



Higher Nationals in Computing

# Unit 20: Advanced Programming Assignment 2

Learner’s name:

ID:

Class:

Subject code: 1651

Assessor name:

Assignment due: Assignment submitted:

**ASSIGNMENT 2 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | **Unit 20: Advanced Programming** | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** |  | **Student ID** |  |
| **Class** |  | **Assessor name** |  |
| **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** |  |

**Grading grid**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P3 | P4 | M3 | M4 | D3 | D4 |
|  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Signature & Date:** | | |

**ASSIGNMENT 2 BRIEF**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 2: Advanced Programming | | |
| **Assignment title** | Application development with class diagram and design patterns | | |
| **Academic Year** |  | | |
| **Unit Tutor** |  | | |
| **Issue date** |  | **Submission date** |  |

|  |
| --- |
| **Submission Format:** |
| *Format:* The submission is in the form of an individual 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/> together with zipped project files.  *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**: (continued from Assignment 1) Your team has shown the efficient of UML diagrams in OOAD and introduction of some Design Patterns in usages. The next tasks are giving a demonstration of using OOAD and DP in a small problem, as well as advanced discussion of range of design patterns.  **Tasks:**  Your team is now separated and perform similar tasks in parallel. You will choose one of the real scenarios that your team introduced about DP in previous phase, then implement that scenario based on the corresponding class diagram your team created. You may need to amend the diagram if it is needed for your implementation. In additional, you should discuss a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice.  In the end, you need to write a report with the following content:   * A final version of the class diagram based on chosen scenario which has potential of using DP. * Result of a small program implemented based on the class diagram, explain how you translate from design diagram to code. * Discussion of a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice (why your DP is the most appropriate in that case). |
|  |

|  |  |  |
| --- | --- | --- |
| Learning Outcomes and Assessment Criteria | | |
| Pass | Merit | Distinction |
| **LO3** Implement code applying design patterns | | |
| **P3** Build an application derived from UML class diagrams. | **M3** Develop code that implements a design pattern for a given purpose. | **D3** Evaluate the use of design patterns for the given purpose specified in M3. |
| **LO4** Investigate scenarios with respect to design patterns | | |
| **P4** Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types. | **M4** Reconcile the most appropriate design pattern from a range with a series of given scenarios. | **D4** Critically evaluate a range of design patterns against the range of given scenarios with justification of your choices. |

Table of Contents

[Unit 20: Advanced Programming Assignment 2 1](#_Toc97151524)

[P3 Build an application derived from UML class diagrams. 1](#_Toc97151525)

[**1.** **Class Diagram** 1](#_Toc97151526)

[**2.** **Source code** 1](#_Toc97151527)

[**2.1.** **Program.cs** 1](#_Toc97151528)

[**2.2.** **Student.cs** 2](#_Toc97151529)

[**2.3.** **StudentManage.cs** 4](#_Toc97151530)

[**2.4.** **Solution Explorer in Visual Studio** 6](#_Toc97151531)

[**3.** **Application screenshots** 6](#_Toc97151532)

[**3.1.** **Menu** 6](#_Toc97151533)

[**3.2.** **Add** 7](#_Toc97151534)

[**3.3.** **Delete** 8](#_Toc97151535)

[**3.4.** **Show all student information** 8](#_Toc97151536)

[**3.5.** **Find a student by ID** 9](#_Toc97151537)

[**3.6.** **Exit** 9](#_Toc97151538)

[P4 Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types. 10](#_Toc97151539)

[**1.** **Design Pattern** 10](#_Toc97151540)

[**1.1.** **Definition** 10](#_Toc97151541)

[**1.2.** **Purpose of Design Patterns** 10](#_Toc97151542)

[**1.3.** **Types of Design Patterns** 10](#_Toc97151543)

[**2.** **Creation Design Patterns** 11](#_Toc97151544)

[**2.1.** **Singleton Design Pattern** 11](#_Toc97151545)

[**2.2.** **A Factory Pattern** 12](#_Toc97151546)

[**2.3.** **Abstract Factory** 14](#_Toc97151547)

[**3.** **Structural Design Patterns** 15](#_Toc97151548)

[**3.1.** **Adapter Pattern** 15](#_Toc97151549)

[**3.2.** **Bridge Pattern** 16](#_Toc97151550)

[**3.3.** **A Composite Pattern** 18](#_Toc97151551)

[**3.4.** **A Decorator Pattern** 19](#_Toc97151552)

[**4.** **Behavioral Design Patterns** 20](#_Toc97151553)

[**4.1.** **Chain of responsibility** 20](#_Toc97151554)

[**4.2.** **Command Pattern** 22](#_Toc97151555)

[**4.3.** **Interpreter Pattern** 23](#_Toc97151556)

[**4.4.** **Iterator Pattern** 24](#_Toc97151557)

[REFERENCES 26](#_Toc97151558)

Table of Figures

[Figure 1 Folder Structure 6](#_Toc97151487)

[Figure 2 Main menu options 7](#_Toc97151488)

[Figure 3 Function addition 7](#_Toc97151489)

[Figure 4 Notification of add success 8](#_Toc97151490)

[Figure 5 Function delete 8](#_Toc97151491)

[Figure 6 Function read data 8](#_Toc97151492)

[Figure 7 Function finding 9](#_Toc97151493)

[Figure 8 Exit option 9](#_Toc97151494)

[Figure 9 Singleton Pattern 12](#_Toc97151495)

[Figure 10 Factory Pattern 13](#_Toc97151496)

[Figure 11 Abstract Factory Pattern 14](#_Toc97151497)

[Figure 12 Adapter Pattern 16](#_Toc97151498)

[Figure 13 Bridge Pattern 17](#_Toc97151499)

[Figure 14 Composite Pattern 18](#_Toc97151500)

[Figure 15 Decorator Pattern 20](#_Toc97151501)

[Figure 16 Chain of Responsibility Pattern 21](#_Toc97151502)

[Figure 17 Command Pattern 22](#_Toc97151503)

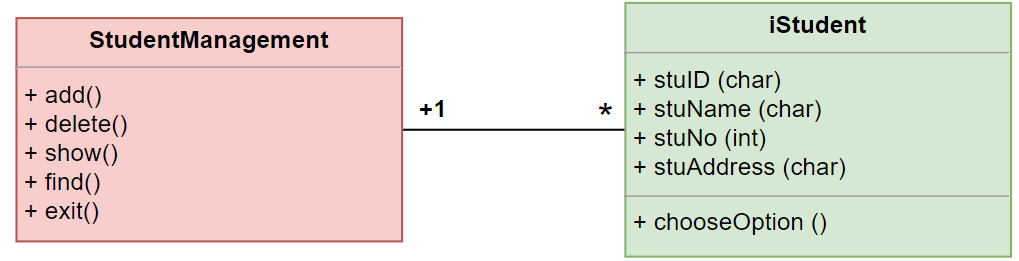
[Figure 18 Interpreter Pattern 24](#_Toc97151504)

[Figure 19 Iterator Pattern 25](#_Toc97151505)

**ASSIGNMENT 2 ANSWERS**

# P3 Build an application derived from UML class diagrams.

1. **Class Diagram**



1. **Source code**
   1. **Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_1651

{

    class Program

    {

        static void Main(string[] args)

        {

            StudentManage manage = new StudentManage();

            int choose = 0;

            do

            {

                choose = manage.Menu();

                switch(choose)

                {

                    case 1:

                        Console.Clear();

                        manage.AddStudent();

                        break;

                    case 2:

                        Console.Clear();

                        Console.Write("Input student id which will be deleted: ");

                        string id = Console.ReadLine();

                        manage.DeleteStudent(id);

                        break;

                    case 3:

                        Console.Clear();

                        manage.DisplayAll();

                        break;

                    case 4:

                        Console.Clear();

                        Console.Write("Input student id to search: ");

                        id = Console.ReadLine();

                        manage.SearchById(id);

                        break;

                    case 5:

                        Console.WriteLine("Pree any key to exit !!!");

                        break;

                    default:

                        Console.Clear();

                        Console.WriteLine("Invalid choose!");

                        break;

                }

            } while (choose != 5);

            Console.ReadKey();

