Higher Nationals in Computing

Unit 20: Advanced Programming

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 16: Cloud computing**		
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	P3	P4	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). The working application must also be demonstrated.

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

Table of Figures

ASSIGNMENT 2 ANSWERS

P3 Build an application derived from UML class diagrams.

1. UML diagrams of application

The UML class diagram is as follows, based on the user requirements for the system that is used to manage students in school:

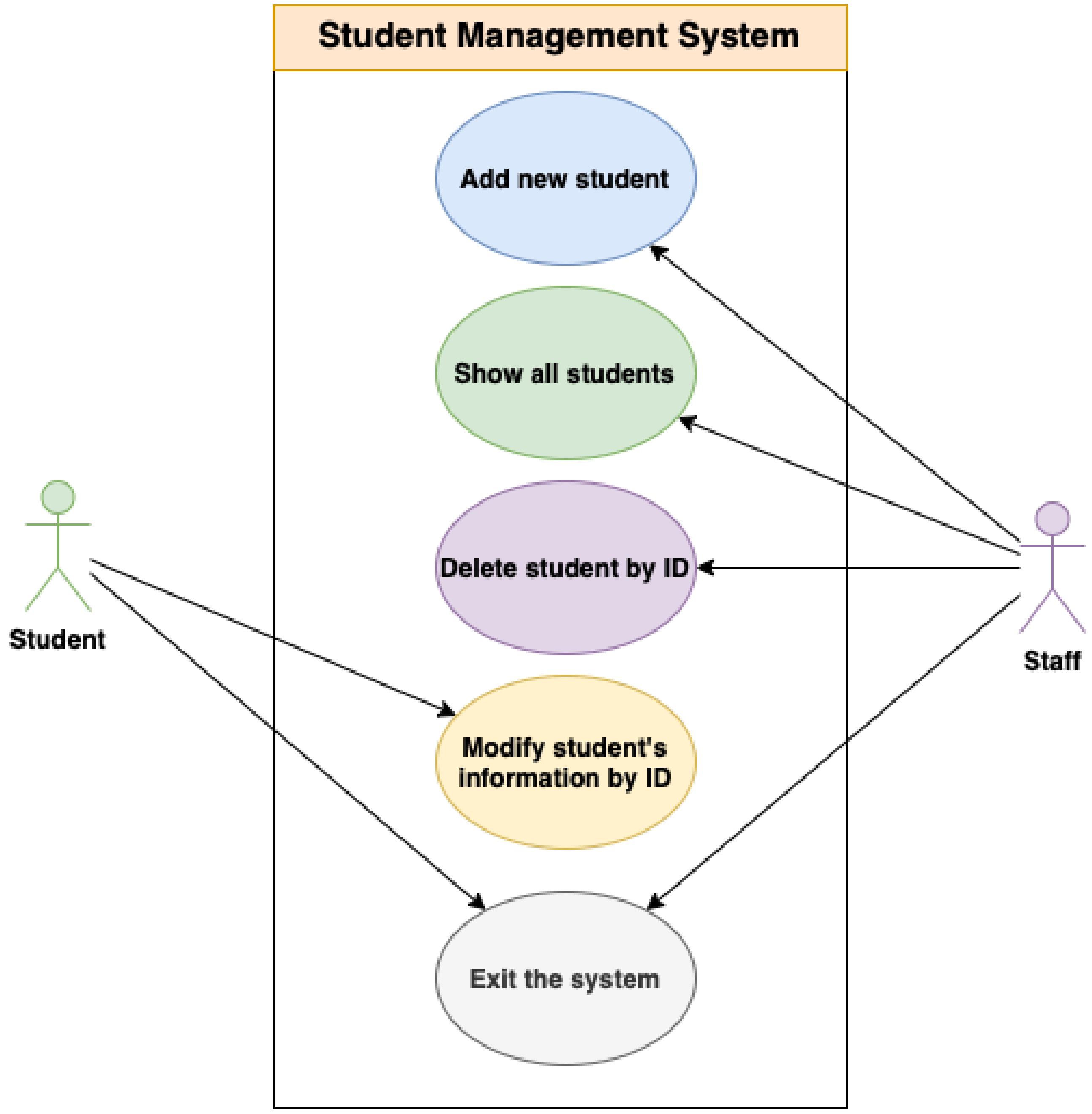


Figure 1 Use case diagram

2. Source code of application based on UML

2.1. Source in Student.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _1651
{
    class Student
    {
        // declaring attributé
        public string idStu;
        public string nameStu;
        public int phNoStu;
        public string addressStu;
        //declaring contrutors: khai báo hàm dựng
        public Student()
        {
            idStu = "";
            nameStu = "";
            phNoStu = 0;
            addressStu ="";
        }

        public Student(string ID,string Name, int Phone, string Class)
        {
            this.idStu = ID;
            this.nameStu = Name;
            this.phNoStu = Phone;
            this.addressStu = Class;
        }

        public void Display()
        {
            Console.WriteLine("Student ID: " + idStu);
            Console.WriteLine("Student Name: " + nameStu);
            Console.WriteLine("Student Phone: " + phNoStu);
            Console.WriteLine("Student Address: " + addressStu);
            Console.WriteLine("---------------------");
        }
        public string stuid
        {
            get { return idStu; }
            set { idStu = value; }
        }
        public string stuname
        {
            get { return nameStu; }
            set { nameStu = value; }
        }
        public int stuphone
        {
            get { return phNoStu; }
            set { phNoStu = value; }
        }

        public string addressstu
        {
            get { return addressStu; }
            set { addressStu = value; }
        }
    }
}
```

2.2. Source code of Program.cs

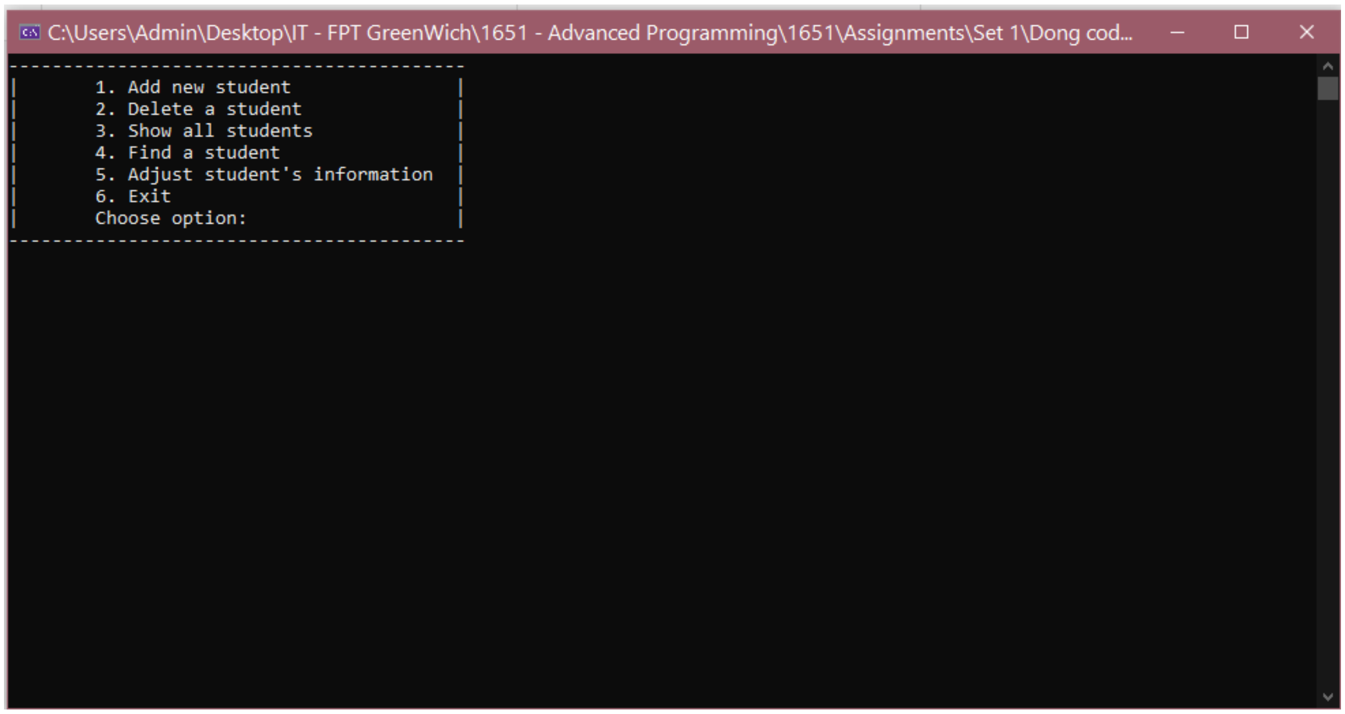
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _1651
{
    class Program
    {
        static void Main(string[] args)
        {
            StudentManage manage = new StudentManage();
            int option = 0;
            do
            {
                option = manage.Menu();
                switch(option)
                {
                    case 1:
                        Console.Clear();
                        manage.AddStudent();
                        break;
                    case 2:
                        Console.Clear();
                        Console.Write("Input student id to delete: ");
                        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.Clear();
                        Console.Write("Input student id to adjust: ");
                        id = Console.ReadLine();
                        manage.AdjustInfo(id);
                        break;
                    case 6:
                        Console.Write(" Press any key to exit");
                        break;
                    default:
                        Console.Clear();
                        Console.WriteLine("Invalid option. Choose option 
again!");
                        break;
                }
            } while (option != 6);

            Console.ReadKey();
        }
    }
}
```

