ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	July 3, 2021	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Tien Hoc	**Student ID**	GCH190844
Class	GCH0705	**Assessor name**	Doan Trung Tung
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	Hoc

Grading grid

P3	P4	M3	M4	D3	D4

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

Contents

1 Introduction 5
2 Scenario analysis 5
2.1 Scenario 5
2.2 Diagram 5
3 Implementation 6
3.1 Code 6
3.2 Program screenshots 13
4 Discussion 15
4.1 Range of similar patterns 15
4.2 Usage of pattern 16

Figure 1: Class Diagram 5
Figure 2: Class StudentComponent 6
Figure 3: Class Student 6
Figure 4: Class ITClub 7
Figure 5: IoTClub and Mobile Club 8
Figure 6: Class MenuProgram 8
Figure 7: Class CompositeSystem 9
Figure 8: Option 3 13
Figure 9: Option 1 13
Figure 10: Option 2 14
Figure 11: Option 4 14
Figure 12: Result of option 4 14

1 Introduction

My team has shown the efficient of UML diagrams in OOAD and introduction of some Design Patterns in usages. My tasks in this stage are giving a demonstration of using OOAD and Design Pattern in a small problem, as well as advanced discussion of range of design patterns.

2 Scenario analysis

2.1 Scenario

This is the program to manage students and clubs. Students have their own personal information and club schedule. Clubs are student groups, and administrators can view information about club members or the gathering schedule of clubs in the university. Collectively, the program will have functions including creating new students, viewing the focused calendar of any club or student, and adding or removing that student from the club.
I use the Composite Pattern because manipulating a club compared to manipulating a student has many similarities.

2.2 Diagram

Diagram:

