is an association relationship since they are not dependent on each other. Let see how it works with a code.

Figure 13 UML of Association

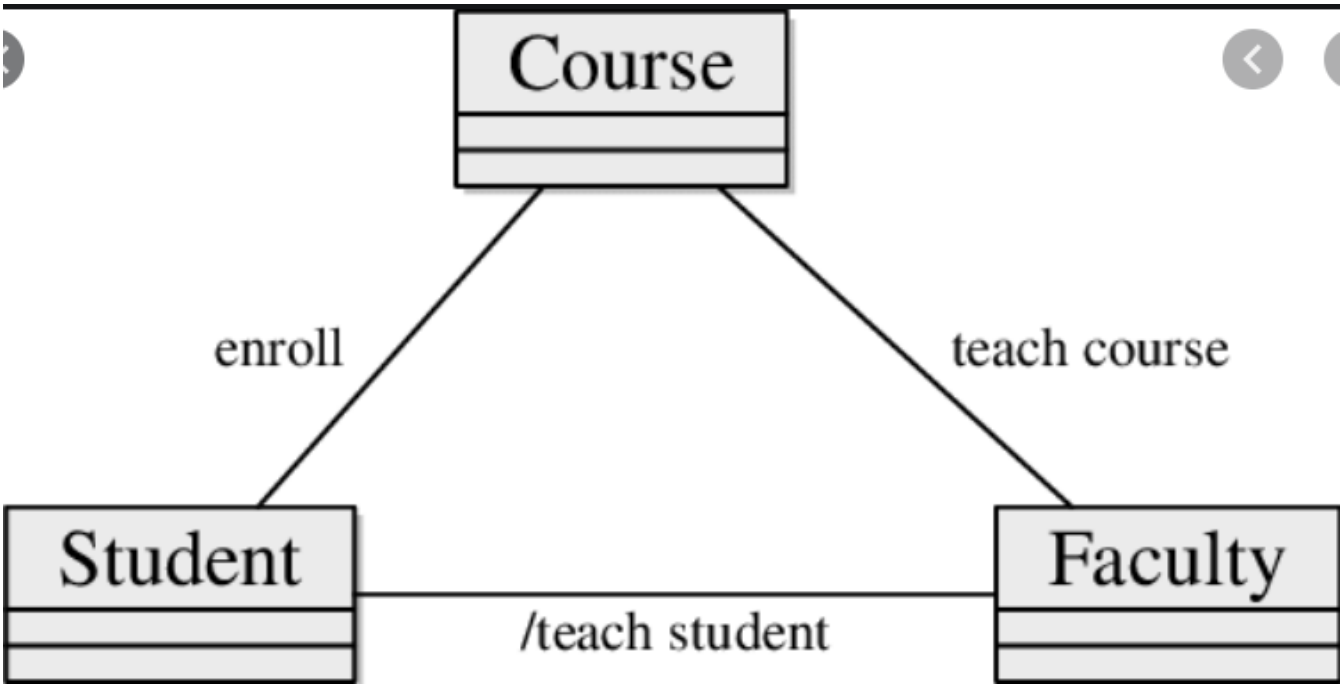

Figure 14 Association Example

3.4. Dependence

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

- Client Class: The client class (dependent class) is a class which depends on the service class
- Service Class: The service class (dependency) is a class that provides service to the client class.
- Injector Class: The injector class injects the service class object into the client class.

Figure 15 UML of Dependency

Types of Dependency Injection

- **Constructor Injection**: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.
- **Property Injection**: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.
- **Method Injection**: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