2.3. Source code of 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();
            Console.Write("Input student ID: ");
            s.stuid = Console.ReadLine();
            Console.Write("Input student Name: ");
            s.stuname = Console.ReadLine();
            Console.Write("Input student Phone: ");
            s.stuphone = Convert.ToInt32(Console.ReadLine());
            Console.Write("Input student address: ");
            s.addressstu = Console.ReadLine();
            Console.WriteLine("---------------------");
            Console.WriteLine("The student have been added!");
            list.Add(s);
        }

        public void DisplayAll()
        {
            foreach(var s in list)
            {
                s.Display();
            }
        }

        public void SearchById(string id)
        {
            bool found = false;
            foreach(var s in list)
            {
                if(s.idStu.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.idStu.Equals(id))
                {
                    list.Remove(s);
                    found = true;
                    Console.WriteLine("-----------------------------------------
");

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

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

                    break;
                }
            }
            if (!found)
            {
                Console.WriteLine("Invalid ID!");
            }
        }

        public void AdjustInfo (string id)
            {
            bool found = false;
            foreach (var s in list)
                {
                if (s.idStu.Equals(id))
                {

                    found = true;
                    Console.WriteLine("------------------------------------------
");

                    Console.WriteLine("The student ID " +id+ " will be
adjust!");
                    s.stuid = id;
                    Console.Write("Input student Name: ");
                    s.stuname = Console.ReadLine();
                    Console.Write("Input student Phone: ");
                    s.stuphone = Convert.ToInt32(Console.ReadLine());
                    Console.Write("Input student address: ");
                    s.addressstu = Console.ReadLine();
                    Console.WriteLine("---------------------");
                    Console.WriteLine("The student have been adjusted!");
                    Console.WriteLine("------------------------------------------
");
                }
            }
            if (!found)
            {
                Console.WriteLine("Invalid ID!");
            }
        }

        public int Menu()
        {
            int select = 0;
            Console.WriteLine("------------------------------------------");
            Console.WriteLine("|\t1. Add new student \t\t |\n|\t2. Delete a
student\t\t |\n|\t3. Show all students\t\t |\n|\t4. Find a student \t\t |\n|\t5.
Adjust student's information  |\n|\t6. Exit\t\t\t\t |");
            Console.WriteLine("|\tChoose option: \t\t\t |");
            Console.WriteLine("------------------------------------------");
            select = Convert.ToInt32(Console.ReadLine());
            return select;
        }
    }
}
```

3. Application screenshots of functions

3.1. Menu function



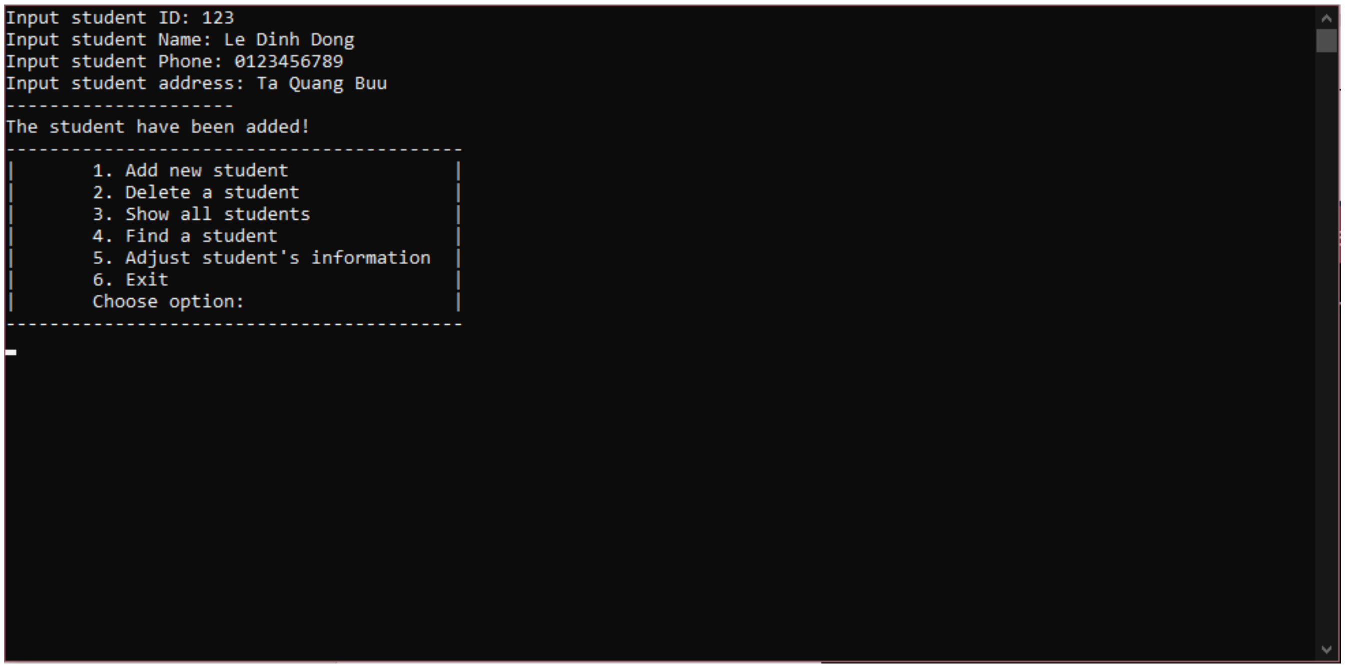
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651 - Advanced Programming\1651\Assignments\Set 1\Dong cod...

