UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

Higher Nationals in Computing

UNIT 20: ADVANCED PROGRAMMING

ASSIGNMENT 1

Learner's name: Trinh Thi Dieu Huyen

ID: GDD18606

Class: GCS0801B.1

Subject code: 1651

Assessor name: **PHAN MINH TAM**

Assignment due:

Assignment submitted:

ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	**Unit 20: Advanced Programming**		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Trinh Thi Dieu Huyen	**Student ID**	GDD18606
Class	1651 GCS0801B.1	**Assessor name**	Phan Minh Tam
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.			
		Student's signature	huyen

Grading grid

P1	P2	M1	M2	D1	D2

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

ASSIGNMENT 1 BRIEF

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

Submission Format:
Format: The submission is in the form of a **group written report and presentation**. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system. *Submission:* Students are compulsory to submit the assignment in due date and in a way requested by the Tutors. The form of submission will be a soft copy in PDF posted on corresponding course of http://cms.greenwich.edu.vn/ *Note:* The Assignment *must* be your own work, and not copied by or from another student or from books etc. If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. Make sure that you know how to reference properly, and that understand the guidelines on plagiarism. *If you do not, you definitely get fail*
Assignment Brief and Guidance:
Scenario: You have recently joined a software development company to help improve their documentation of their in-houses software libraries which were developed with very poor documentation. As a result, it has been very difficult for the company to utilise their code in multiple

projects due to poor documentation. Your role is to alleviate this situation by showing the efficient of UML diagrams in OOAD and Design Patterns in usages.

Tasks

You and your team need to explain characteristics of Object-oriented programming paradigm by applying Object-oriented analysis and design on a given (assumed) scenario. The scenario can be small but should be able to presents various characteristics of OOP (such as: encapsulation, inheritance, polymorphism, override, overload, etc.).

The second task is to introduce some design patterns (including 3 types: creational, structural and behavioral) to audience by giving real case scenarios, corresponding patterns illustrated by UML class diagrams.

To summarize, you should analyze the relationship between the object-orientated paradigm and design patterns.

The presentation should be about approximately 20-30 minutes and it should be summarized of the team report.

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

Table of Contents

ASSIGNMENT 1 ANSWERS 1

LO1 Examine the key components related to the object-orientated programming paradigm, analyzing design pattern types 1

P1 Examine the characteristics of the object-orientated paradigm as well as the various class relationships 1

1. Outline the object-oriented paradigm as well as the various class relationships 1

1.1. Object/Class 1

1.2. Abstraction 2

1.3. Encapsulation 3

1.4. Inheritance 4

1.5. Polymorphism 5

2. Object-oriented class relationships 6

M1 Determine a design pattern from each of the creational, structural, and behavioral pattern types 7