        }

    }

}

* 1. **Student.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_1651

{

    class Student

    {

        public string stuID;

        public string stuName;

        public int stuNo;

        public string stuClass;

        public Student()

        {

            stuID = "";

            stuName = "";

            stuNo = 0;

            stuClass ="";

        }

        public Student(string ID,string Name, int No, string Class)

        {

            this.stuID = ID;

            this.stuName = Name;

            this.stuNo = No;

            this.stuClass = Class;

        }

        public void Display()

        {

            Console.WriteLine("ID: " + stuID);

            Console.WriteLine("Name: " + stuName);

            Console.WriteLine("Phone: " + stuNo);

            Console.WriteLine("Class: " + stuClass);

            Console.WriteLine("---------------------");

        }

        public string stuid

        {

            get { return stuID; }

            set { stuID = value; }

        }

        public string stuname

        {

            get { return stuName; }

            set { stuName = value; }

        }

        public int stuno

        {

            get { return stuNo; }

            set { stuNo = value; }

        }

        public string stuclass

        {

            get { return stuClass; }

            set { stuClass = value; }

        }

    }

}

* 1. **StudentManage.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_1651

{

    class StudentManage

    {

        private List<Student> list;

        public StudentManage()

        {

            list = new List<Student>();

        }

        public void AddStudent()

        {

            Student s = new Student();

            bool check = false;

            do

            {

                    Console.Write("Input ID of student: ");

                    s.stuid = Console.ReadLine();

                    Console.Write("Input Name of student: ");

                    s.stuname = Console.ReadLine();

                    Console.Write("Input Phone number of student: ");

                    s.stuno = Convert.ToInt32(Console.ReadLine());

                    Console.Write("Input Class of student: ");

                    s.stuclass = Console.ReadLine();

                    if (s.stuid == null || s.stuid == String.Empty

                            ||s.stuname == null || s.stuname == String.Empty

                            ||s.stuclass == null || s.stuclass == String.Empty)

                        {

                        Console.WriteLine("Please input all information of student!!");

                        Console.WriteLine("-------------------------------------------------");

                        }

                    else

                        {

                        Console.WriteLine("-----------The student have been added!----------\n");

                        list.Add(s);

                        check = true;

                        }

            } while (!check);

        }

        public int Menu()

        {

            int select = 0;

            Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

            Console.WriteLine("\nOption 1: Add student \nOption 2: Delete student\nOption 3: Show all\nOption 4: Search student by ID\nOption 5: Exit");

            Console.WriteLine("\nPlease input your choose: ");

            Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

            select = Convert.ToInt32(Console.ReadLine());

            return select;

        }

        public void DisplayAll()

        {

            Console.WriteLine("All student information");

            Console.WriteLine("-------------------------------");

            foreach(var s in list)

            {

                s.Display();

            }

        }

        public void SearchById(string id)

        {

            bool found = false;

            foreach(var s in list)

            {

                if(s.stuID.Equals(id))

                {

                    s.Display();

                    found = true;

                    break;

                }

            }

            if(!found)

            {

                Console.WriteLine("Not found!");

            }

        }

        public void DeleteStudent(string id)

        {

            bool found = false;

            foreach (var s in list)

            {

                if (s.stuID.Equals(id))

                {

                    list.Remove(s);

                    found = true;

                    Console.WriteLine("---------------------");

                    Console.WriteLine("The student ID "+id+" have been deleted!");

                    break;

                }

            }

            if (!found)

            {

                Console.WriteLine("Not found!");