```
-----------------------------------------
|        1. Add new student             |
|        2. Delete a student            |
|        3. Show all students           |
|        4. Find a student              |
|        5. Adjust student's information |
|        6. Exit                        |
|        Choose option:                 |
-----------------------------------------
```

Figure 2 Menu

3.2. Add function

After you input the student information to the system, you need to input all data of each field that the system requires, unless it shows error. If everything is okay, there will appear the nofitication of successful “The student have been added”

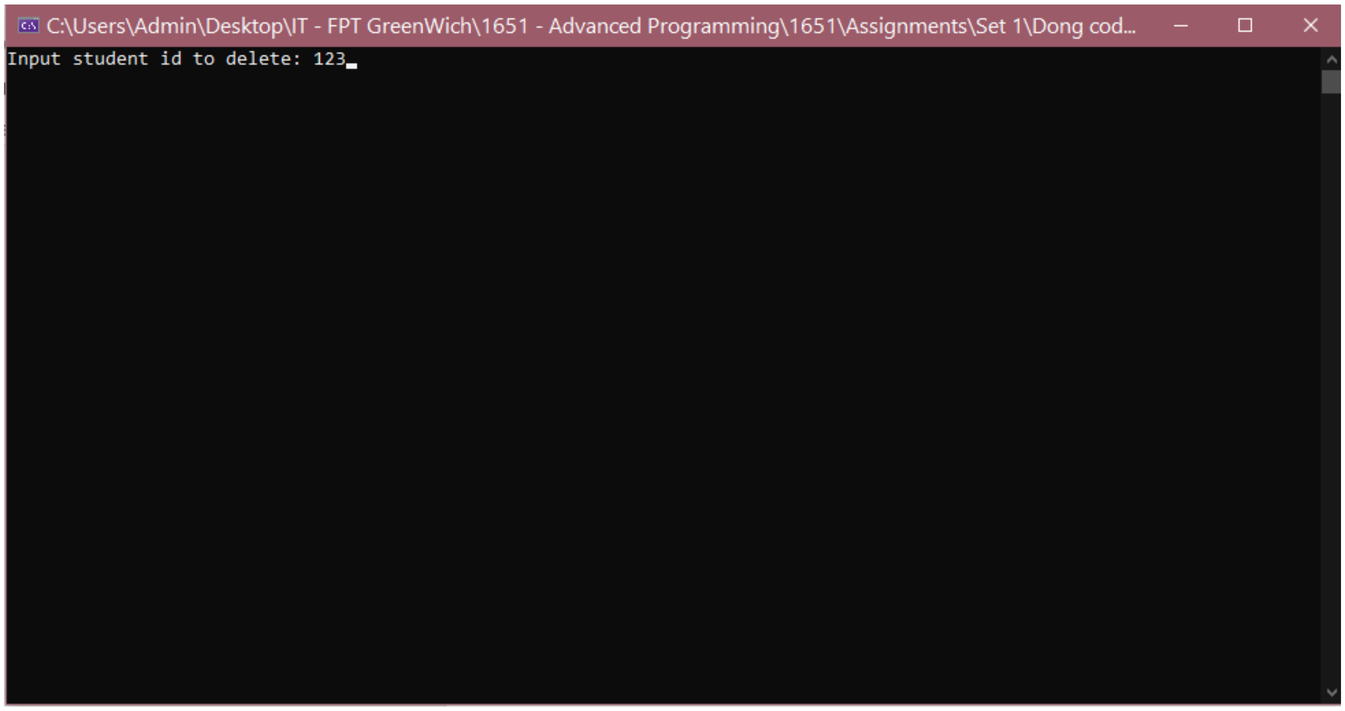


```
Input student ID: 123
Input student Name: Le Dinh Dong
Input student Phone: 0123456789
Input student address: Ta Quang Buu
---------------------
The student have been added!
------------------------------------------
|       1. Add new student               |
|       2. Delete a student              |
|       3. Show all students             |
|       4. Find a student                |
|       5. Adjust student's information  |
|       6. Exit                          |
|       Choose option:                   |
------------------------------------------
```

Figure 3 Addition Function

3.3. Delete function

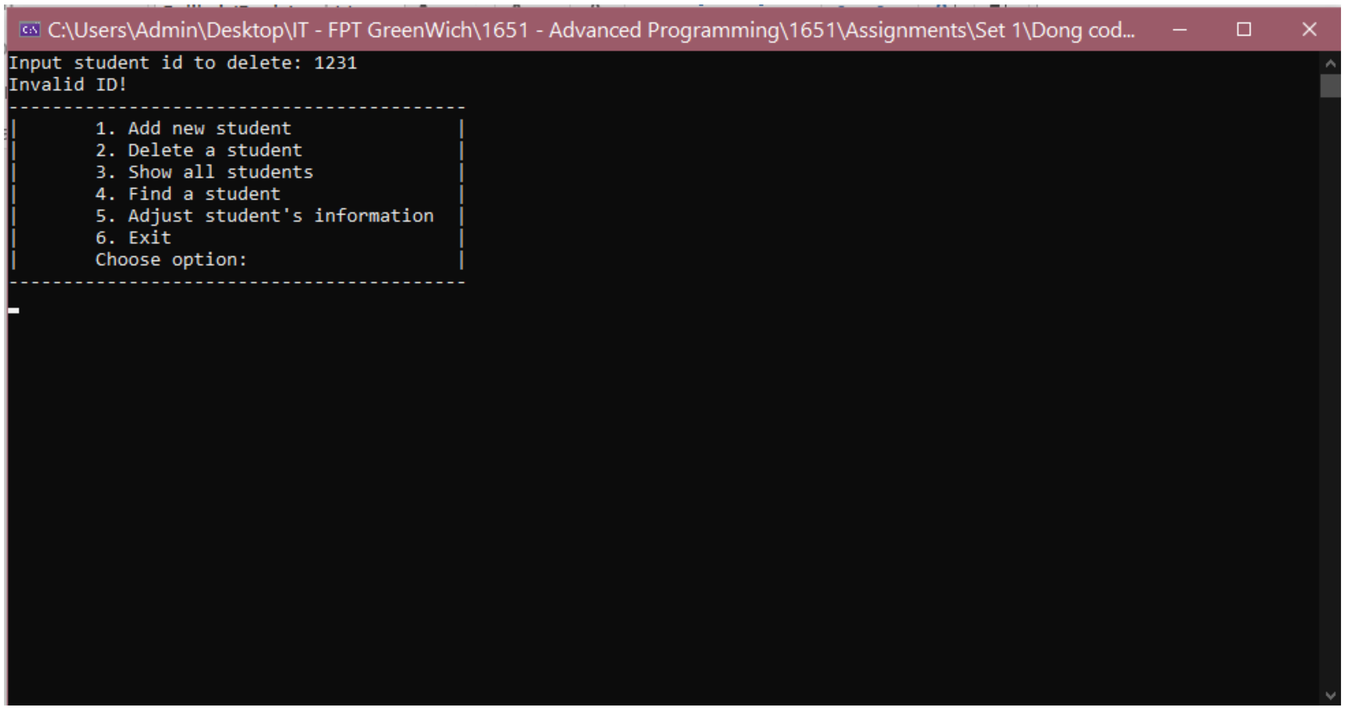
Same with addition function, delete needs to get ID of student which user want to delete.



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651 - Advanced Programming\1651\Assignments\Set 1\Dong cod...
Input student id to delete: 123
```