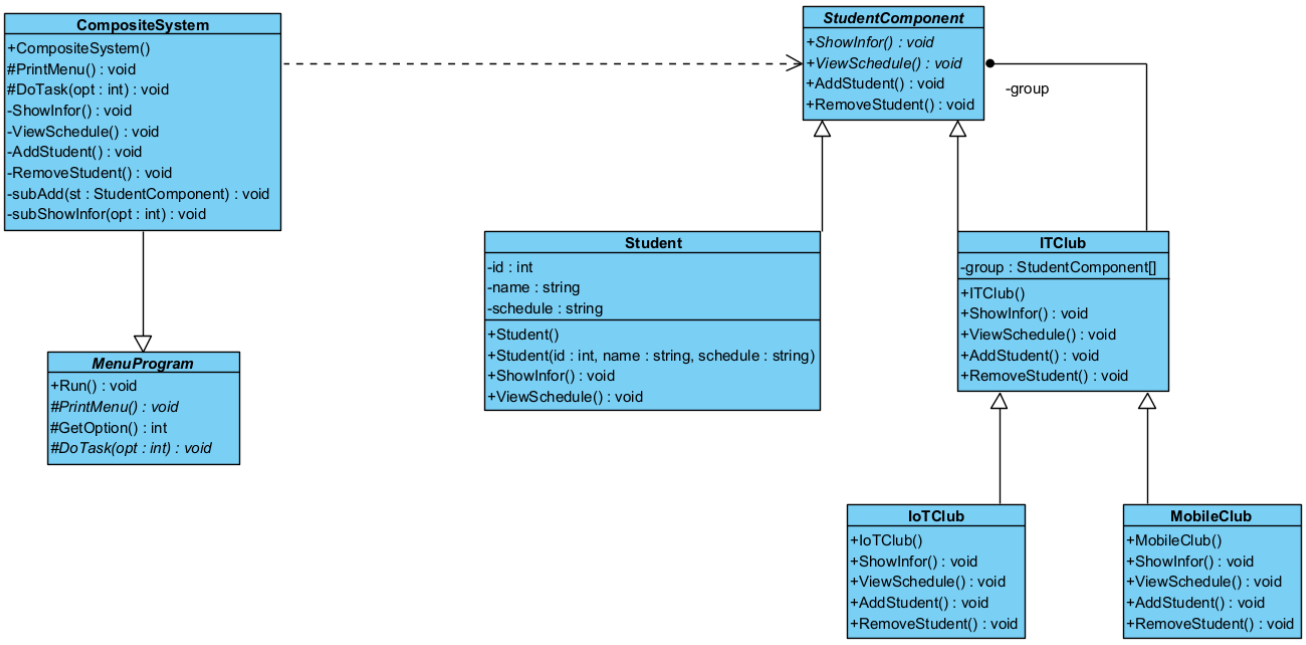


Figure 1: Class Diagram

Explanation class diagram:

StudentComponent class is an abstract class with 2 abstract methods and 2 virtual methods. This class is used to declare the interface for the objects in the component. And there are methods that implement the default behavior for the interface common to all classes. The other 2 classes that inherit from StudentComponent are the Student class and the ITClub class. The Student class defines the behavior that can be handled on a Student object. And the ITClub class stores Student objects through the StudentComponent class. The ITClub class implements Student related operations in the StudentComponent interface. The relationship between ITClub and StudentComponent is aggregation because the ITClub class defines a collection of StudentComponent objects. The IoTClub class and MobileClub class inherit from ITClub and use a collection of student groups through the StudentComponent. Finally, the administrator will manipulate the objects in the component through the StudentComponent class.

In order to encapsulate the operations and make the program easy to use, I created the CompositeSystem class. The CompositeSystem class inherits from MenuProgram. MenuProgram is an abstract class that has functions for the purpose of performing a task in a list of tasks. CompositeSystem is a menu for users to manipulate more easily in management.

3 Implementation

3.1 Code

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4
5   namespace CompositePattern
6   {
        21 references
7       abstract class StudentComponent
8       {
            11 references
9           public abstract void ShowInfor();
            10 references
10          public abstract void ViewSchedule();
11
            8 references
12          public virtual void AddStudent(StudentComponent st) { }
            8 references
13          public virtual void RemoveStudent(StudentComponent st) { }
14      }
15  }
16
```

Figure 2: Class StudentComponent

Class StudentComponent is an abstract class with 2 abstract methods and 2 virtual methods

```
7       class Student : StudentComponent
8       {
            4 references
9           public int ID { get; set; }
            4 references
10          public string Name { get; set; }
            3 references
11          public string Schedule { get; set; }
            0 references
12          public Student()
13          {
14
15          }
            1 reference
16          public Student(int id, string name, string schedule)
17          {
18              ID = id;
19              Name = name;
20              Schedule = schedule;
21          }
            11 references
22          public override void ShowInfor()
23          {
24              Console.WriteLine("ID: " + ID + " - Name: " + Name + " - Schedule: " + Schedule);
25          }
26
            10 references
27          public override void ViewSchedule()
28          {
29              Console.WriteLine("Schedule of " + Name + ": " + Schedule);
30          }
31      }
```

Figure 3: Class Student

Class Student inherits from StudentComponent and obliges to override 2 abstract methods. In the Student Class, there are two constructor methods, one with parameters and one without.

```
7   class ITClub : StudentComponent
8   {
9       private List<StudentComponent> group;
        1 reference
10      public ITClub()
11      {
12          group = new List<StudentComponent>();
13      }
14
        11 references
15      public override void ShowInfor()
16      {
17          foreach (StudentComponent st in group)
18          {
19              st.ShowInfor();
20          }
21      }
22
        10 references
23      public override void ViewSchedule()
24      {
25          foreach (StudentComponent st in group)
26          {
27              st.ViewSchedule();
28          }
29      }
30
        8 references
31      public override void AddStudent(StudentComponent st)
32      {
33          group.Add(st);
34      }
35
        8 references
36      public override void RemoveStudent(StudentComponent st)
37      {
38          group.Remove(st);
39      }
40  }
```

Figure 4: Class ITClub

Class ITClub inherits from StudentComponent and overrides all four methods of the base class. In the ITClub class there is a constructor method with no parameters, in this constructor there is a group initialization - this is a list of StudentComponent objects.
Overridden methods add or remove objects from the group and view information or focus schedules of the objects in the group.