            }

        }

    }

}

* 1. **Solution Explorer in Visual Studio**

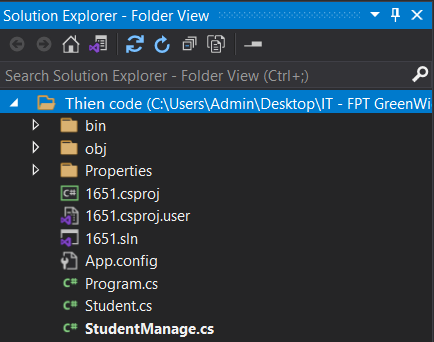


Figure Folder Structure

1. **Application screenshots**
   1. **Menu**

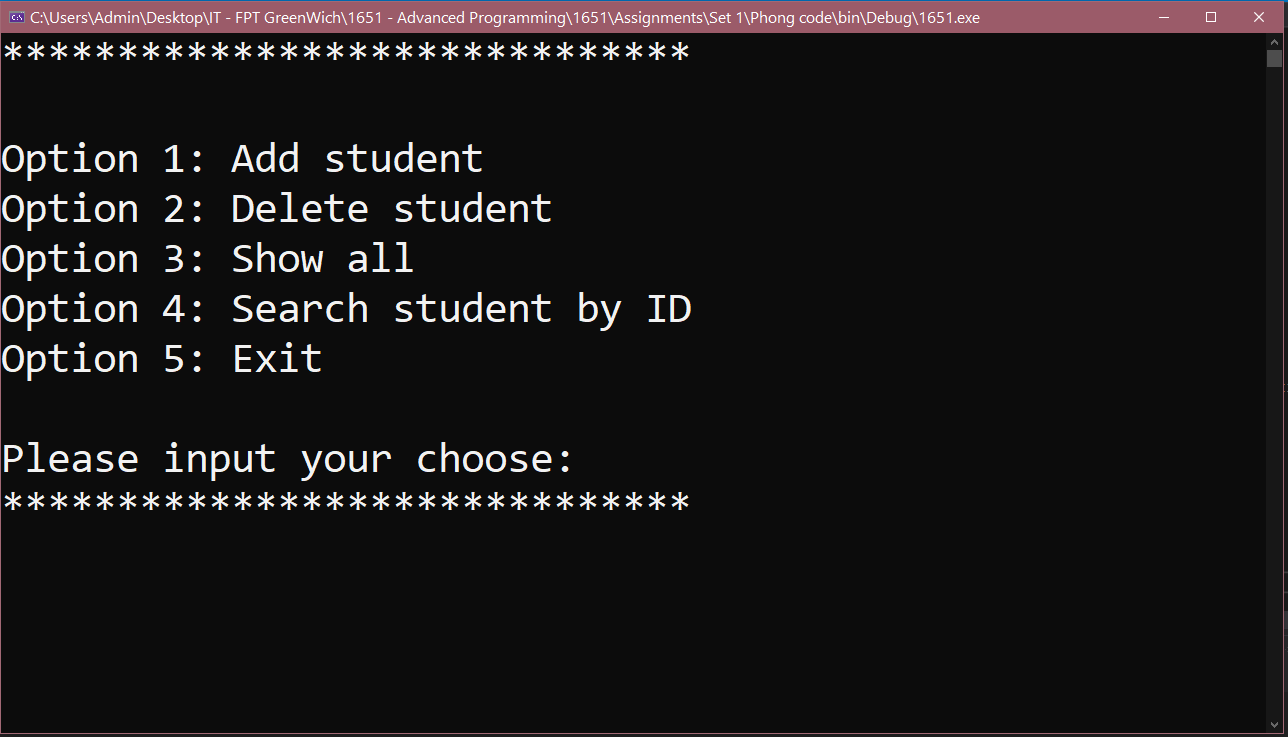


Figure Main menu options

* 1. **Add**

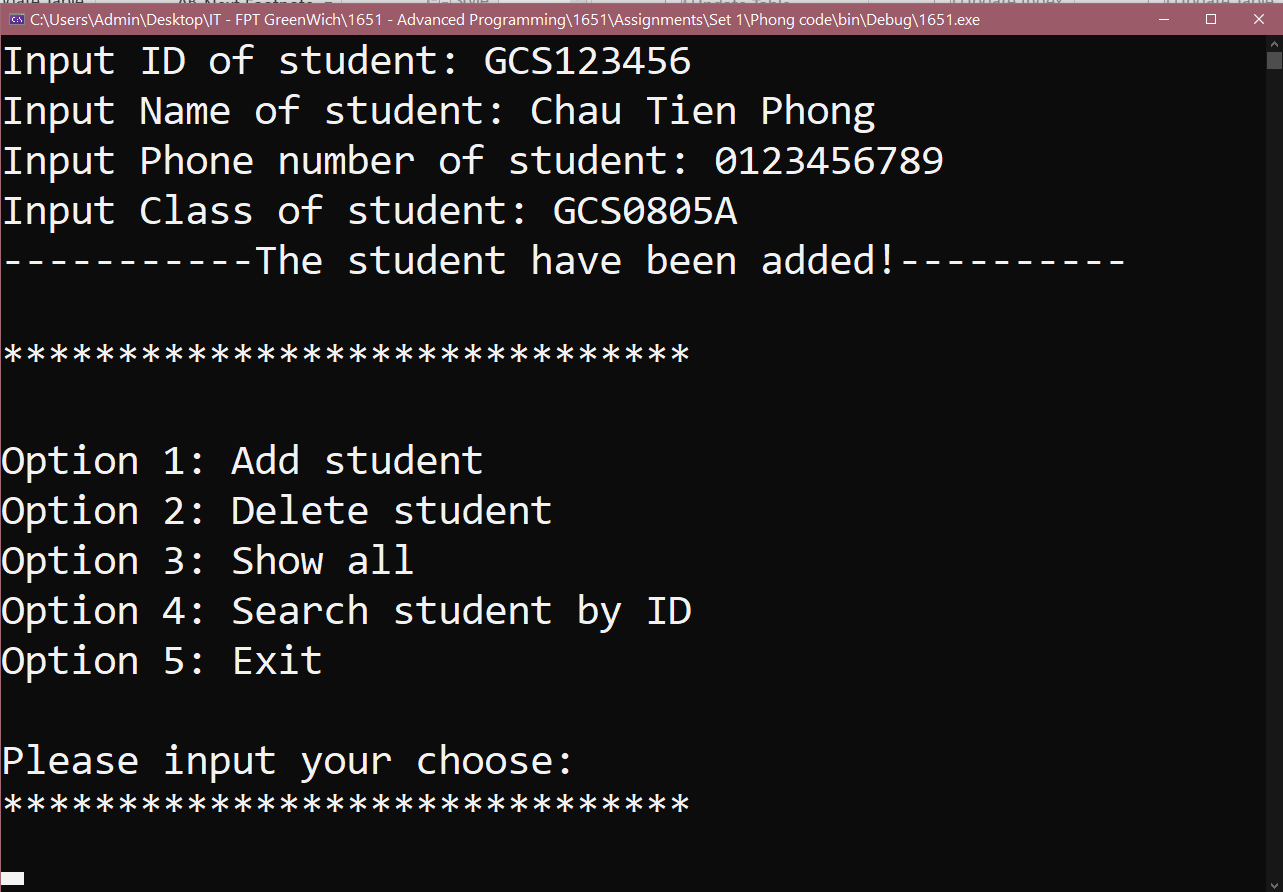


Figure Function addition

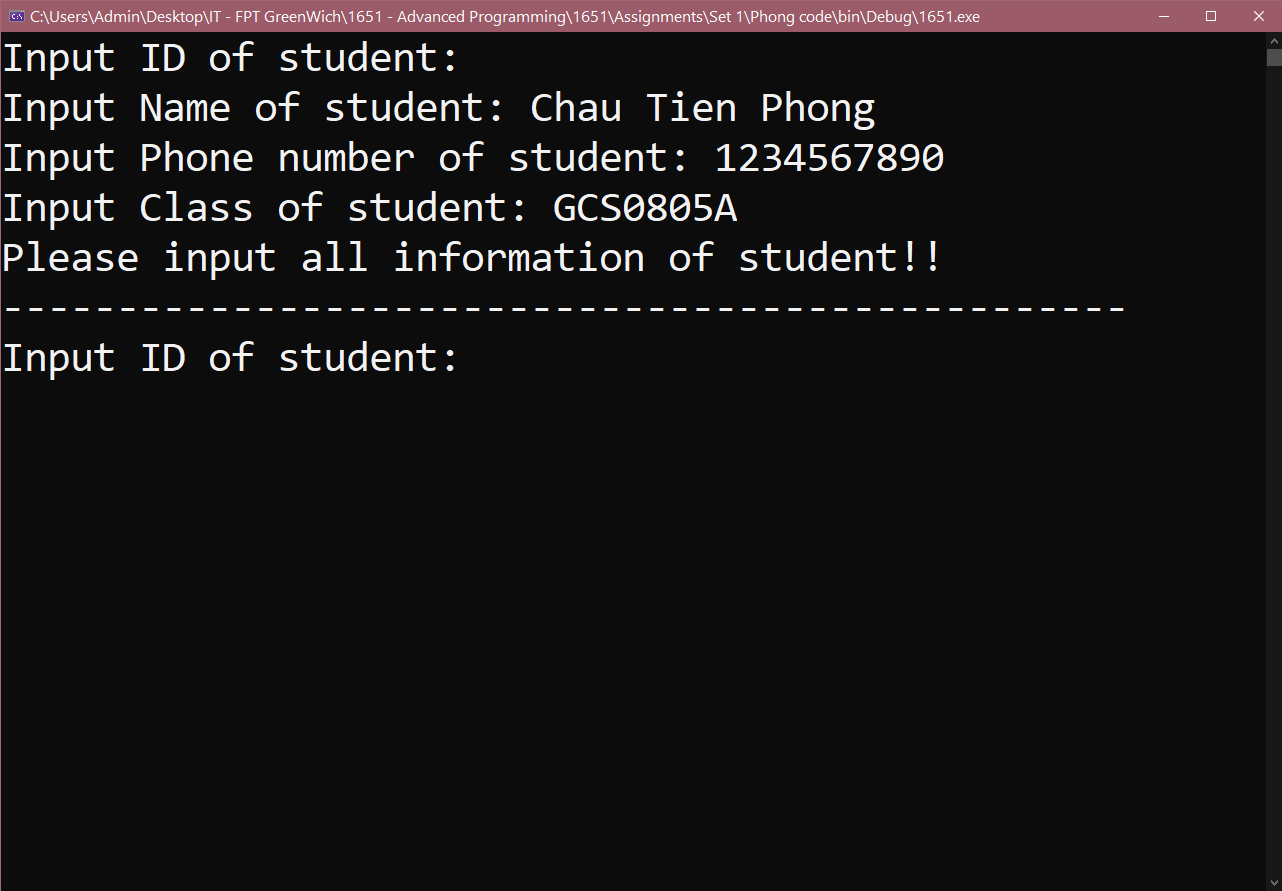


Figure Notification of add success

* 1. **Delete**

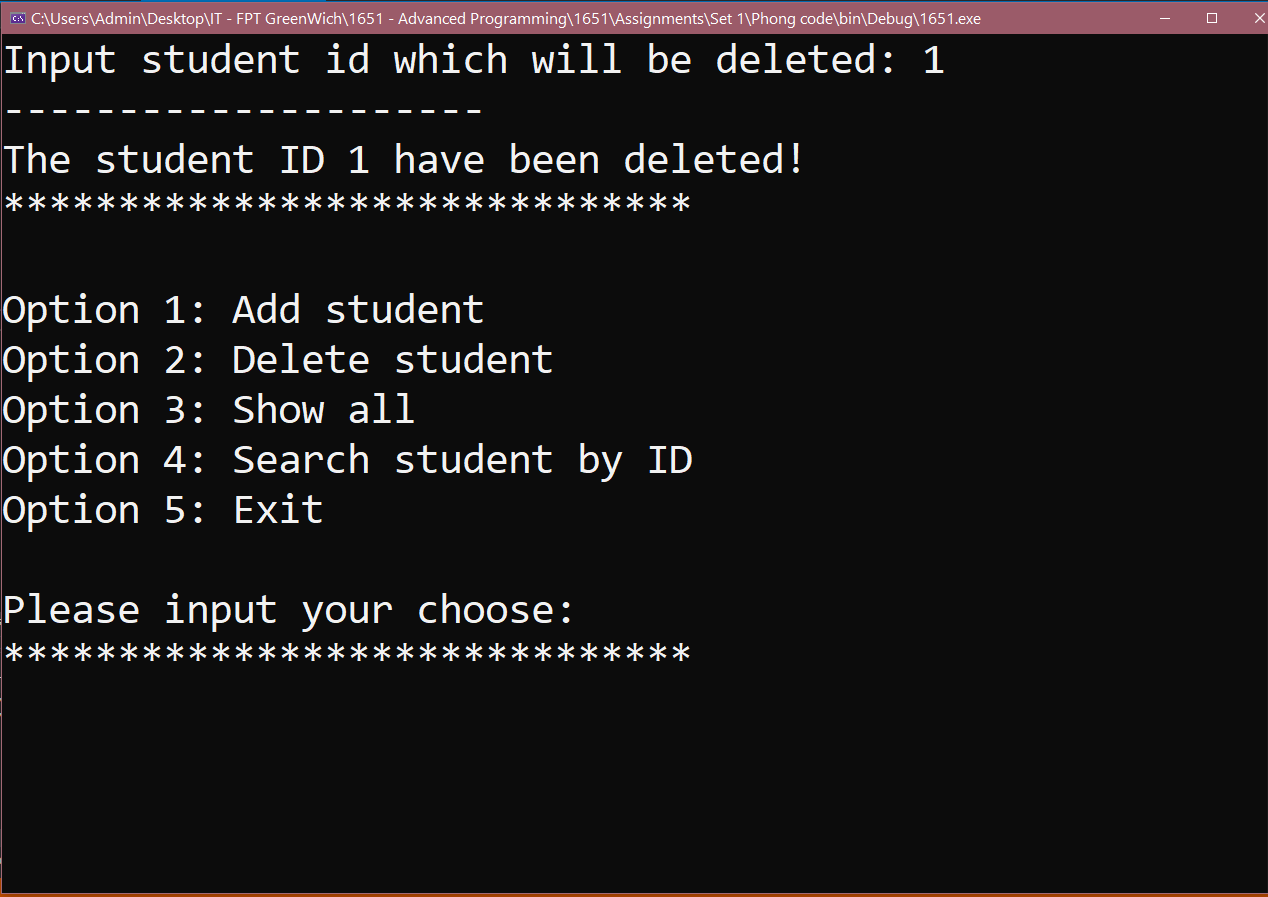


Figure Function delete

* 1. **Show all student information**

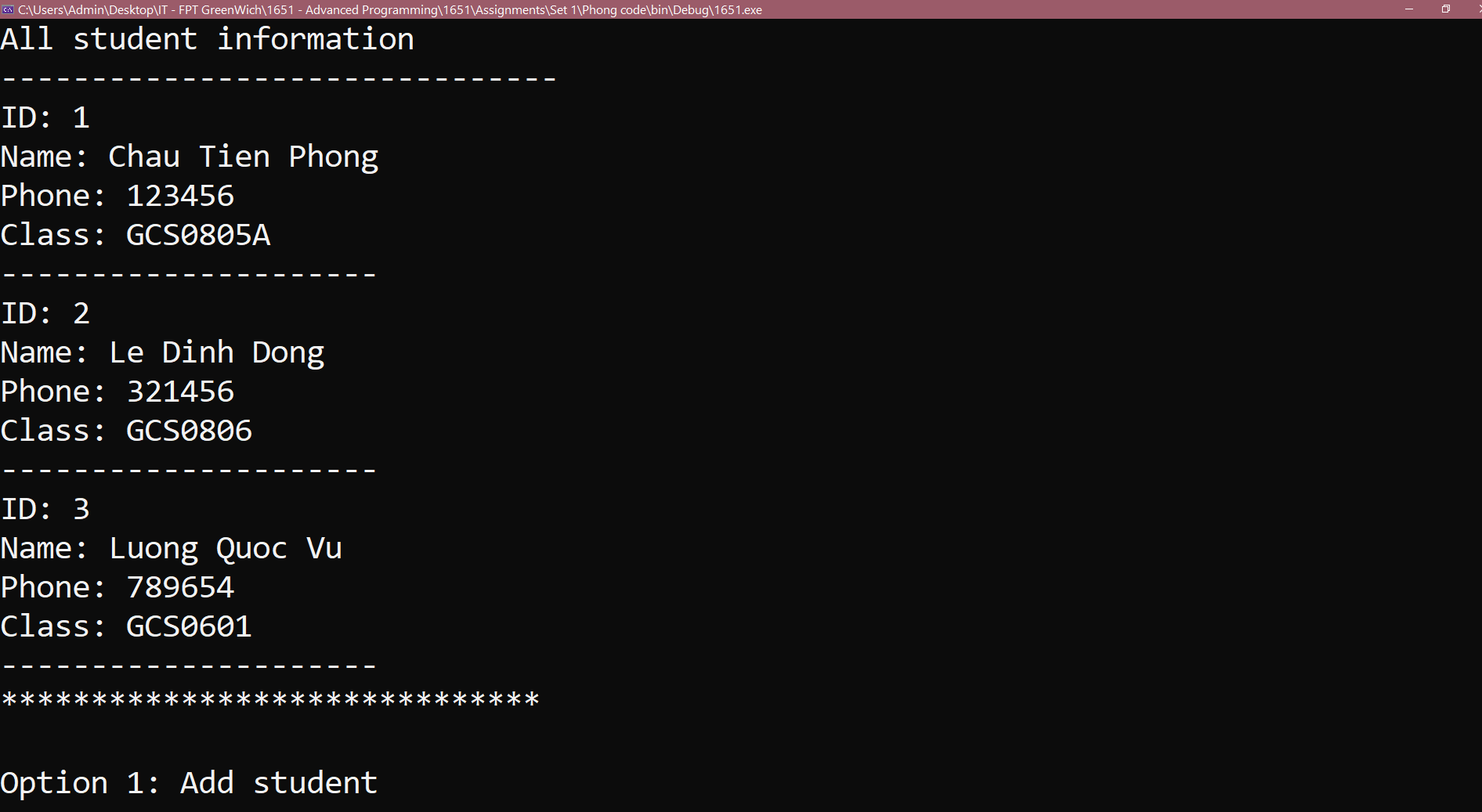


Figure Function read data

* 1. **Find a student by ID**

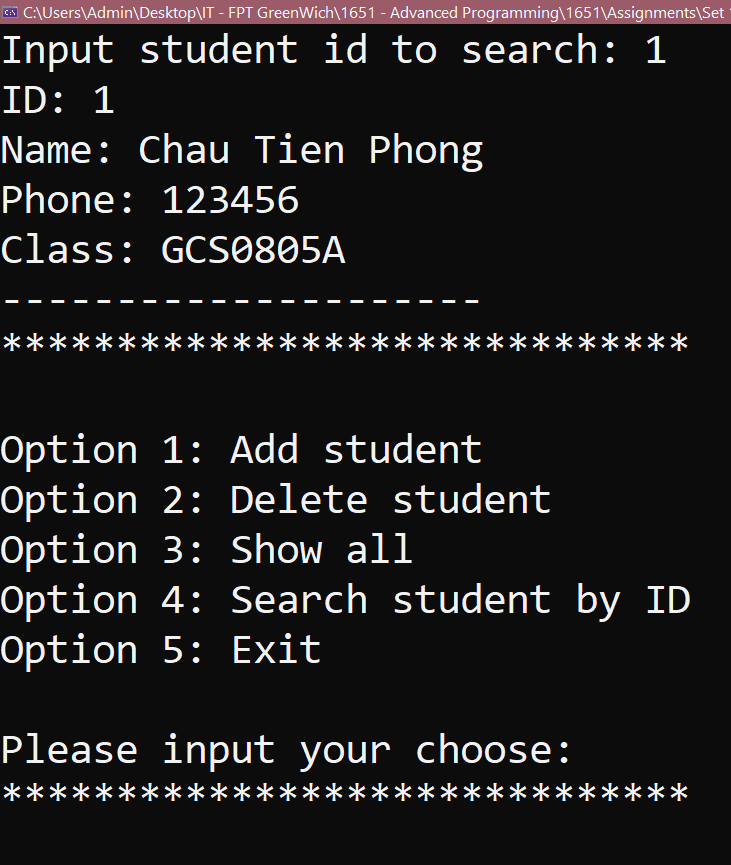


Figure Function finding

* 1. **Exit**

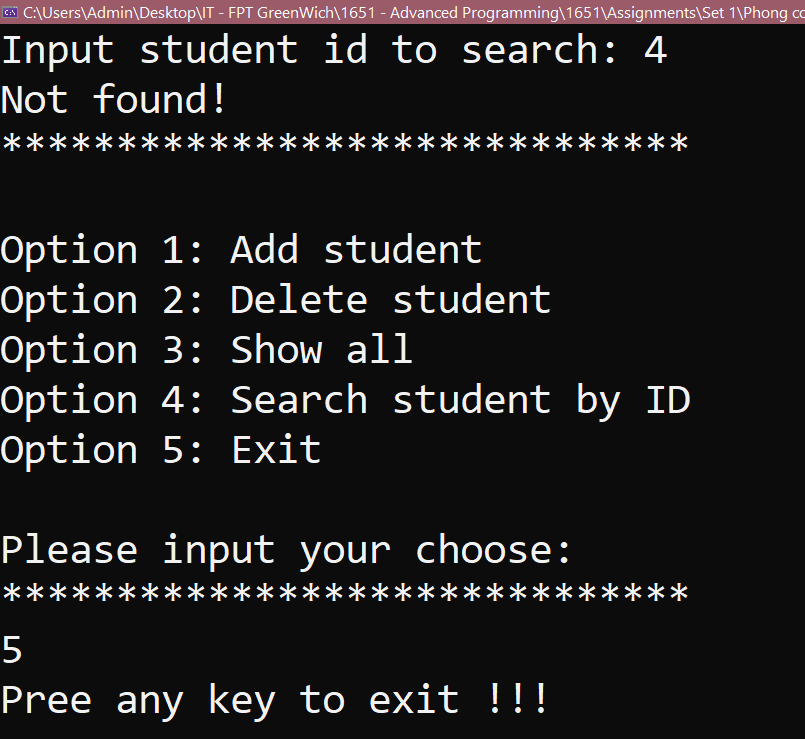


Figure Exit option

# P4 Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types.

1. **Design Pattern**
   1. **Definition**

A design pattern is a generic repeatable solution to a typically occurring problem in software design in software engineering. A design pattern isn't a finished design that can be turned into code right away. It's a description or template for solving an issue that may be applied to a variety of scenarios.

* 1. **Purpose of Design Patterns**

Design patterns can help developers save time by providing tried-and-true development paradigms. Effective software design necessitates taking into account challenges that may not be apparent until later in the implementation process. For coders and architects familiar with the patterns, reusing design patterns helps to eliminate subtle flaws that can lead to large difficulties and improves code readability.