Figure 4 Delete function

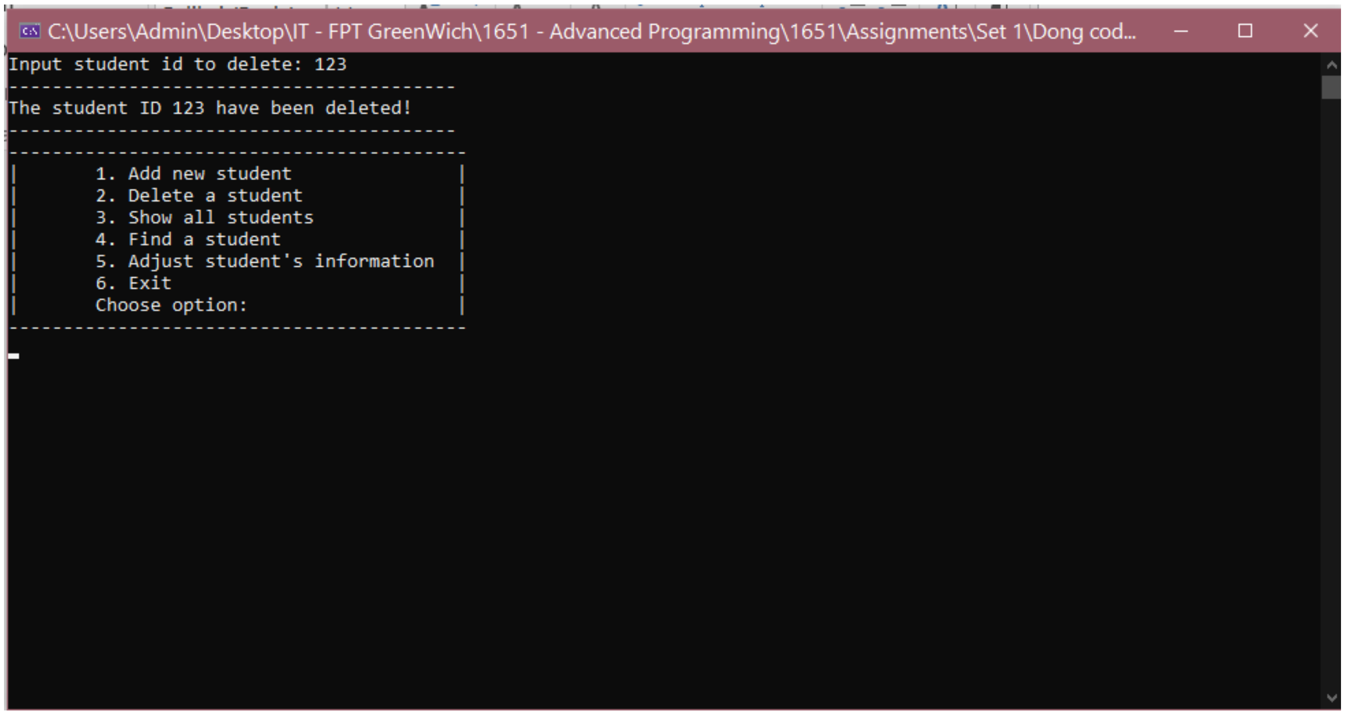
If you enter an invalid ID, the message "Invalid ID!" will appear, followed by the menu.



```
Input student id to delete: 1231
Invalid ID!
-----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
-----------------------------------------
```

Figure 5 Nofitication of invalid ID

If you enter a valid id and it is found in the system, you will receive a success notification.

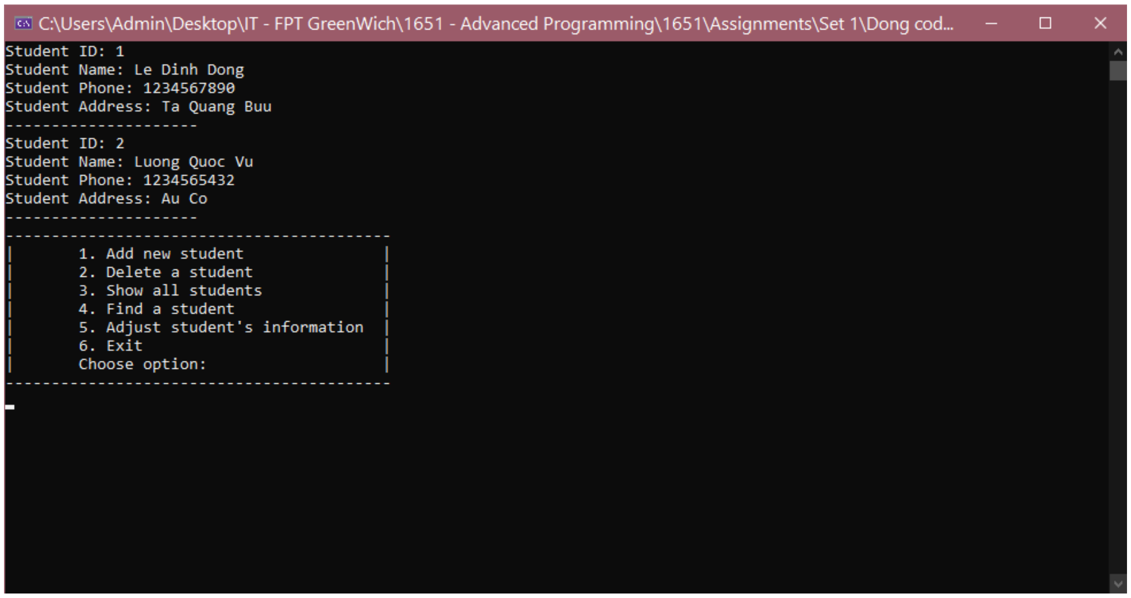


```
Input student id to delete: 123
----------------------------------------
The student ID 123 have been deleted!
----------------------------------------
-----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
-----------------------------------------
```

Figure 6 Nofitication of success

3.4. Show all student function

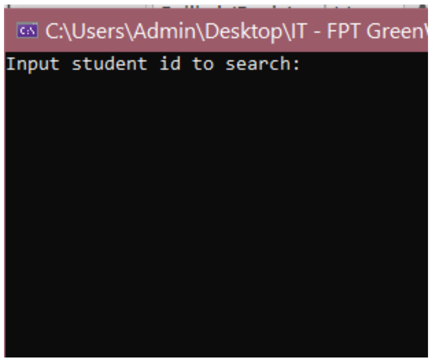
Nothing will appear if you haven't supplied any student information, and the menu will appear again. If you have it, you will be able to:



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651 - Advanced Programming\1651\Assignments\Set 1\Dong cod...
Student ID: 1
Student Name: Le Dinh Dong
Student Phone: 1234567890
Student Address: Ta Quang Buu
--------------------
Student ID: 2
Student Name: Luong Quoc Vu
Student Phone: 1234565432
Student Address: Au Co
--------------------
-----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
-----------------------------------------
```

Figure 7 Show student info

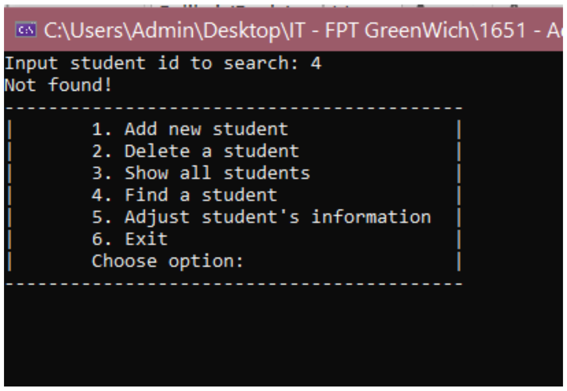
3.5. Find a student by ID function



```
C:\Users\Admin\Desktop\IT - FPT Green
Input student id to search:
```

Figure 8 Find student by ID function

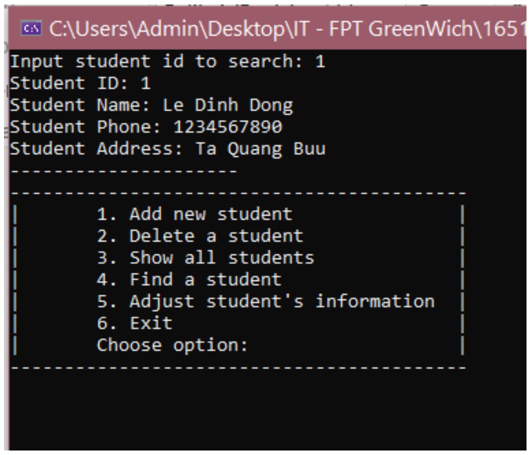
The warning "Not Found!" will appear if you enter an invalid ID.



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651 - A
Input student id to search: 4
Not found!
-----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
-----------------------------------------
```

Figure 9 Nofitication of invalid ID

The student information from your finding will be displayed if it is genuine.