1. Creational Patterns: Singleton 7

2. Structural patterns: Decorator 8

3. Behavioral patterns: Iterator 10

LO2 Design a series of UML class diagrams 10

P2 Design and build class diagrams using a UML tool 10

1. Our scenario detail 10

2. Use-case diagrams 11

3. Class diagrams 11

4. Pseudo-codes 12

5. Activity diagrams 14

M2 Define class diagrams for specific design patterns using a UML tool 15

1. Class diagram for all type in Creational Patterns 15

2. Class diagram for all type in Structural Patterns 24

3. Class diagram for all type in Behavioral Patterns 35

REFERENCES 57

Table of Figures

Figure 1: Class and Object 1

Figure 2: Abstraction 2

Figure 3: Encapsulation 3

Figure 4: Inheritance 4

Figure 5: Polymorphism 5

Figure 6: Object-oriented class relationships 6

Figure 7: Structure of Singleton pattern 8

Figure 8: Structure of Decorator pattern 9

Figure 9: Structure of Iterator pattern 10

Figure 10: Use-case diagrams for restaurant management system 11

Figure 11: Class diagrams for restaurant management system 12

Figure 12: Activity diagram for restaurant management system 14

Figure 13: Activity diagram of Staff 15

Figure 14: Activity diagram of Admin 15

Figure 15: Class diagram for Singleton Pattern 16

Figure 16: Class diagram for Prototype Pattern 17

Figure 17: Class diagram for Builder Pattern 18

Figure 18: Class diagram for Factory Method Pattern 20

Figure 19: Class diagram for Abstract Factory Pattern 22

Figure 20: Class diagram for Proxy Pattern 24

Figure 21: Class diagram for Decorator Pattern 25

Figure 22: Class diagram for Adapter Pattern 26

Figure 23: Class diagram for Facade Pattern 27

Figure 24: Class diagram for Flyweight Pattern 29

Figure 25: Class diagram for Composite Pattern 31

Figure 26: Class diagram for Bridge Pattern 33

Figure 27: Class diagram for Visitor Pattern 35

Figure 28: Class diagram for Observer Pattern 36

Figure 29: Class diagram for Strategy (Policy) Pattern 38

Figure 30: Class diagram for Template Method Pattern 40

Figure 31: Class diagram for Command Pattern 41

Figure 32: Class diagram for Iterator Pattern 43

Figure 33: Class diagram for Memento Pattern 45

Figure 34: Class diagram for State Pattern 47

Figure 35: Class diagram for Mediator Pattern 50

Figure 36:Class diagram for Chain of Responsibility Pattern 52

Figure 37: Class diagram for Interpreter Pattern 54

Table of Tables

No table of figures entries found.

ASSIGNMENT 1 ANSWERS

LO1 Examine the key components related to the object-orientated programming paradigm, analyzing design pattern types

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

1. Outline the object-oriented paradigm as well as the various class relationships

Object-Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic. OOP allows the decomposition of a problem into a number of entities called objects, then builds data and functions around these objects.

1.1. Object/Class

Class and Object are the basic concepts of Object-Oriented Programming which revolve around real-life entities. The class is the blueprint or prototype that contains variables for storing data and functions to perform operations on the data user-defined from which objects are created. Basically, the class combines the fields and methods (member function which defines actions) into a single unit. The classes support polymorphism, inheritance, and also provide the concept of derived classes and base classes. Objects are the software bundle of related variables and methods or an instance of a class. It may represent a person, a place, or any item that the program must handle.

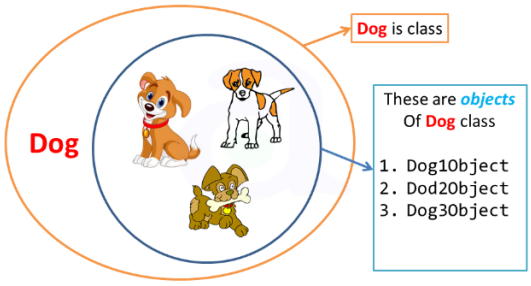

Figure 1: Class and Object

Example code:

```
// C# program to illustrate the initialization of an object
using System;
// Class Declaration
public class Dog {
   // Instance Variables
   String name; String breed; int age; String color;
```

```
    // Constructor Declaration of Class
    public Dog(String name, String breed, int age, String color)
    { this.name = name; this.breed = breed; this.age = age; this.color = color; }
    public String getName() { return name; } // Property 1
    public String getBreed() { return breed; } // Property 2
    public int getAge() { return age; } // Property 3
    public String getColor() { return color; } // Property 4
    public String toString() // Method 1
    {
        return ("Name: " + this.getName() + "\nBreed: " + this.getBreed()
               + "\nAge: " + this.getAge() + "\nColor: " + this.getColor());
    }
 public static void Main(String[] args) // Main Method
    {
        Dog lucky = new Dog("lucky", "husky", 5, "black"); // Creating object
        Console.WriteLine(lucky.toString());
    }
}
```

1.2. Abstraction

Abstraction is the process of modeling only relevant features. Means hide unnecessary details that are irrelevant for current purpose (and/or user). It also reduces complexity and aids understanding. Abstraction provides the freedom to defer implementation decisions by avoiding commitments to details. Therefore, the properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