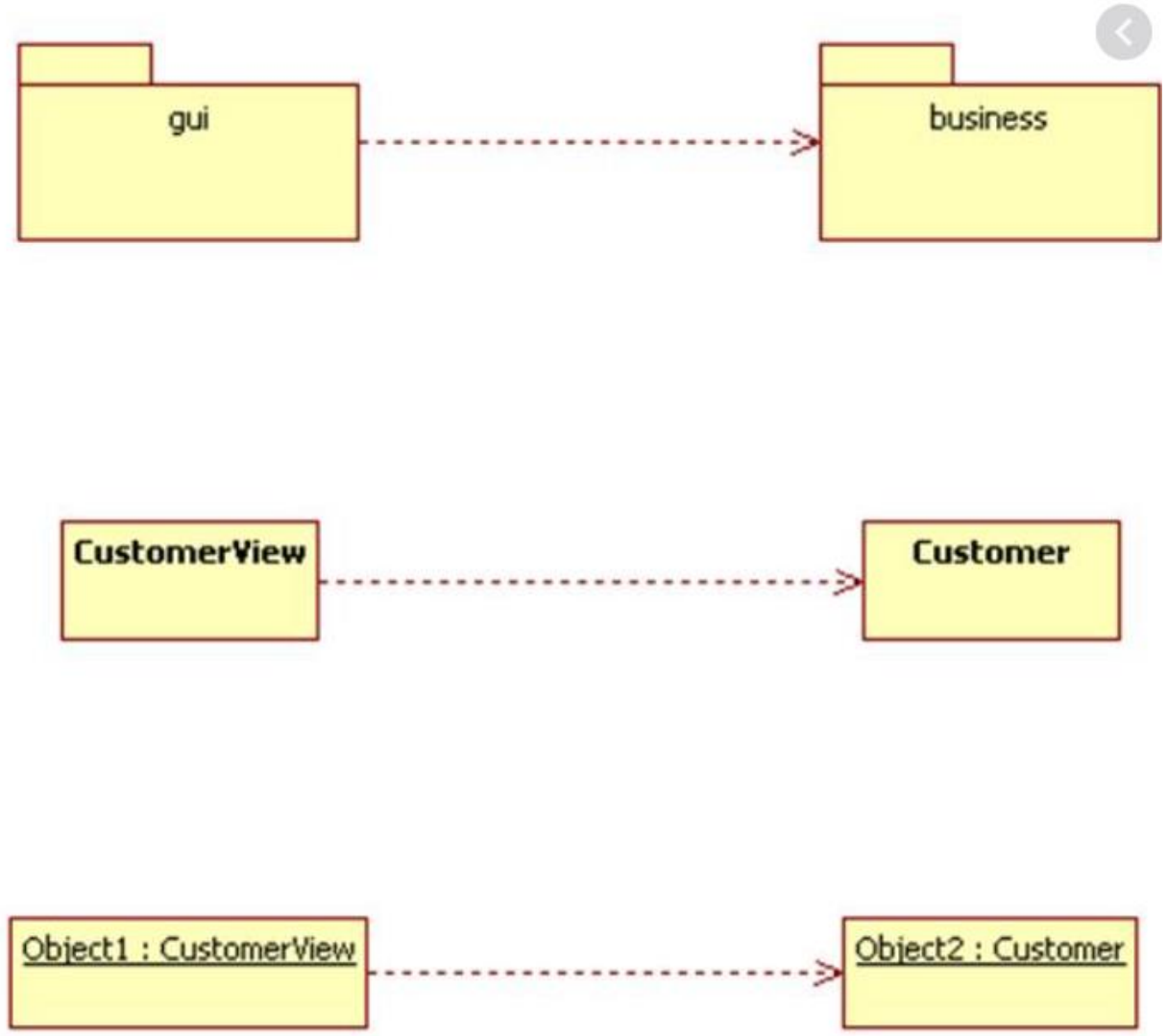

Figure 16 Dependency Example

P2 Design and build class diagrams using a UML tool.

1. Introduction about scenario

A new school named Green has been built this year and it needs an effective system to manage the new students in the school. The system will help the headmaster easily to manage the students and teachers. The users can find their information based on their names, ids, etc. which are input before.

This report will provide a management system to help the school manage the members in school including teachers and students. So this system should have several basic functions like add, delete, adjust, show, or find based on their information. There are some functions that system will have:

- The manager can input the student's information like: name, age, address, ID, gender, etc.
- The user can find out someone's information by using the display functions. By input the id it will show all of the non-confidential information like name, age, class.
- The show functions is the one when using it will show all information of user in the system which are not secret information
- The manager can add the new students every years
- The manager can also delete when they are end of the course