People frequently only know how to apply specific software design strategies to specific challenges. These methods are challenging to apply to a broader set of issues. Design patterns are generic solutions that are documented in a way that does not necessitate specifics connected to a specific situation.

Patterns also allow engineers to speak about software interactions using well-known, well-understood terms. Compared to ad-hoc designs, common design patterns can be improved over time, making them more robust.

* 1. **Types of Design Patterns**
     1. **Creation Design Patterns**

These design patterns are focused on class instantiation or object production. Two subsets of these patterns are class-creational patterns and object-creational patterns. In the instantiation process, class-creation patterns make extensive use of inheritance, whereas object-creation patterns make extensive use of delegation. Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype are all creational design patterns.

* + 1. **Structural Design Patterns**

These design patterns are concerned with combining different classes and objects to form larger structures and add additional functionality. Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy are structural design patterns.

* + 1. **Behavioral Design Patterns**

Behavioral patterns are all about identifying and recognizing shared communication patterns between things. Chain of duty, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor are examples of behavioral patterns.

1. **Creation Design Patterns**
   1. **Singleton Design Pattern**
      1. **Definition**

According to the Singleton Pattern, all you have to do is "create a class with only one instance and offer a global point of access to it." To put it another way, a class must make sure that only one instance is produced and that only one object can be utilized by all other classes.