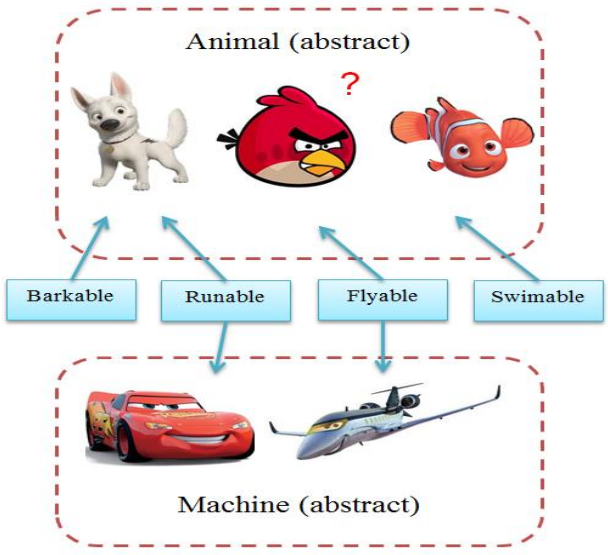

Figure 2: Abstraction

Example code:

```
// C# program to calculate the area of a square using the concept of data abstraction
using System;
namespace Demoabstraction {
abstract class Shape { // Abstract class
   public abstract int area(); // Abstract method
}
// Square class inheriting from the Shape class
class Square : Shape {
   private int side; // private data member
   public Square(int x = 0) { side = x; } // method of  square class
   // Overriding of the abstract method of Shape class using the override keyword
   public override int area()
   {
       Console.Write("Area of Square: "); return (side * side);
   }
}
class GFG {
   static void Main(string[] args)
   {   // Creating the reference of Shape class, which refer to Square class instance
       Shape sh = new Square(4);
       double result = sh.area(); // Calling the method
       Console.Write("{0}", result);
    }
  }
}
```

1.3. Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. On the other hand, encapsulation also is a protective shield that prevents the data from being accessed by the code outside this shield. Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class in which it is declared.

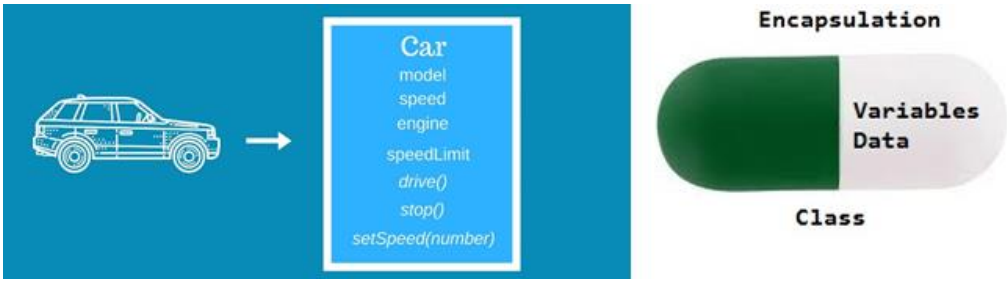

Figure 3: Encapsulation

Example code:

```
// C# program to illustrate encapsulation
using System;
public class DemoEncap {
  // private variables declared these can only be accessed by public methods of class
  private String studentName;
  private int studentAge;
  // using accessors to get and set the value of studentName
  public String Name { get {return studentName;} set {studentName = value;} }
  // using accessors to get and set the value of studentAge
  public int Age { get {return studentAge;}  set {studentAge = value;} }
}
class GFG {
  static public void Main()
  {
        DemoEncap obj = new DemoEncap(); // creating object
        // calls set accessor of the property Name, and pass "Ankita" as value of the
standard field 'value'
        obj.Name = "Ankita";
        // calls set accessor of the property Age, and pass "21" as value of the stand
ard field 'value'
        obj.Age = 21;
        // Displaying values of the variables
        Console.WriteLine("Name: " + obj.Name);
        Console.WriteLine("Age: " + obj.Age);
  }
}
```

1.4. Inheritance

Inheritance is one class allowed to inherit the features (fields and methods) of another class. In other words, it is a process of object reusability. On the other hand, inheritance is also a feature of the object-oriented language used to represent specialization/generalization relationships between classes.