2. Use-case diagrams about scenario

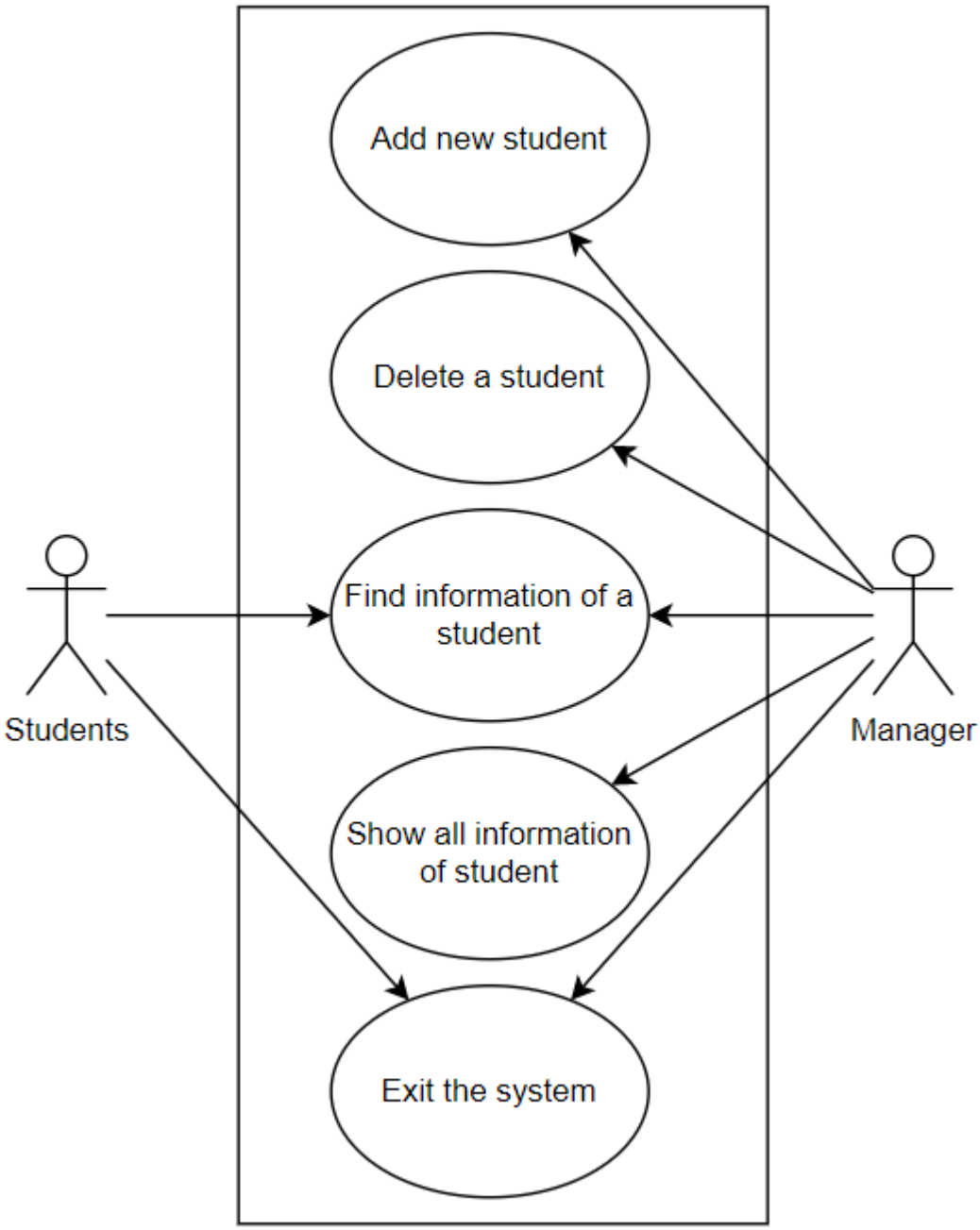

Figure 17 Use-case diagrams about scenario

3. Class diagrams about scenario

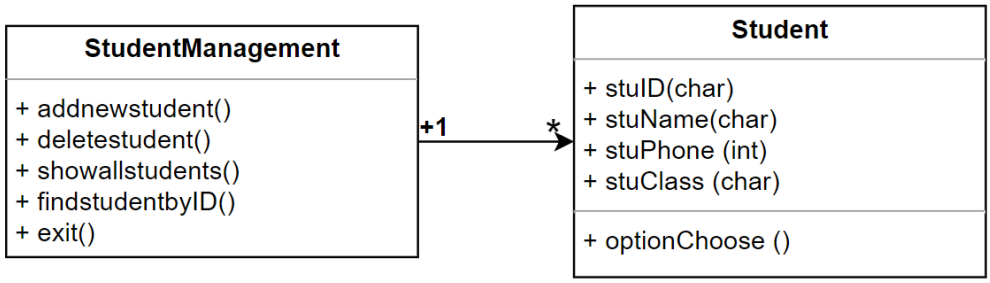

Figure 18 Class diagrams about scenario

4. Pseudo-codes

4.1. Select menu function

```
Begin
    char select;
    Show "Menu:"
    "1. Add new student"
    "2. Delete student"
    "3. Show all students"
    "4. Find student by ID"
    "5. Exit"
    print "Input your select:";
    get select;
    switch (select)
        case 1: Add
            break;
        case 2: Delete
            break;
        case 3: Show
            break;
        case 4: Find
            break;
        case 5: Exit
            break;
        default: print "Invalid select!!";
        display Menu;
End
```