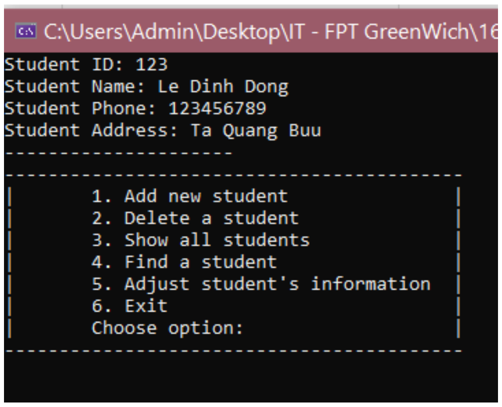


```
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651
Input student id to search: 1
Student ID: 1
Student Name: Le Dinh Dong
Student Phone: 1234567890
Student Address: Ta Quang Buu
--------------------
----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
----------------------------------------
```

Figure 10 Nofitication of successful

3.6. Adjust student's information

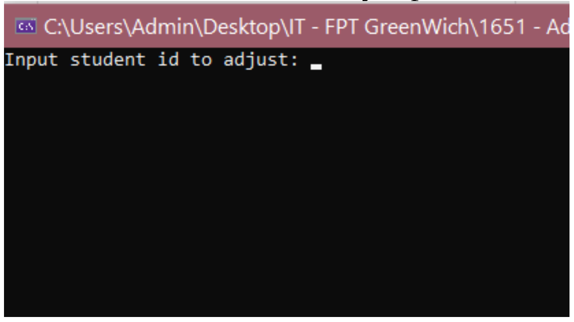
This is information of a student which have been input



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\16
Student ID: 123
Student Name: Le Dinh Dong
Student Phone: 123456789
Student Address: Ta Quang Buu
--------------------
----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
----------------------------------------
```

Figure 11 - Show information to adjust

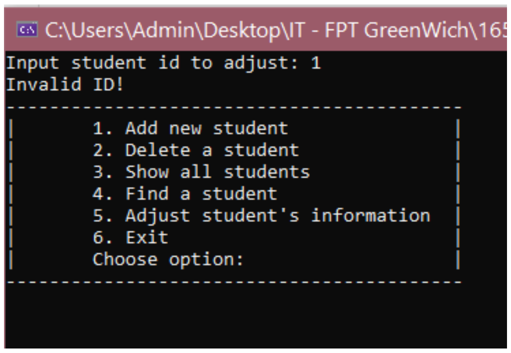
After that we change the information of this student, by input ID first.



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\1651 - Ad
Input student id to adjust: _
```

Figure 12 Input ID to change infor

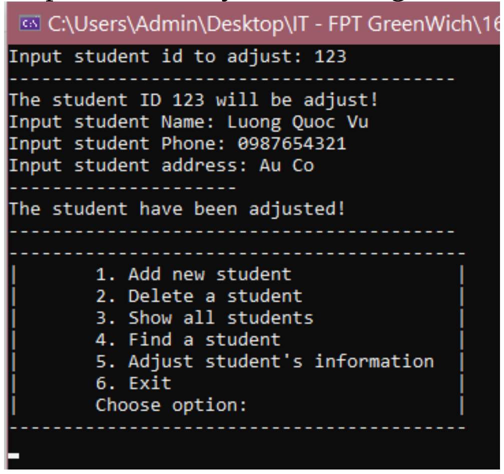
If invalid ID, show nofitication.



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\165
Input student id to adjust: 1
Invalid ID!
----------------------------------------
|       1. Add new student             |
|       2. Delete a student            |
|       3. Show all students           |
|       4. Find a student              |
|       5. Adjust student's information|
|       6. Exit                        |
|       Choose option:                 |
----------------------------------------
```

Figure 13 Nofitication of failure

If valid ID, they system will keep that ID and you can change Name, Phone and Address