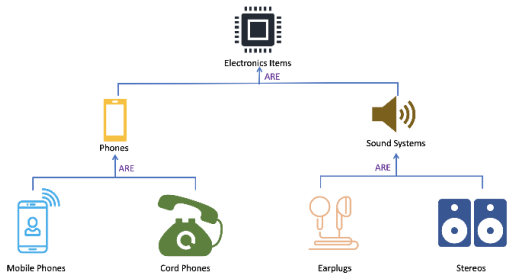

Figure 4: Inheritance

Example code:

```
// C# program to illustrate the concept of inheritance
using System;
namespace ConsoleApplication1 {
class GFG { // Base class
    // data members
    public string name;
    public string subject;
    public void readers(string name, string subject) // public method of base class
    {
      this.name = name; this.subject = subject;
      Console.WriteLine("Myself: " + name);
      Console.WriteLine("My Favorite Subject is: " + subject);
    }
}
class GeeksforGeeks : GFG { // inheriting the GFG class using :
   public GeeksforGeeks() // constructor of derived class
    { Console.WriteLine("GeeksforGeeks"); }
}
class Sudo {
   static void Main(string[] args)
    {
      GeeksforGeeks g = new GeeksforGeeks(); // creating object of derived class
      g.readers("Jay", "C#"); // calling the method of base class, using the derived class
object
    }
  }
}
```

1.5. Polymorphism

Polymorphism is the ability of an entity to behave differently in different situations (existing in multiple forms). Mean a phenomenon where objects of different classes can understand the same message in different ways. In other words, many forms of a single object are called Polymorphism. To express polymorphism, classes must have inheritance relationships with some same superclass. In addition, the polymorphism method must be overridden in subclasses.

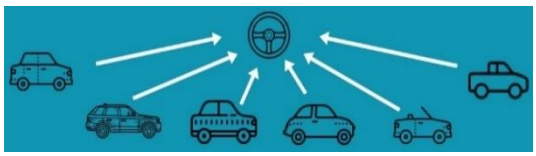

Figure 5: Polymorphism

Example code:

```
class Circle {
       protected const double PI = 3.14; protected double Radius = 14.9;
       public virtual double Area() { return PI * Radius * Radius;
       }
   }
   class Cone : Circle { protected double Side = 10.2;
       public override double Area() { return PI * Radius * Side; }
       static void Main(string[] args) {
           Circle objRunOne = new Circle();
           Console.WriteLine("Area is: " + objRunOne.Area());
           Circle objRunTwo = new Cone();
           Console.WriteLine("Area is: " + objRunTwo.Area());
       }
   }
```

2. Object-oriented class relationships

There are four relationships popular in Object-Oriented Programming are Inheritance, Aggregation, Association, and Composition. Besides, still have many types others include the types combined or the specific types. Therein, Inheritance is established when A inherits B and called an "IS-A" relationship; Aggregation is established when the object brought some other objects and called a "HAS-A" relationship; Association is established when object associates with other objects and they depend on this; Composition is established when the object is composed of other objects or a compound object.

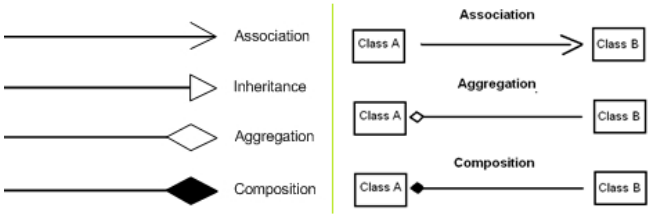

Figure 6: Object-oriented class relationships

➔ Aggregation and Composition are subsets of association meaning they are specific cases of association. In both aggregation and composition object of one class owns the object of another class. However, still have the subtle difference that is Aggregation implies a relationship where the child can exist independently of the parent; *for example: have Class (parent) and Student (child), when deleting the Class then the Students still exist*. On the other hand, Composition implies a relationship where the child cannot exist independent of the parent; *for example: have House (parent) and Room (child). The apparent that Rooms don't exist separate from a House*.

Example code about four relationships:

```
/*
 Inheritance - The manager must be the company's employee and holds a manager position
 Aggregation - He has a direct line to his office
 Association - There are employees that work under him
 Composition - His monthly salary consists of Rate + Hours
*/
public class Manager : Employee
{
    private readonly decimal perHourRate = 10.00;
    private readonly int perMonthHour = 8 * 22; // Daily 8 hours 22 days
    public string Extension { get; set;}
    public List<Employee> Team { get; set; }
    public decimal Salary => perHourRate * perMonthHour;
}
```

M1 Determine a design pattern from each of the creational, structural, and behavioral pattern types

The design pattern is a time-tested solution to a common software problem. Patterns enable a common design vocabulary, improving communication, easing documentation. Besides, it captures design expertise and allows that expertise to be transferred. On the other hand, it also provides a common reusable solution to the problems often encountered in software design.