Figure 19 Select menu functions

4.2. Add new student

```
Begin
    class student
        char stuID;
        char stuName;
    ArrayList studentlist = new ArrayList;
    print "Input ID of Student: "
        get stuID;
    print "Input student Name: "
        Get stuName;
    print "Input student Phone: "
        Get stuPhone;
    print "Input student Class: "
        Get stuClass;
Save to student;
        Add student to studentlist;
End
```

Figure 20 Add new student Function

4.3. Delete a student

```
Begin
    print "Enter the ID of the student who will be removed:";
    get ID;
        If(ID = stu.ID)
            { Delete student from studentlist;
            print "Student ID n have been deleted!";}
End
```

Figure 21 Delete a student Function

4.4. Show all student's information

```
Begin
     show all student in studentlist;
     print "      Information about each student:"
     print "Name of Student: "
     print "Student ID: "
     print "Student Phone: "
     print "Student ID: "
     print "-----------------------------------"
     print "Name of Student: "
     print "Student ID: "
     print "Student Phone: "
     print "Student ID: "
     etc.
End
```

Figure 22 Show all students Functions

4.5. Find a student by ID

```
Begin
    print "Enter the student ID for whom you want to see information:";
    get ID;
    If (ID = stu.ID)
        show student from studentlist;
        print "     Information about student:"
        print "Name of Student: "
        print "Student ID: "
        print "Student Phone: "
        print "Student ID: "
End
```