The singleton design pattern is divided into two types:

* + - * Early Instantiation: the generation of an instance at the time of loading.
      * Lazy Instantiation: creates an instance only when it is needed.
    1. **Example**

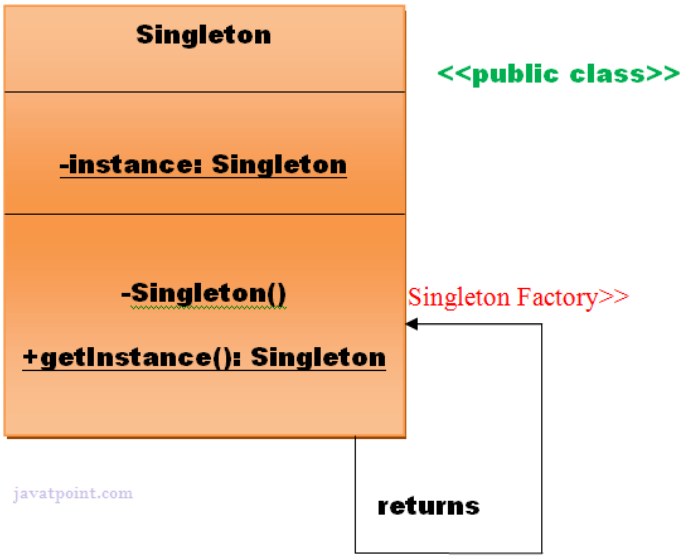


Figure Singleton Pattern

The Singleton design pattern has the following advantages:

* + - * It saves memory because an object is not created for each request. Only one instance is utilized over and over.
      * The Singleton design pattern is most commonly found in multi-threaded and database applications. It's utilized for logging, caching, thread pools, and other configuration options.
  1. **A Factory Pattern**
     1. **Definition**

A Factory Pattern, also known as a Factory Method Pattern, states that you should only create an interface or abstract class for producing objects and leave it up to the subclasses to pick which class to instantiate. In other words, subclasses are in charge of creating the class instance.

* + 1. **Example**

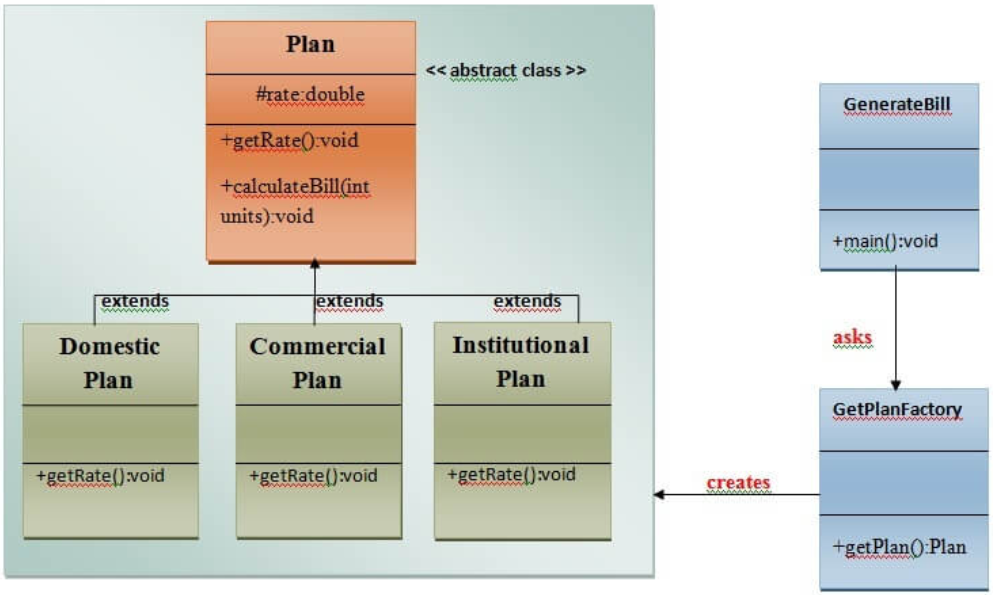


Figure Factory Pattern

* Virtual Constructor is another name for the Factory Method Pattern.

Factory Design Pattern Benefits

* Factory Method Pattern allows sub-classes to select the type of objects they want to produce.
* It encourages loose coupling by removing the need for application-specific classes to be bound into the code. That is, the code only interacts with the resultant interface or abstract class, and it will function with any classes that implement or extend that interface or abstract class.