```
class IoTClub : ITClub
{
    private List<StudentComponent> group;

    1 reference
    public IoTClub()
    {
        group = new List<StudentComponent>();
    }

    11 references
    public override void ShowInfor()
    {
        base.ShowInfor();
    }

    10 references
    public override void ViewSchedule()
    {
        base.ViewSchedule();
    }

    8 references
    public override void AddStudent(StudentComponent st)
    {
        base.AddStudent(st);
    }

    8 references
    public override void RemoveStudent(StudentComponent st)
    {
        base.RemoveStudent(st);
    }
}
```

```
class MobileClub : ITClub
{
    private List<StudentComponent> group;

    1 reference
    public MobileClub()
    {
        group = new List<StudentComponent>();
    }

    11 references
    public override void ShowInfor()
    {
        base.ShowInfor();
    }

    10 references
    public override void ViewSchedule()
    {
        base.ViewSchedule();
    }

    8 references
    public override void AddStudent(StudentComponent st)
    {
        base.AddStudent(st);
    }

    8 references
    public override void RemoveStudent(StudentComponent st)
    {
        base.RemoveStudent(st);
    }
}
```

Figure 5: IoTClub and Mobile Club

The IoTClub and MobileClub classes both inherit from ITClub and have similar functions and tasks.

```
public abstract class MenuProgram
{
    1 reference
    public void Run()
    {
        bool running = true;
        while (running)
        {
            PrintMenu();
            int opt = GetOption();
            DoTask(opt);
            if (opt == 0) running = false;
        }
    }

    2 references
    protected abstract void DoTask(int opt);
    2 references
    protected abstract void PrintMenu();

    1 reference
    protected int GetOption()
    {
        Console.Write("Enter your option: ");
        int opt = Convert.ToInt32(Console.ReadLine());
        return opt;
    }
}
```

Figure 6: Class MenuProgram

The MenuProgram class is an abstract function with two abstract methods, DoTask and PrintMenu. DoTask takes an integer as an input parameter.

There are two other methods, Run and GetOption. GetOption has an integer return type, and this integer is entered by the user. The meaning of the Run function is to run a loop. This loop prints the menu of PrintMenu(), then does the job of the DoTask function by receiving the command from the GetOption function. The loop will stop when the user enters 0.

```
7   class CompositeSystem : MenuProgram
8   {
9       private List<Student> group;
10      private StudentComponent it;
11      private StudentComponent iot;
12      private StudentComponent mobile;
13
        1 reference
14      public CompositeSystem()
15      {
16          group = new List<Student>();
17          it = new ITClub();
18          iot = new IoTClub();
19          mobile = new MobileClub();
20      }
        2 references
21      protected override void DoTask(int opt)
22      {
23          switch (opt)
24          {
25              case 1: ShowInfor(); break;
26              case 2: ViewSchedule(); break;
27              case 3: AddStudent(); break;
28              case 4: RemoveStudent(); break;
29              default:
30                  Console.WriteLine("Invalid option");
31                  break;
32          }
33      }
34
```

Figure 7: Class CompositeSystem

The CompositeSystem class is the PrintMenu and DoTask implementation of MenuProgram, so this class inherits class MenuProgram.

```
35          private void RemoveStudent()
36          {
37              Console.Write("Enter ID: ");
38              int id = Convert.ToInt32(Console.ReadLine());
39              bool found = false;
40
41              foreach (Student st in group)
42              {
43                  if (st.ID == id)
44                  {
45                      it.RemoveStudent(st);
46                      iot.RemoveStudent(st);
47                      mobile.RemoveStudent(st);
48                      Console.WriteLine(st.Name + " has been removed");
49                      found = true;
50                  }
51              }
52              if (!found)
53              {
54                  Console.WriteLine("No student with ID: " + id);
55              }
56          }
57
```

The RemoveStudent function requires the user to enter the student ID. Then use foreach to browse all students, if found, will remove students from the group and if not found, will notify.