Figure 23 Find a student Function

5. Acitivity diagrams

5.1. Add new student

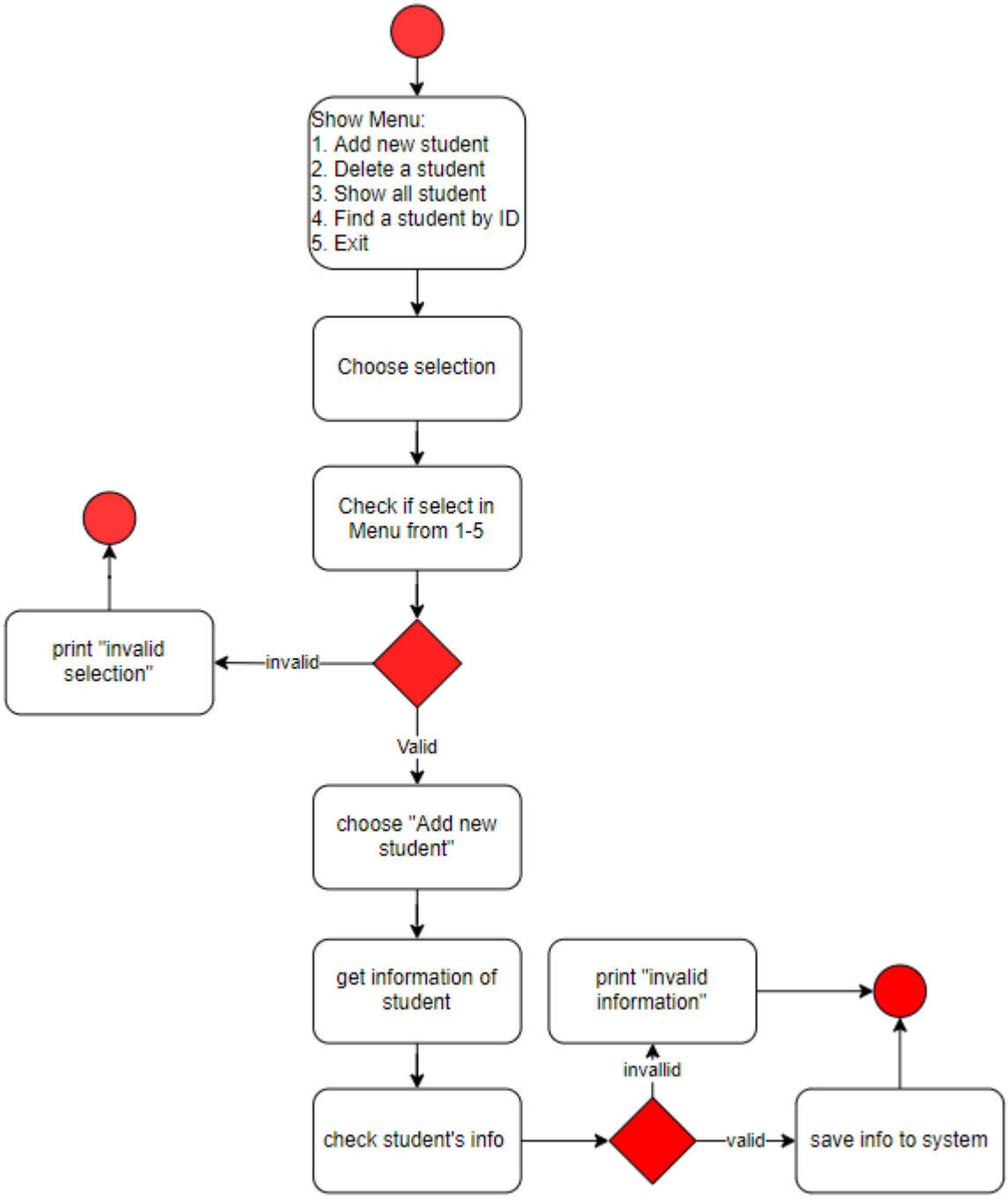

Figure 24 Acitivity Diagrams - Add new Student

5.2. Delete a student

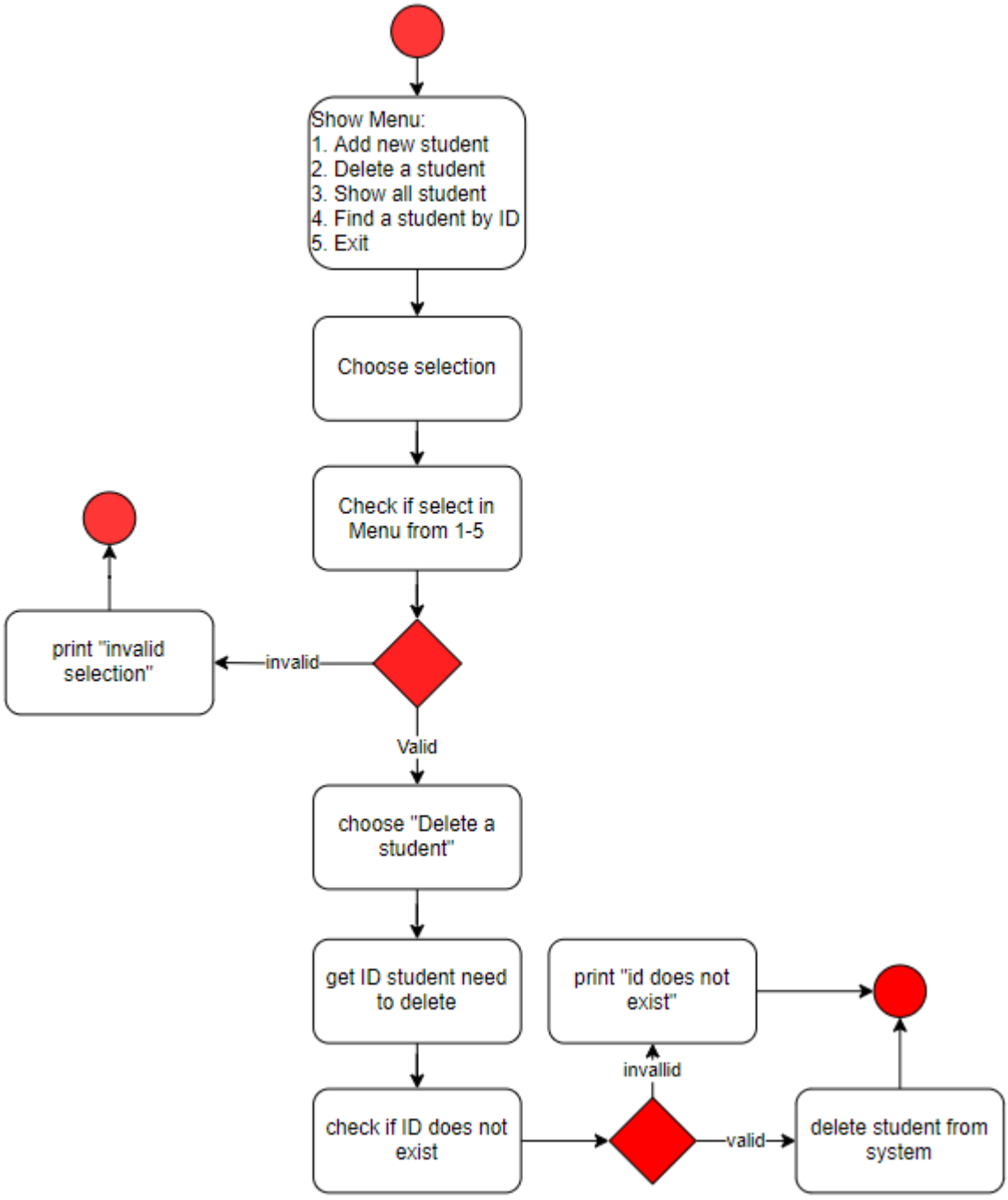

Figure 25 Acitivity Diagrams - Delete a student

5.3. Show all students

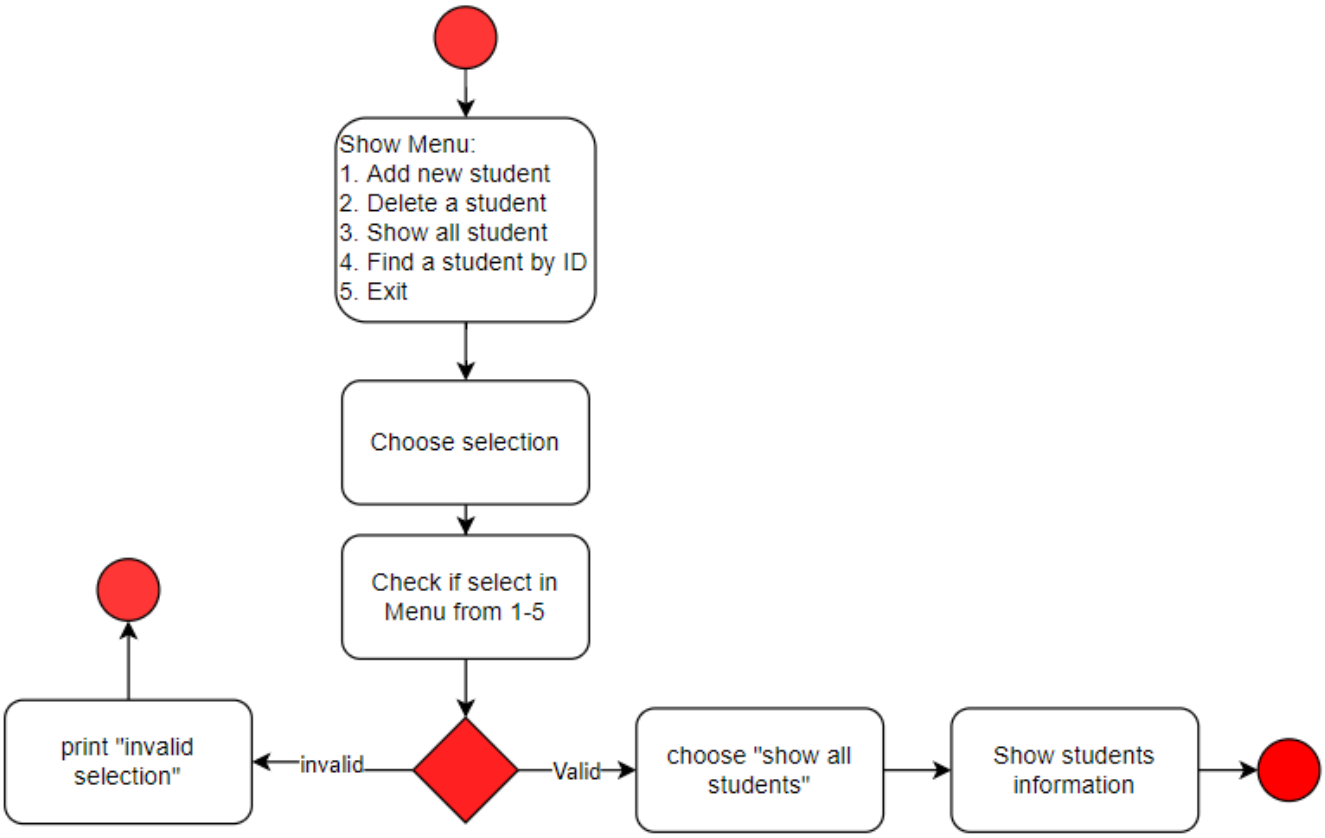

Figure 26 Acitivity Diagrams - Show all students

5.4. Find a student by ID

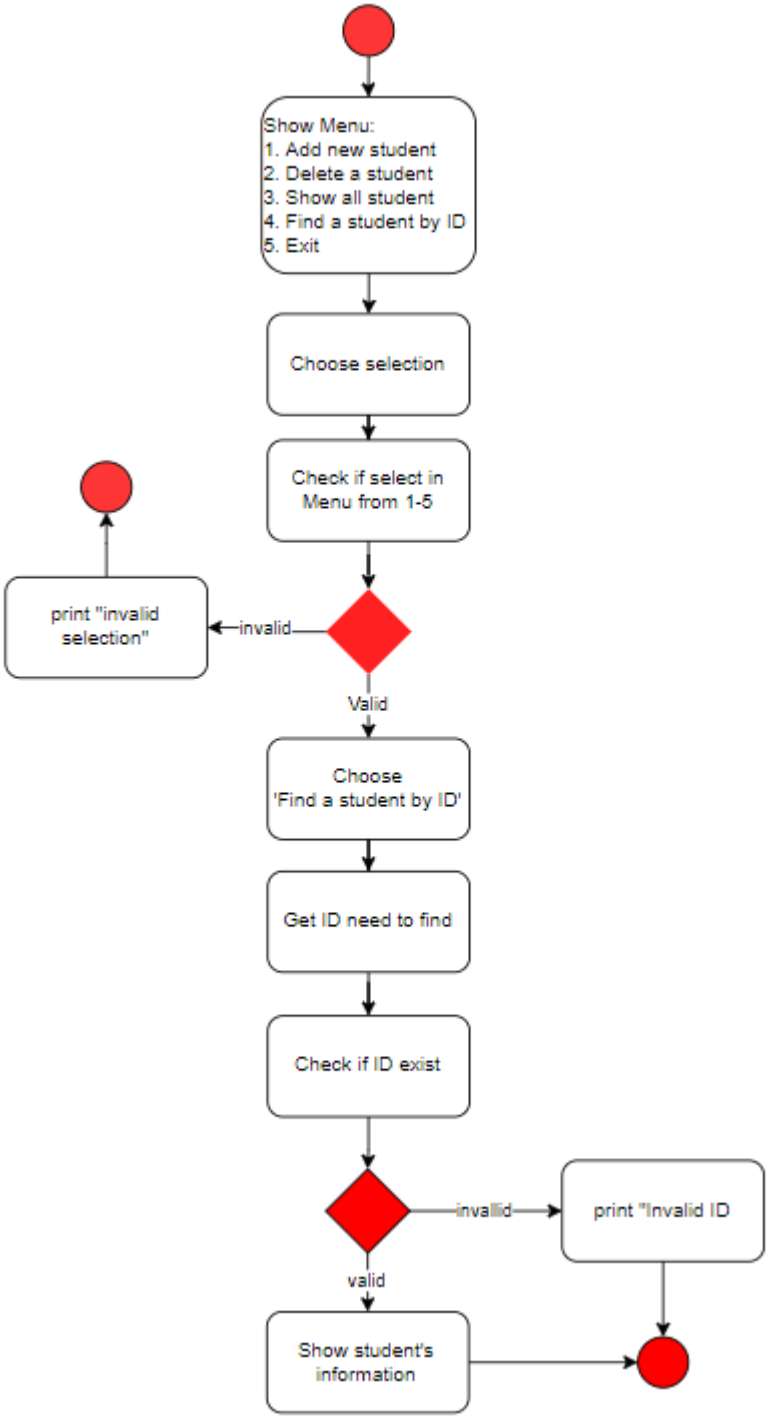

Figure 27 Acitivity Diagrams - Find a student

5.5. Exit the system

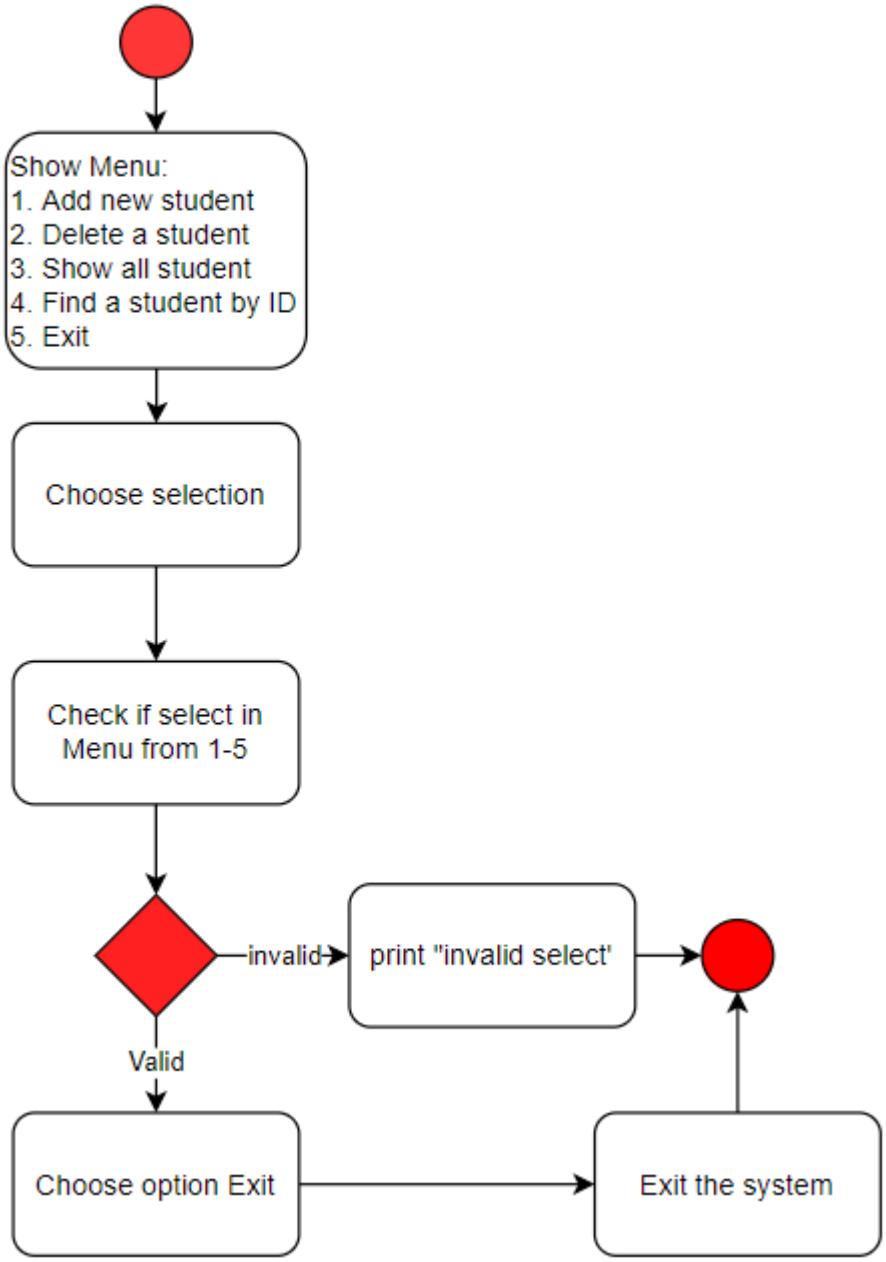

Figure 28 Acitivity Diagrams - Exit the system

REFERENCES

Vidya, V, A (2021) Object Oriented Programming Using C# .NET [online] Available at: https://www.c-sharpcorner.com/UploadFile/84c85b/object-oriented-programming-using-C-Sharp-net/ [Accessed at: 08 June.2021]