Application of the Factory Design Pattern

* When a class doesn't know what sub-classes will be needed to generate the objects
* When a class wants its sub-classes to specify the objects that will be created
* When the parent classes decide whether or not to create objects for their sub-classes.
  1. **Abstract Factory**
     1. **Definition**

According to the Abstract Factory Pattern, you can create families of linked (or dependent) items by simply defining an interface or abstract class without identifying their particular sub-classes. This means that a class can return a factory of classes using Abstract Factory. As a result, the Abstract Factory Pattern is a level higher than the Factory Pattern. Kit is another name for an Abstract Factory Pattern.

* + 1. **Example**

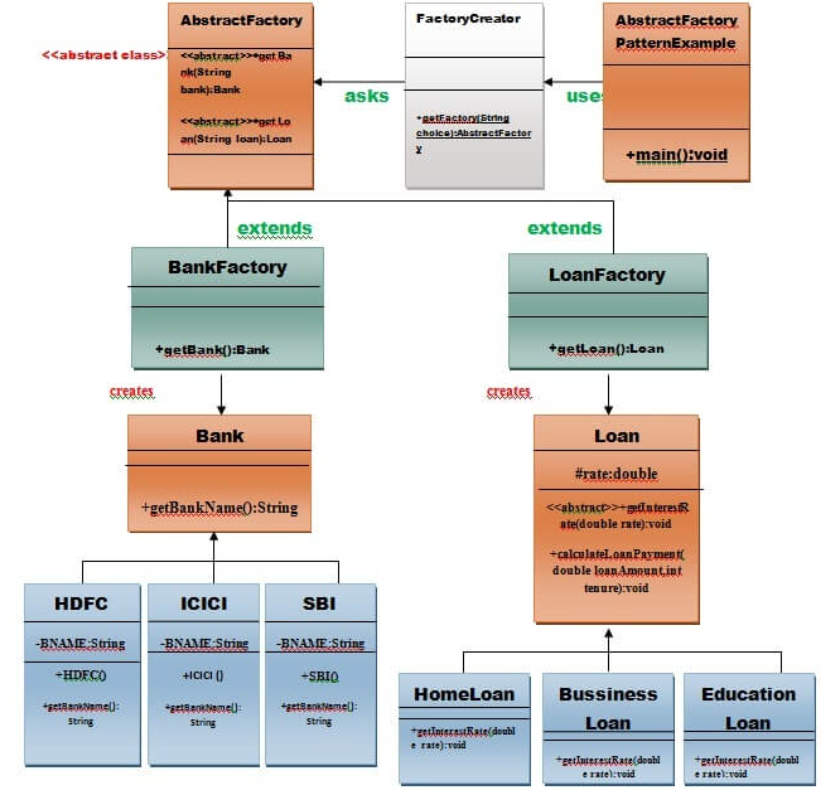


Figure Abstract Factory Pattern

Abstract Factory Pattern's Benefits

* + - * The Abstract Factory Pattern keeps client code separate from concrete (implementation) classes.
      * It simplifies the transfer of object families.
      * It encourages object consistency.

Application of the Abstract Factory Pattern

* + - * When the system must be independent of the creation, composition, and representation of its objects.
      * This constraint must be imposed when a family of related items must be utilized together.
      * When you wish to create a library of objects that only discloses interfaces and hides implementations.
      * When the system must be configured with one of several object families.

1. **Structural Design Patterns**
   1. **Adapter Pattern**
      1. **Definition**

An Adapter Pattern simply "converts the interface of a class into another interface that a customer desires," according to the definition. To put it another way, to provide an interface that meets the needs of the client while utilizing the services of a class with a different interface. Wrapper is another name for the Adapter Pattern.

* + 1. **Example**

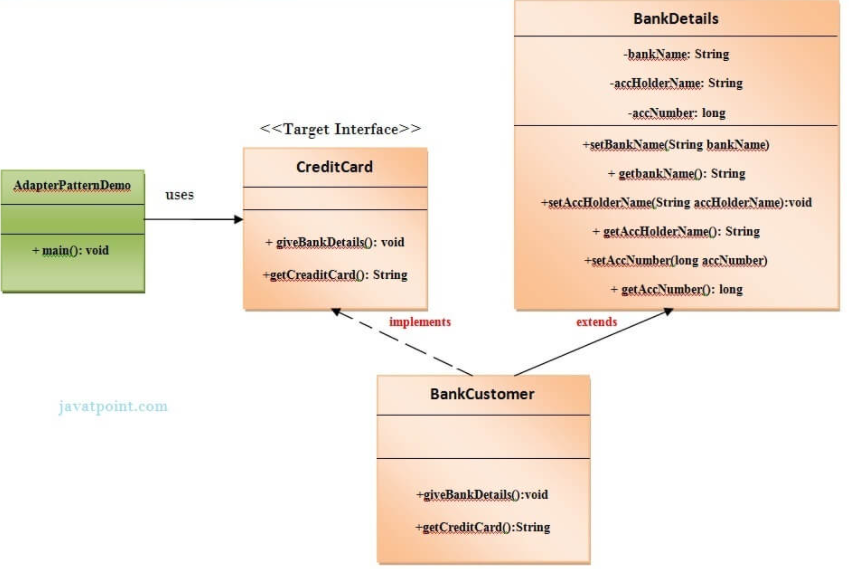


Figure Adapter Pattern

The Benefits of the Adapter Pattern

* + - * It enables the interaction of two or more previously incompatible items, as well as the reuse of existing functionality.

Application of the Adapter pattern:

* + - * When an object has to use a class that has an incompatible interface.
      * When you need to make a reusable class that works with other classes that don't have the same interface.
      * When you need to make a reusable class that works with other classes that don't have the same interface.
  1. **Bridge Pattern**
     1. **Definition**

Simply "decouple the functional abstraction from the implementation such that the two can vary independently," according to the Bridge Pattern. The Handle or Body Pattern is another name for the Bridge Pattern.

* + 1. **Example**

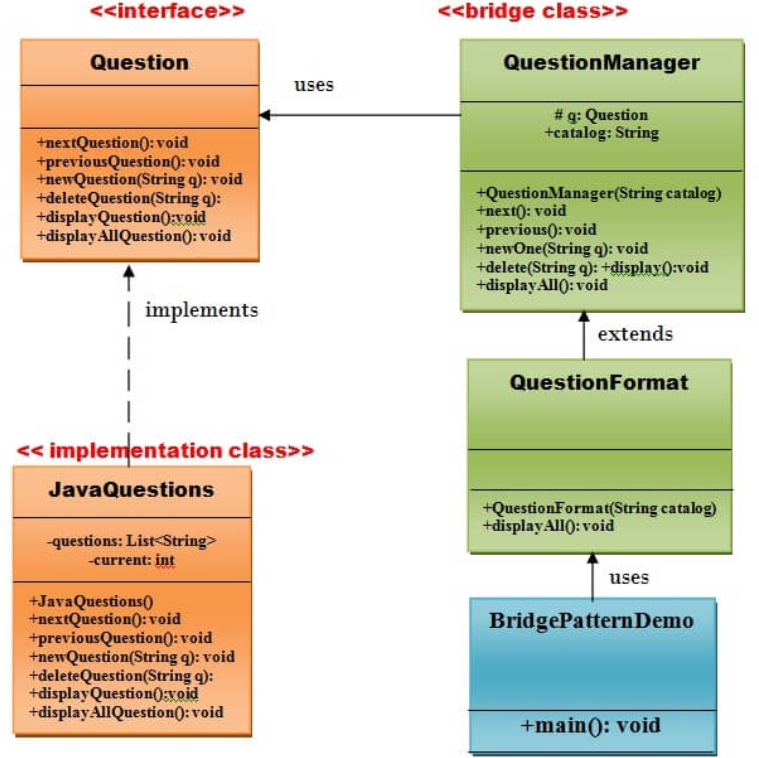


Figure Bridge Pattern

The Bridge Pattern has the following advantages:

* + - * It allows for the separation of implementation and interface;
      * It enhances extensibility.
      * It allows the client to be unaware of implementation specifics.

When you don't want a permanent binding between the functional abstraction and its implementation, use the Bridge Pattern.