```
58          private void AddStudent()
59          {
60              string n = "yes";
61              while (n == "yes")
62              {
63                  Console.Write("Enter ID: ");
64                  int id = Convert.ToInt32(Console.ReadLine());
65                  Console.Write("Enter Name: ");
66                  string name = Console.ReadLine();
67                  Console.Write("Enter Schedule: ");
68                  string schedule = Console.ReadLine();
69
70                  Student st = new Student(id, name, schedule);
71                  subAdd(st);
72                  group.Add(st);
73
74                  Console.Write("Add more?\nEnter your choice: ");
75                  n = Console.ReadLine();
76              }
```

AddStudent function to add students, using while loop. The user will have to enter the information and select the group to add. After adding, the user enters 'yes' if he wants to add another student.

```
93          private void ViewSchedule()
94          {
95              Console.WriteLine("1. IT Club");
96              Console.WriteLine("2. IoT Club");
97              Console.WriteLine("3. Mobile Club");
98              Console.Write("Enter option: ");
99              int opt = Convert.ToInt32(Console.ReadLine());
100             switch (opt)
101             {
102                 case 1: it.ViewSchedule(); break;
103                 case 2: iot.ViewSchedule(); break;
104                 case 3: mobile.ViewSchedule(); break;
105                 default:
106                     Console.WriteLine("Invalid option");
107                     break;
108             }
109         }
```

The ViewSchedule function is used to view the calendars of the groups. The user will enter the natural number corresponding to which group the user wants to see.

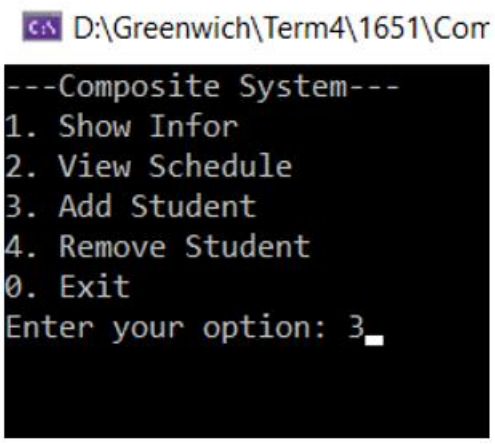
```
111         private void ShowInfor()
112         {
113             Console.WriteLine("1. Show Student / 2. Show Club");
114             Console.Write("Enter your choice: ");
115             int choice = Convert.ToInt32(Console.ReadLine());
116             if (choice == 1)
117             {
118                 Console.Write("Enter ID: ");
119                 int id = Convert.ToInt32(Console.ReadLine());
120                 bool found = false;
121
122                 foreach (Student st in group)
123                 {
124                     if (st.ID == id)
125                     {
126                         st.ShowInfor();
127                         found = true;
128                     }
129                 }
130                 if (!found)
131                 {
132                     Console.WriteLine("No student with ID: " + id);
133                 }
134             }
135             else if (choice == 2)
136             {
137                 Console.WriteLine("1. IT Club");
138                 Console.WriteLine("2. IoT Club");
139                 Console.WriteLine("3. Mobile Club");
140                 Console.Write("Enter your choice: ");
141                 int opt = Convert.ToInt32(Console.ReadLine());
142                 subShowInfor(opt);
143             }
144             else
145             {
146                 Console.WriteLine("Invalid option");
147             }
148         }
```

ShowInfor uses if/elseif conditional statements to handle whether the user wants to see the information of a group or a student.