```
C:\Users\Admin\Desktop\IT - FPT GreenWich\16
Input student id to adjust: 123
----------------------------------------
The student ID 123 will be adjust!
Input student Name: Luong Quoc Vu
Input student Phone: 0987654321
Input student address: Au Co
--------------------
The student have been adjusted!
----------------------------------------
-----------------------------------------
|       1. Add new student              |
|       2. Delete a student             |
|       3. Show all students            |
|       4. Find a student               |
|       5. Adjust student's information |
|       6. Exit                         |
|       Choose option:                  |
-----------------------------------------
```

Figure 14 Nofitication of successful

And show nofitication if successful.

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

1. Design Pattern

1.1. Definition

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

- **Usage of Design Pattern**: Design Patterns have two main usages in software development.

- **Common platform for developers**: Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.
- **Best Practices**: Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

1.2. Purpose of Design Patterns

A design pattern is a tool for communication, not a template for software assembly. If you have a hierarchy of abstract interfaces with multiple parallel concrete implementations, then the creation of an intermediate object that has knowledge of which concrete hierarchy is required for consistency is preferable to distributing a type check throughout the code.

A design pattern is a generalized solution to a reoccurring problem in software development. It is not a solution itself, but a means of describing an approach. Most programmers who have been around the block a few times have come up with these solutions independently. Design patterns may offer a few ideas but are of limited use to the lone developer.

1.3. Types of Design Patterns

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than

	instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

2. Creation Design Patterns

It's all about class instantiation in these design patterns. This pattern can be further subdivided into patterns for creating classes and patterns for creating objects. While class-creation patterns make good use of inheritance in the instantiation process, object-creation patterns make good use of delegation.

2.1. Singleton Design Pattern

Definition: Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Solution

All implementations of the Singleton have these two steps in common:

Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

Advantages	Disadvantage
• You can be sure that a class has only a single instance. • You gain a global access point to that instance. • The singleton object is initialized only when it's requested for the first time.	• Violates the *Single Responsibility Principle*. The pattern solves two problems at the time. • The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other. • The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times. • It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern

Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.

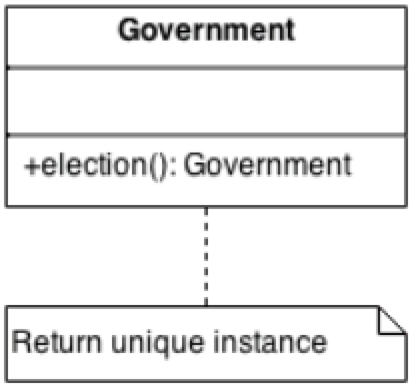


Figure 15 Singleton

2.2. A Factory Pattern

Definition: Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Advantages	**Disadvantages**
• You avoid tight coupling between the creator and the concrete products. • *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support. • *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.	• The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best-case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

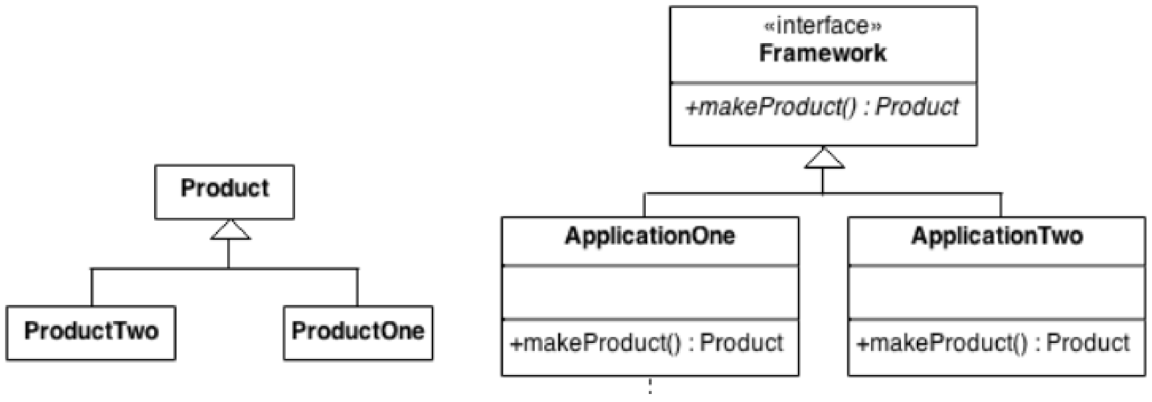


Figure 16 Factory Pattern

2.3. Abstract Factory

Definition: Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Advantages	**Disadvantages**
• You can be sure that the products you're getting from a factory are compatible with each other. • You avoid tight coupling between concrete products and client code. • *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support. • *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.	• The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.