* + - * When both the functional abstraction and its implementation need to be extended using sub-classes, use the Bridge Pattern.
      * It's mostly used in places where changes to the implementation don't affect the clients**.**
  1. **A Composite Pattern**
     1. **Definition**

**A Composite Pattern** says that just "allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".

* + 1. **Example**

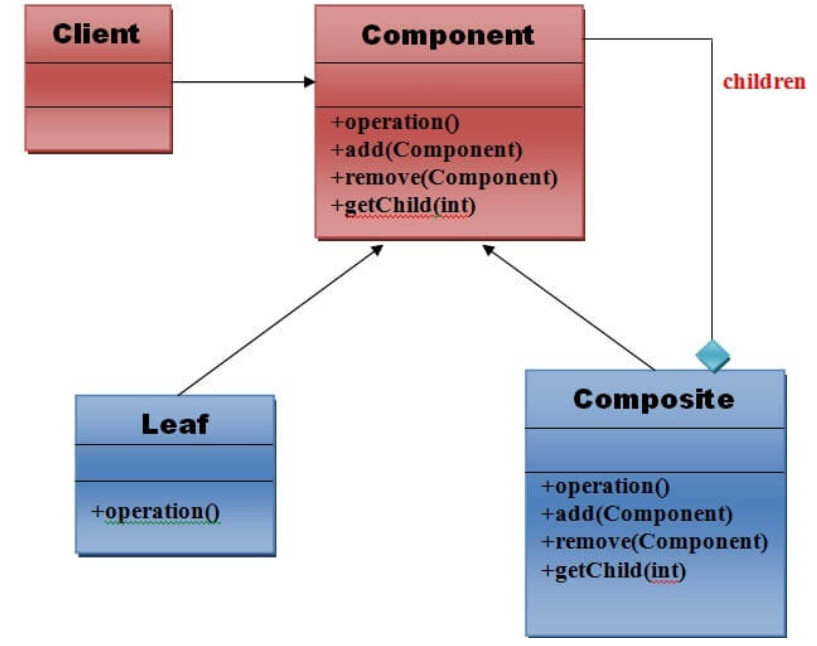


Figure Composite Pattern

* + - * It defines class hierarchies that contain primitive and complicated objects as an advantage of the Composite Design Pattern.
      * It makes it simpler to add new types of components.
      * It provides structure flexibility through a manageable class or interface.
      * When you want to express a full or partial hierarchy of items, use the Composite Pattern.
      * When responsibilities must be introduced to particular objects dynamically without affecting other objects. Where the object's duty may change from time to time.
  1. **A Decorator Pattern**
     1. **Definition**

**A Decorator Pattern** says that just "attach a flexible additional responsibility to an object dynamically". In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime. The Decorator Pattern is also known as Wrapper.

* + 1. **Example**

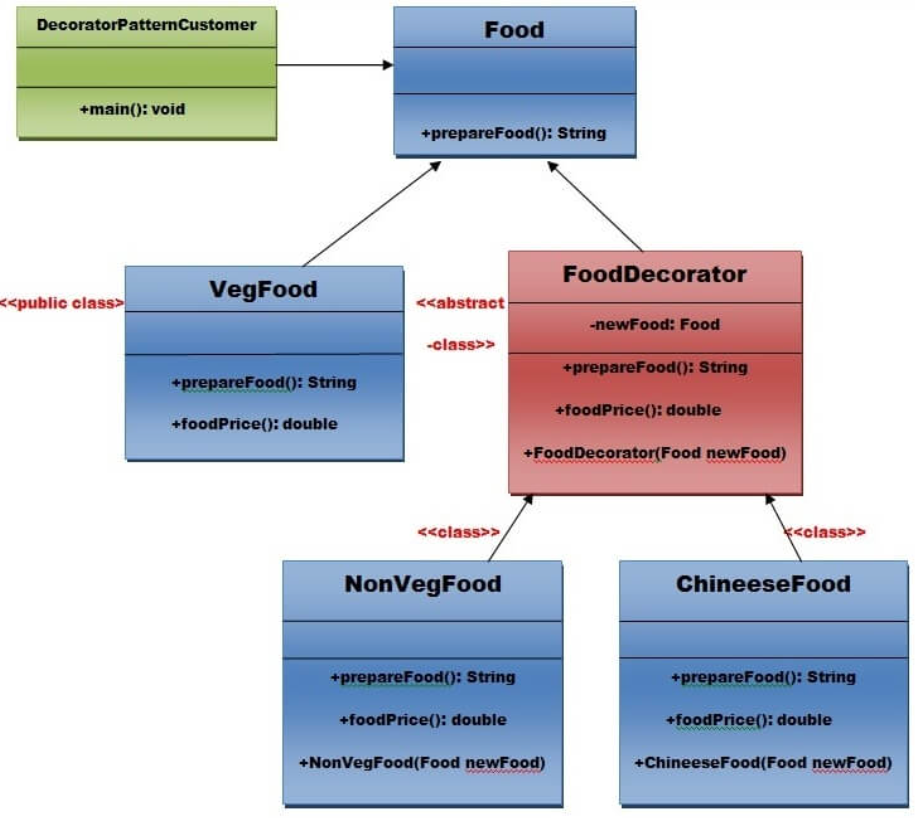


Figure Decorator Pattern

**Usage of Decorator Pattern**

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

1. **Behavioral Design Patterns**
   1. **Chain of responsibility**
      1. **Definition**

In **chain of responsibility**, sender sends a request to a chain of objects. The request can be handled by any object in the chain. A Chain of Responsibility Pattern says that just "avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request". For example, an ATM uses the Chain of Responsibility design pattern in money giving process. In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

* + 1. **Example**

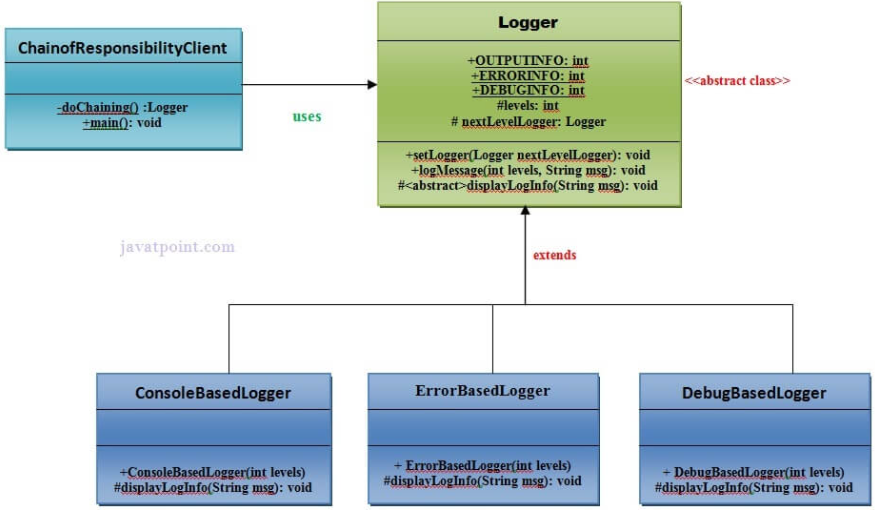


Figure Chain of Responsibility Pattern

**Advantage of Chain of Responsibility Pattern**

* It reduces the coupling.
* It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

**Usage of Chain of Responsibility Pattern**:

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way.
  1. **Command Pattern**
     1. **Definition**

A **Command Pattern** says that "encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command". It is also known as Action or Transaction.

* + 1. **Example**

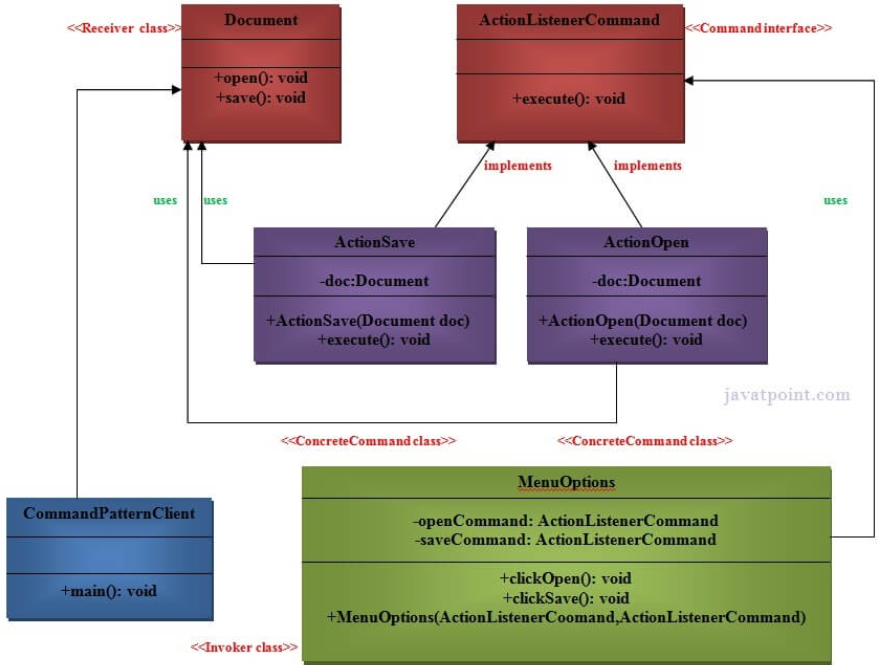


Figure Command Pattern

**Advantage of command pattern**

* It separates the object that invokes the operation from the object that actually performs the operation.
* It makes easy to add new commands, because existing classes remain unchanged.