```
150         protected override void PrintMenu()
151         {
152             Console.WriteLine("---Composite System---");
153             Console.WriteLine("1. Show Infor");
154             Console.WriteLine("2. View Schedule");
155             Console.WriteLine("3. Add Student");
156             Console.WriteLine("4. Remove Student");
157             Console.WriteLine("0. Exit");
158         }
159
            1 reference
160         private void subShowInfor(int opt)
161         {
162             if (opt == 1) { it.ShowInfor(); }
163             else if (opt == 2) { iot.ShowInfor(); }
164             else if (opt == 3) { mobile.ShowInfor(); }
165             else { Console.WriteLine("Invalid option"); }
166         }
167
            1 reference
168         private void subAdd(Student st)
169         {
170             Console.WriteLine("Add Student to");
171             Console.WriteLine("1. IT Club");
172             Console.WriteLine("2. IoT Club");
173             Console.WriteLine("3. Mobile Club");
174             Console.Write("Enter option: ");
175             int opt = Convert.ToInt32(Console.ReadLine());
176             switch (opt)
177             {
178                 case 1: it.AddStudent(st); break;
179                 case 2: iot.AddStudent(st); break;
180                 case 3: mobile.AddStudent(st); break;
181                 default:
182                     Console.WriteLine("Invalid option");
183                     break;
184             }
185         }
```

PrintMenu function to print out the menu, this menu is a summary of the program's functions. The two sub functions are small functions that support AddStudent and ShowInfor.

3.2 Program screenshots



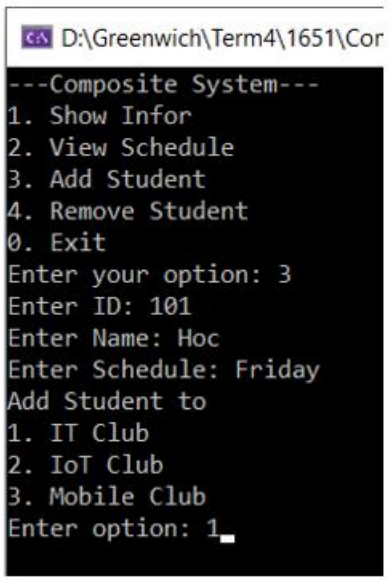
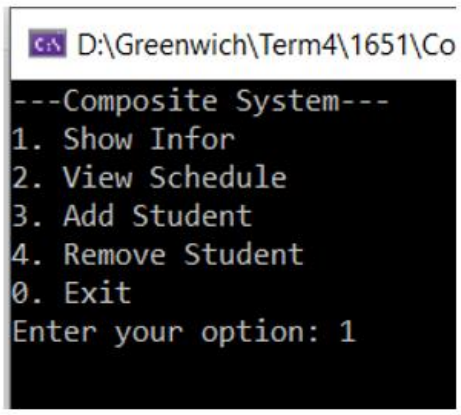


Figure 8: Option 3



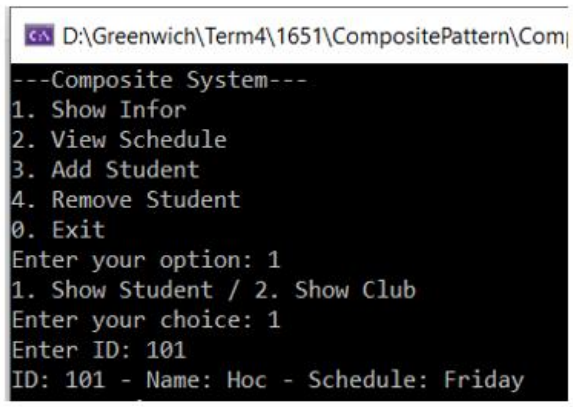


Figure 9: Option 1

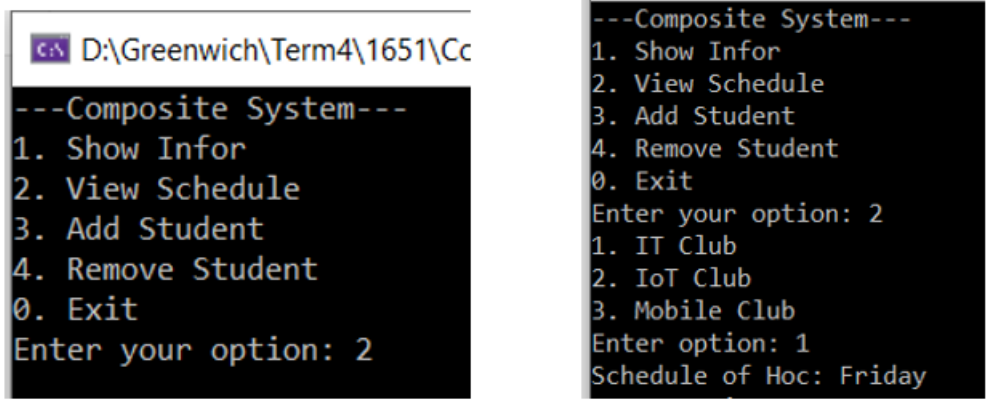


Figure 10: Option 2

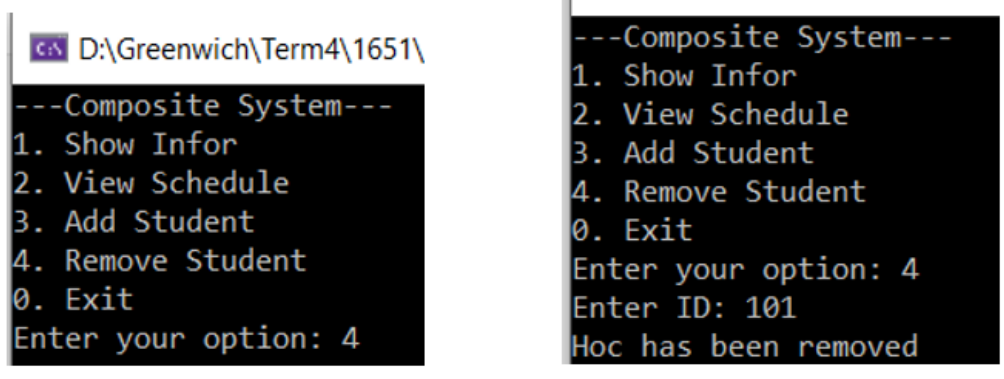


Figure 11: Option 4

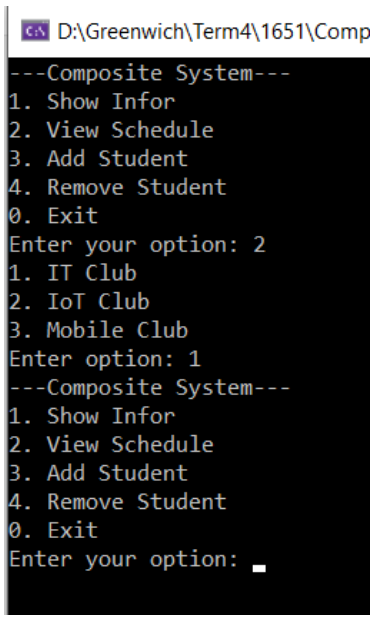


Figure 12: Result of option 4

4 Discussion

4.1 Range of similar patterns

Composite pattern has structure diagrams rely on recursive composition to organize a number of objects, and has two other similar patterns, Decorator and Proxy.

Decorator Pattern

The decorator pattern is a design pattern in object-oriented programming that allows behavior to be dynamically added to an individual object without impacting the behavior of other objects in the same class. The decorator technique is frequently used to adhere to the Single Responsibility Principle since it allows functionality to be partitioned into classes with distinct concerns. Because an object's functionality can be enhanced without defining a totally new object, using decorator can be more efficient than sub classing.