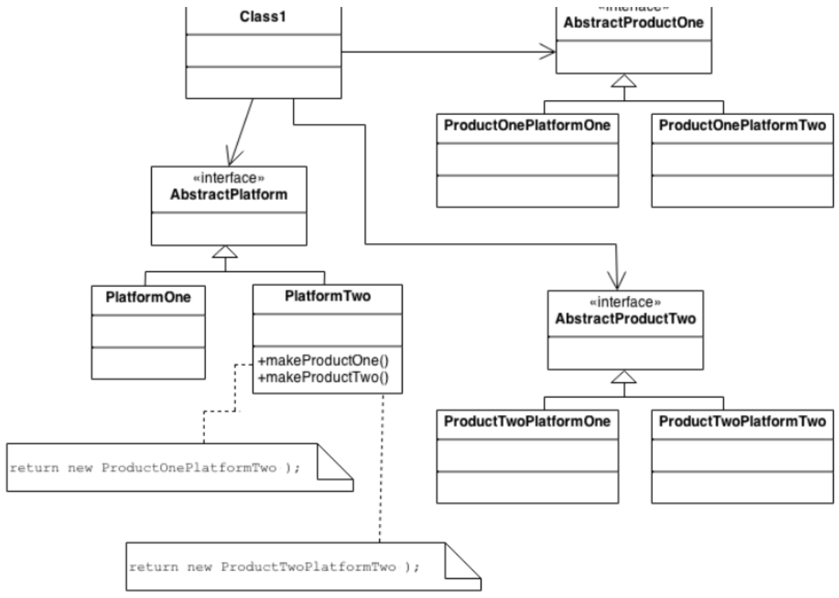


Figure 17 Abstract Pattern

3. Structural Design Patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

3.1. Adapter Pattern

Definition: Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Advantages	Disadvantages
• Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program. • Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface. •	• The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code. •

Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.

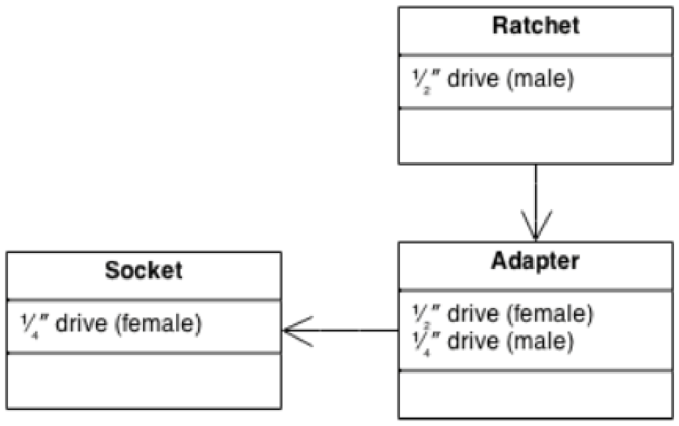


Figure 18 Adapter Pattern

3.2. Bridge Pattern

Definition: Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Advantages	**Disadvantages**
• You can create platform-independent classes and apps. • The client code works with high-level abstractions. It isn't exposed to the platform details. • Open/Closed Principle. You can introduce new abstractions and implementations independently from each other. • Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation. •	• You might make the code more complicated by applying the pattern to a highly cohesive class. •

Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

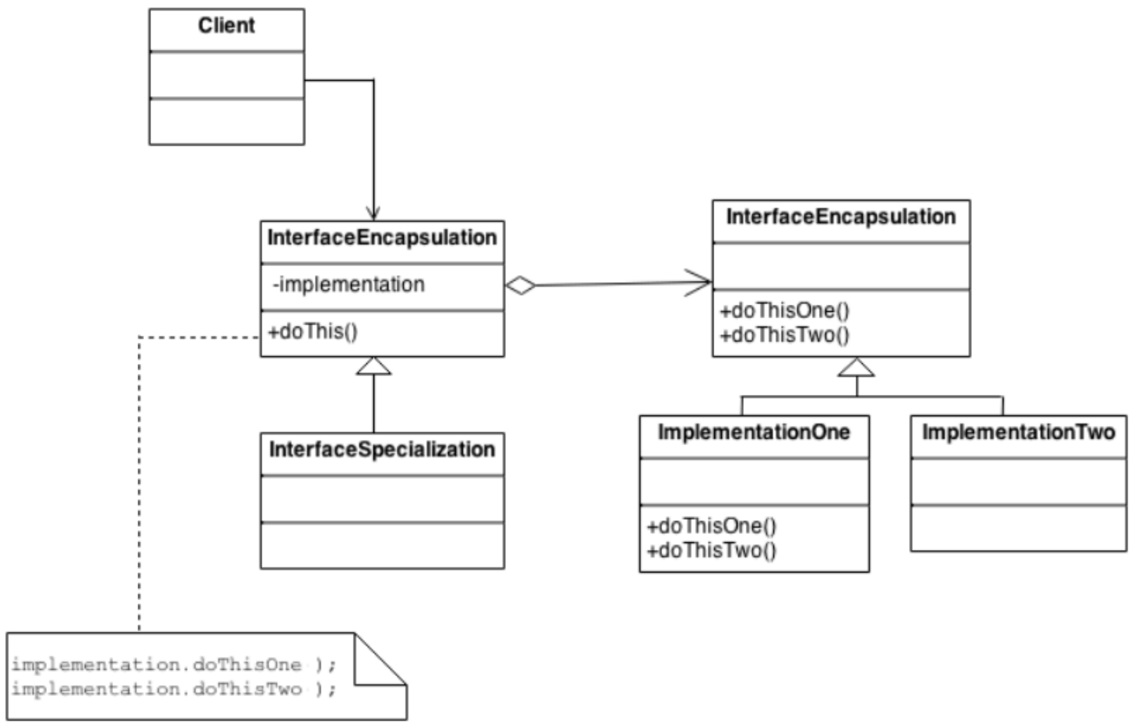


Figure 19 Bridge Pattern

3.3. A Composite Pattern

Definition: Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Advantages	**Disadvantages**
• You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage. • Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree. •	• It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.

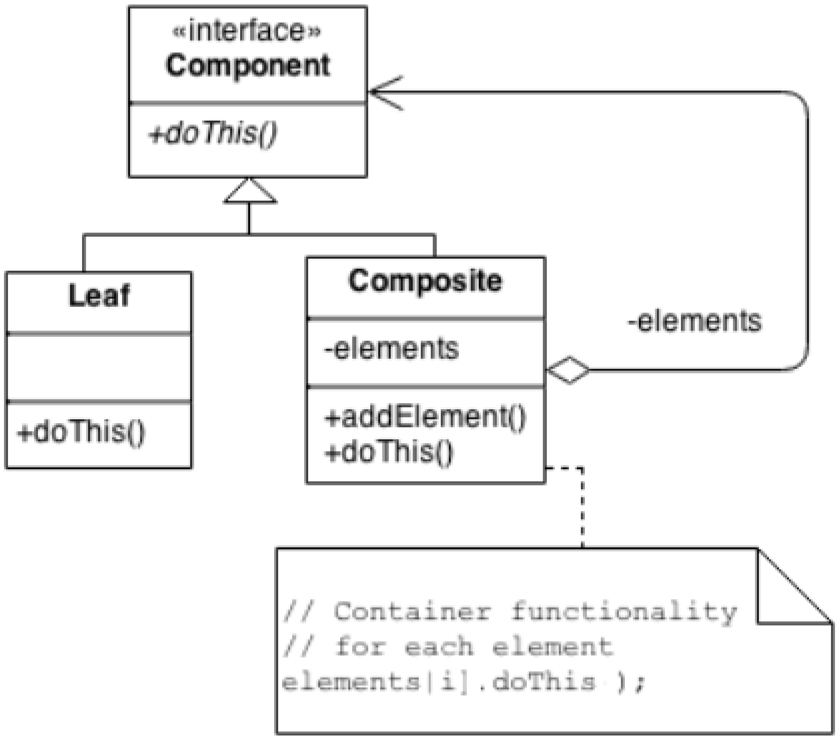


Figure 20 Composite Pattern

3.4. A Decorator Pattern

Definition: Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Advantages	**Disadvantages**
• You can extend an object’s behavior without making a new subclass. • You can add or remove responsibilities from an object at runtime.	• It’s hard to remove a specific wrapper from the wrappers stack. • It’s hard to implement a decorator in such a way that its behavior doesn’t

• You can combine several behaviors by wrapping an object into multiple decorators. • Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes. •	depend on the order in the decorators stack. • The initial configuration code of layers might look pretty ugly. •

Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.
Another example: assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.

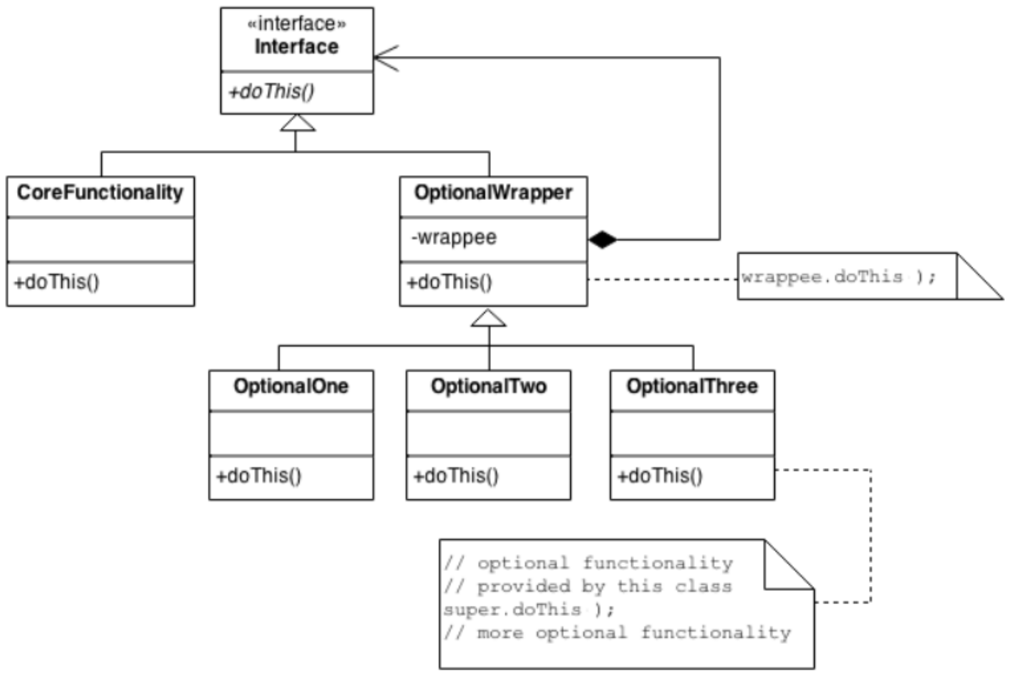


Figure 21 Decorator Pattern

4. Behavioral Design Patterns

4.1. Chain of responsibility

Definition: Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Advantages	**Disadvantages**
• You can control the order of request handling. • Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations. • Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.	• Some requests may end up unhandled.

Example

When a user clicks a button, the event propagates through the chain of GUI elements that starts with the button, goes along its containers (like forms or panels), and ends up with the main application window. The event is processed by the first element in the chain that's capable of handling it. This example is also noteworthy because it shows that a chain can always be extracted from an object tree.

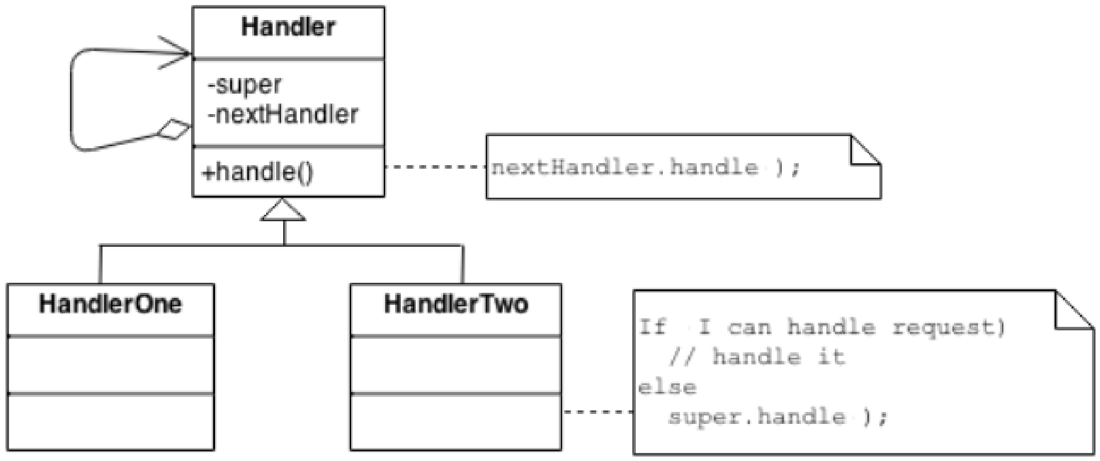


Figure 22 Chain of Repository

4.2. Command Pattern

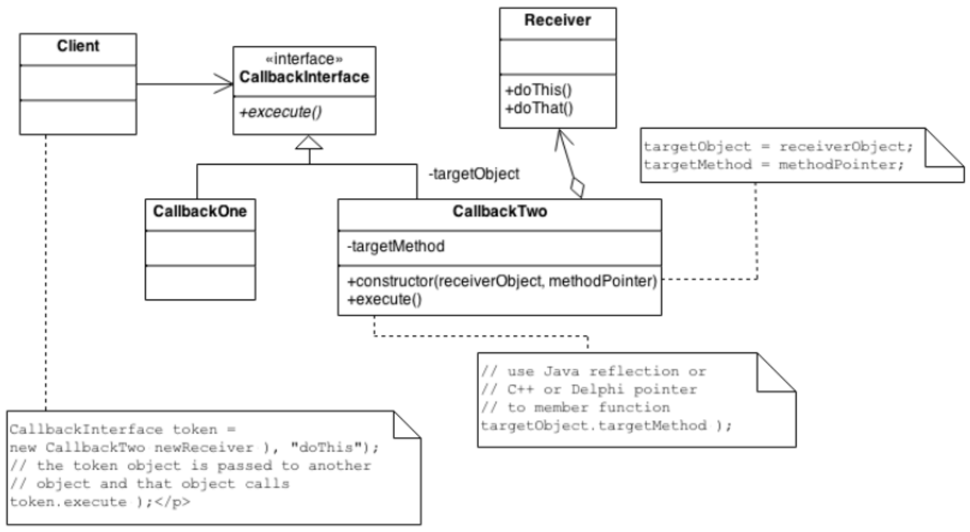
Definition: Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Advantages	**Disadvantages**
• Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations. • Open/Closed Principle. You can introduce new commands into the app without breaking existing client code.	• The code may become more complicated since you're introducing a whole new layer between senders and receivers.

• You can implement undo/redo. • You can implement deferred execution of operations. • You can assemble a set of simple commands into a complex one. •	

Example

The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.