**Usage of command pattern**

* When you need parameterize objects according to an action perform.
* When you need to create and execute requests at different times.
* When you need to support rollback, logging or transaction functionality.
  1. **Interpreter Pattern**
     1. **Definition**

An **Interpreter Pattern** says that "to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language". Basically the Interpreter pattern has limited area where it can be applied. We can discuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions that is why it is not frequently used.

* + 1. **Example**

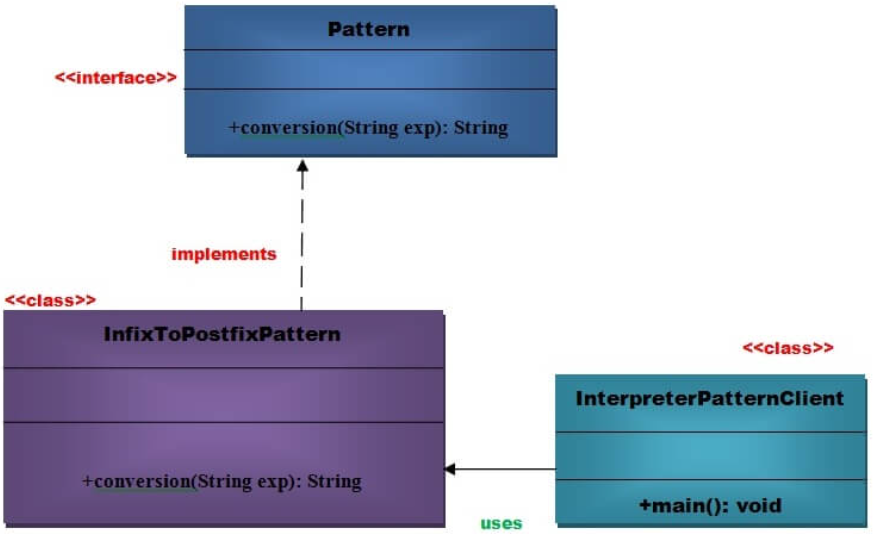


Figure Interpreter Pattern

**Advantage of Interpreter Pattern**

* It is easier to change and extend the grammar.
* Implementing the grammar is straightforward.

**Usage of Interpreter pattern**:

* When the grammar of the language is not complicated.
* When the efficiency is not a priority.
  1. **Iterator Pattern**
     1. **Definition**

According to GoF, **Iterator Pattern** is used "to access the elements of an aggregate object sequentially without exposing its underlying implementation". The Iterator pattern is also known as Cursor.

* + 1. **Example**

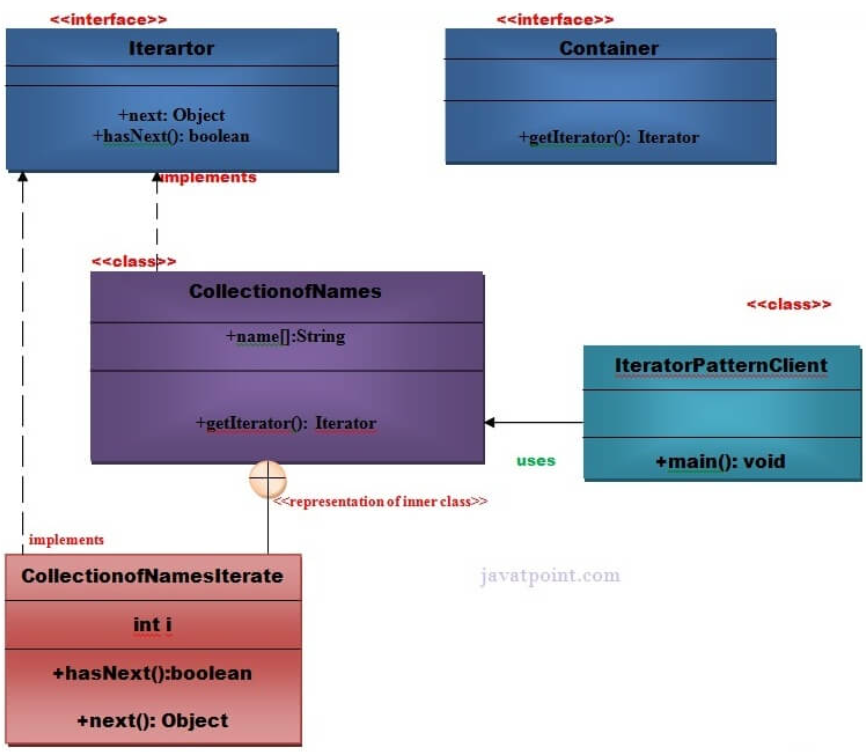


Figure Iterator Pattern

**Advantage of Iterator Pattern**

* It supports variations in the traversal of a collection.
* It simplifies the interface to the collection.

**Usage of Iterator Pattern**:

* When you want to access a collection of objects without exposing its internal representation.
* When there are multiple traversals of objects need to be supported in the collection.

# 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: March 1. 2022]

Tutorialspoint () What is abstraction in C#? [online] Available at: <https://www.tutorialspoint.com/What-is-abstraction-in-Chash> [Accessed at: March 1. 2022]

Tutorialspoint () OOAD - Object Oriented Paradigm [online] Available at: <https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.htm> [Accessed at: March 1. 2022]

Tutorialspoint () Object-Oriented Paradigm [online] Available at: <https://www.tutorialspoint.com/software_architecture_design/object_oriented_paradigm.htm> [Accessed at: March 1. 2022]

Sourcemaking () Design Patterns [online] Available at: <https://sourcemaking.com/design_patterns> [Accessed at: March 1. 2022]

Samual, S (2018) Association, Composition and Aggregation in C# [online] Available at: <https://www.tutorialspoint.com/Association-Composition-and-Aggregation-in-Chash> [Accessed at: March 1. 2022]

Koderhq () C# Composition Tutorial [online] Available at: <https://www.koderhq.com/tutorial/csharp/oop-composition/> [Accessed at: March 1. 2022]

Jignest, T (2019) Understanding Polymorphism In C# [online] Available at: <https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/> [Accessed at: March 1. 2022]

Jignest, T (2019) Association, Aggregation and Composition [online] Available at: <https://www.c-sharpcorner.com/UploadFile/ff2f08/association-aggregation-and-composition/> [Accessed at: March 1. 2022]

Geeksforgeeks (2021) C# | Abstract Classes [online] Available at: <https://www.geeksforgeeks.org/c-sharp-abstract-classes/> [Accessed at: March 1. 2022]

Geeksforgeeks (2020) C# | Class and Object [online] Available at: <https://www.geeksforgeeks.org/c-sharp-class-and-object/> [Accessed at: March 1. 2022]

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: March 1. 2022]

Dotnettutorials () Dependency Injection in C# [online] Available at: <https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/> [Accessed at: March 1. 2022]

Atul, R () Dependency Injection in C# [online] Available at: <https://www.c-sharpcorner.com/UploadFile/85ed7a/dependency-injection-in-C-Sharp/> [Accessed at: March 1. 2022]

Arjun, T (2018) Composition vs Aggregation in C# [online] Available at: <https://www.tutorialspoint.com/composition-vs-aggregation-in-chash> [Accessed at: March 1. 2022]