More flexibility than static inheritance. The Decorator pattern allows you to assign responsibilities to objects in a more flexible fashion than static (multiple) inheritance allows. Decorators allow you to add and remove responsibilities at runtime by simply attaching and removing them. Inheritance, on the other hand, necessitates the creation of a new class for each extra responsibility. This results in a large number of classes and raises the system's complexity. Additionally, having multiple Decorator classes for a single Component class allows you to mix and match responsibilities.

Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects. The Decorator allows you to layer your business logic, define a decorator for each layer, and construct objects at runtime using various combinations of this logic. Because all of these objects have the same interface, the client programs may treat them all the same.

Use the pattern when it’s awkward or not possible to extend an object’s behavior using inheritance. Many programming languages have a keyword that can be used to restrict a class from being extended further. The only method to reuse existing behavior in a final class would be to wrap it in your own wrapper using the Decorator approach.

Proxy Pattern

The proxy pattern is a software design paradigm in computer programming. In its most basic form, a proxy is a class that acts as an interface to something else. A network connection, a huge object in memory, a file, or some other resource that is expensive or impossible to recreate could all be interfaced by the proxy. In a nutshell, a proxy is a wrapper or agent object that the client uses to gain access to the real serving object behind the scenes. The proxy can be used to simply forward data to the real object or to offer additional functionality. Extra functionality, such as caching when actions on the real object are resource heavy or validating preconditions before operations on the real object are called, can be supplied in the proxy. Because both implement the same interface, using a proxy object is identical to using the real object for the client.

According to the Proxy pattern, you should create a new proxy class with the same interface as the original service object. Then you update your app to send the proxy object to all clients of the original object. The proxy constructs an actual service object and delegate all work to it when it receives a request from a client.
If you need to run something before or after the class's primary logic, the proxy allows you to do so without modifying the class. The proxy can be supplied to any client that expects a real service object because it implements the same interface as the original class.

Applicability:

- Virtual Proxy. This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
- Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
- Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.

- Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.
- Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.

Composite Pattern

The composite pattern is a partitioning design pattern in software engineering. The composite pattern refers to a collection of objects that are considered in the same way as a single instance of the same type. A composite's goal is to "compose" things into tree structures in order to describe part-whole hierarchies. Clients can use the composite pattern to treat individual objects and compositions consistently. The composite pattern refers to a collection of objects that are considered in the same way as a single instance of the same type. A composite's goal is to "compose" things into tree structures in order to describe part-whole hierarchies. Clients can use the composite pattern to treat individual objects and compositions consistently.

Use the Composite pattern when you have to implement a tree-like object structure. Simple leaves and complicated containers are two basic element kinds that have a common interface in the Composite pattern. Both leaves and other containers can be used to make a container. This allows you to create a tree-like nested recursive object structure.

Use the pattern when you want the client code to treat both simple and complex elements uniformly. The Composite design defines a single interface for all of its pieces. The client does not have to care about the concrete class of the objects it works with while using this interface.

There are several reasons for me to use Composite pattern instead of Decorator. A Decorator is like a Composite but only has one child component. It doesn't make sense to remove a specific wrapper from the wrapper stack.

4.2 Usage of pattern

Based on the above explanations, it can be seen that the Composite pattern is the right pattern for the problem of the project. Although it has advantages and disadvantages, Composite has addressed the most of the necessary needs. The following is a list of Composite's pros and cons in solving the problem:

Advantages:

- The pattern helps to achieve uniformity (use of similar functions) across the object hierarchy that contains primitive as well as composite object types.
- The pattern makes it easier to add new kinds of components.
- The pattern makes it easier for the client to achieve the desired functionality without worrying about what kind of object is it dealing with.

Disadvantages:

- The composite pattern can become too general sometimes because of its uniformity, as for example, it is difficult to restrict objects that can be included in the composite group.
- The client must be able to distinguish between composite and non-composite objects as the composite class is mostly extended to provide access to its individual group members in the hierarchy.