```
targetObject = receiverObject;
targetMethod = methodPointer;
```

```
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod );
```

```
CallbackInterface token =
new CallbackTwo newReceiver ), "doThis");
// the token object is passed to another
// object and that object calls
token.execute );</p>
```

Figure 23 Command Pattern

4.3. Memento Pattern

Definition: Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Advantages	**Disadvantages**

• You can produce snapshots of the object's state without violating its encapsulation. • You can simplify the originator's code by letting the caretaker maintain the history of the originator's state. •	• The app might consume lots of RAM if clients create mementos too often. • Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos. • Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays untouched. •

Example

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.

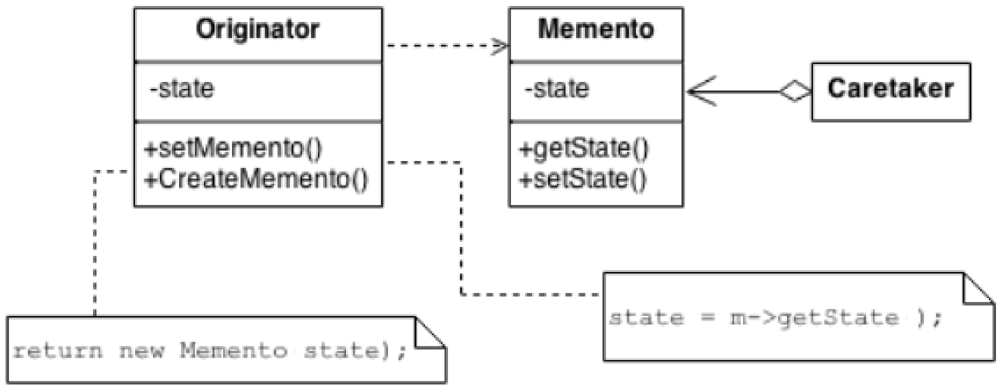


Figure 24 Memento Pattern

4.4. Iterator Pattern

Definition: Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Advantages	**Disadvantages**
• Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes. • Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything. • You can iterate over the same collection in parallel because each iterator object contains its own iteration state. • For the same reason, you can delay an iteration and continue it when needed. •	• Applying the pattern can be an overkill if your app only works with simple collections. • Using an iterator may be less efficient than going through elements of some specialized collections directly. •

Example

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.

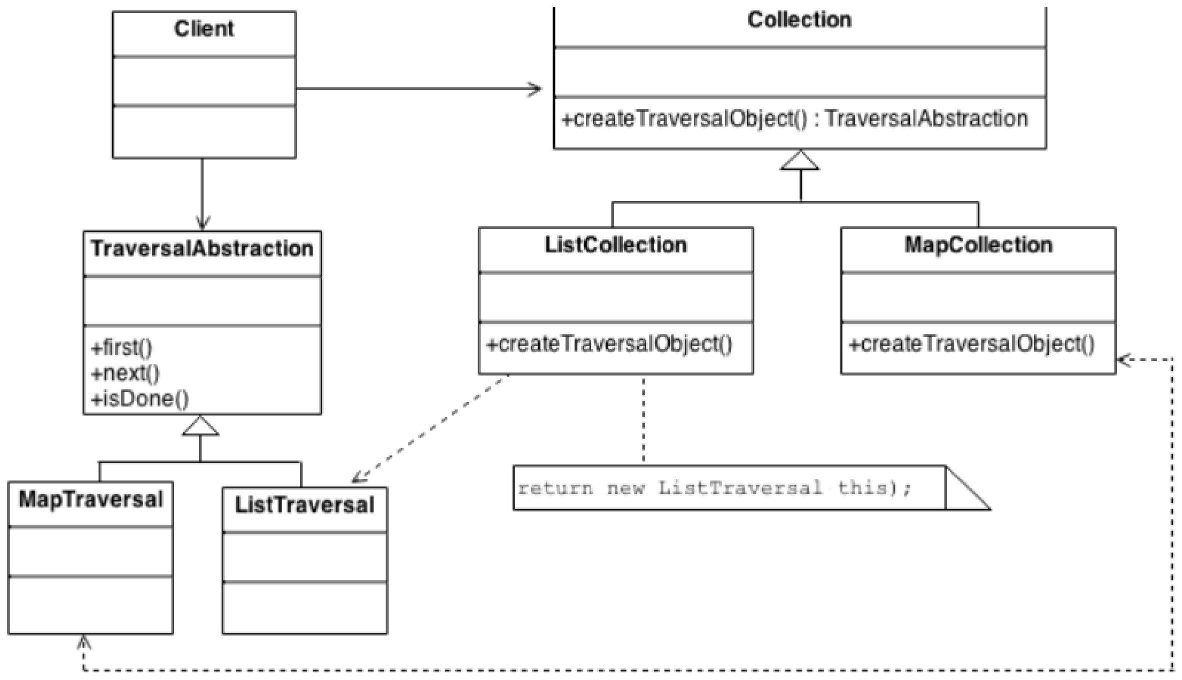


Figure 25 Iterator Pattern

REFERENCES

Tutorialspoint () OOAD - Object Oriented Paradigm [online] Available at: https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.htm [Accessed at: 03 Feb.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: 03 Feb.2022]

Geeksforgeeks (2021) C# | Abstract Classes [online] Available at: https://www.geeksforgeeks.org/c-sharp-abstract-classes/ [Accessed at: 03 Feb.2022]

Jignest, T (2019) Association, Aggregation and Composition [online] Available at: https://www.c-sharpcorner.com/UploadFile/ff2f08/association-aggregation-and-composition/ [Accessed at: 03 Feb.2022]

Dotnettutorials () Dependency Injection in C# [online] Available at: https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/ [Accessed at: 03 Feb.2022]

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: 03 Feb.2022]

Tutorialspoint () Object-Oriented Paradigm [online] Available at: https://www.tutorialspoint.com/software_architecture_design/object_oriented_paradigm.htm [Accessed at: 03 Feb.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: 03 Feb.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: 03 Feb.2022]

Koderhq () C# Composition Tutorial [online] Available at: https://www.koderhq.com/tutorial/csharp/oop-composition/ [Accessed at: 03 Feb.2022]