Tutorialsteacher () Dependency Injection [online] Available at: https://www.tutorialsteacher.com/ioc/dependency-injection [Accessed at: 08 June.2021]

Tutorialspoint () OOAD - Object Oriented Paradigm [online] Available at: https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.htm [Accessed at: 08 June.2021]

Tutorialspoint () Object-Oriented Paradigm [online] Available at: https://www.tutorialspoint.com/software_architecture_design/object_oriented_paradigm.htm [Accessed at: 08 June.2021]

Samual, S (2018) Association, Composition and Aggregation in C# [online] Available at: https://www.tutorialspoint.com/Association-Composition-and-Aggregation-in-Chash [Accessed at: 08 June.2021]

Royce, W. W. (1970) "Managing the development of large software systems: concepts and techniques", Proceedings of IEEE WESCON, August 1970. [online] Available at: https://sites.google.com/site/derekmolloyee402/introduction/chapter-1---introduction-to-object-oriented-programming [Accessed at: 08 June.2021]

Koderhq () C# Composition Tutorial [online] Available at: https://www.koderhq.com/tutorial/csharp/oop-composition/ [Accessed at: 08 June.2021]

Jignest, T (2019) Understanding Polymorphism In C# [online] Available at: https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/ [Accessed at: 08 June.2021]

Jignest, T (2019) Association, Aggregation and Composition [online] Available at: https://www.c-sharpcorner.com/UploadFile/ff2f08/association-aggregation-and-composition/ [Accessed at: 08

June.2021]

Ibrahima, B (2017) Association, Aggregation, and Composition in C# [online] Available at: https://medium.com/@ibrahimyengue/association-aggregation-and-composition-in-c-8cbeaa81201d [Accessed at: 08 June.2021]

Geeksforgeeks (2021) C# | Abstract Classes [online] Available at: https://www.geeksforgeeks.org/c-sharp-abstract-classes/ [Accessed at: 08 June.2021]

Erin, D (2020) What is Object Oriented Programming? OOP Explained in Depth [online] Available at: https://www.educative.io/blog/object-oriented-programming [Accessed at: 08 June.2021]

Dotnettutorials () Dependency Injection in C# [online] Available at: https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/ [Accessed at: 08 June.2021]

Boehm, B. W. (1981) Software Engineering Economics, Ch. 4 Prentice Hall, Upper Saddle River, NJ [online] Available at: https://sites.google.com/site/derekmolloyee402/introduction/chapter-1---introduction-to-object-oriented-programming [Accessed at: 08 June.2021]