There are many design patterns. Each pattern can be adapted to a different situation. Design Patterns in the object-oriented world is a reusable solution to common software design problems that occur repeatedly during the development of real applications. It is a template or description of how to solve problems that can be used in many situations. There are 23 design patterns and these patterns are grouped into three main categories are Creational Patterns, Structural Patterns, and Behavioral Patterns.

Below, I will introduce one design pattern for each pattern type in three main patterns. Detail, in creational I will say about singleton, in structural I will say about decorator, in behavioral I will say about iterator.

1. Creational Patterns: Singleton

In singleton design pattern ensures a class has only one instance in the program and provides a global point of access to it. Mean a class that only allows a single instance of itself to be created and usually gives simple access to that instance. Most commonly, singletons don't allow any parameters to be specified when creating the instance since the second request of an instance

with a different parameter could be problematic. If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.

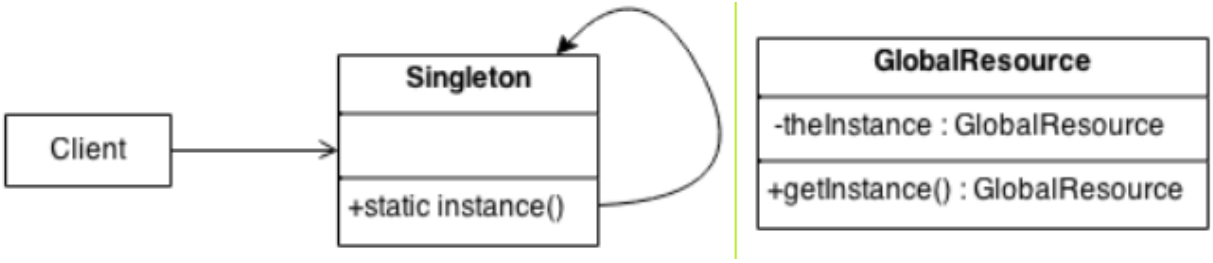

Figure 7: Structure of Singleton pattern

Example code:

```
// Bad code! Do not use it!
public sealed class Singleton
{
    //Private Constructor.
    private Singleton()
    {

    }
    private static Singleton instance = null;

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

2. Structural patterns: Decorator

A 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.

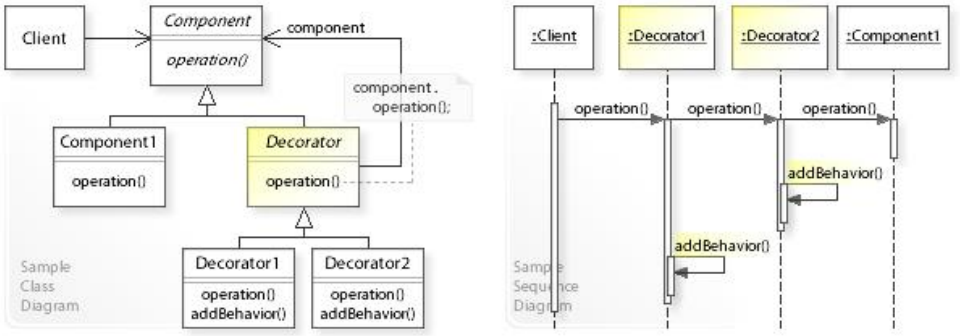

Figure 8: Structure of Decorator pattern

Example code:

```
namespace WikiDesignPatterns {
   public interface IBike
   {    string GetDetails();
        double GetPrice();
   }
   public class AluminiumBike : Ibike     {
        public double GetPrice() { return 100; }
        public string GetDetails() { return "Aluminium Bike"; }
   }
   public class CarbonBike : Ibike {
        public double GetPrice() { return 1000; }
        public string GetDetails() { return "Carbon"; }
   }
public abstract class BikeAccessories : Ibike     {
       private readonly IBike _bike;
       public BikeAccessories(IBike bike) { _bike = bike; }
       public virtual double GetPrice()   { return _bike.GetPrice(); }
       public virtual string GetDetails() { return _bike.GetDetails(); }
   }
   public class SecurityPackage : BikeAccessories
   {    public SecurityPackage(IBike bike):base(bike) { }
        public override string GetDetails() { return base.GetDetails() + " + Security Package"; }
        public override double GetPrice()   { return base.GetPrice() + 1; }
   }
   public class SportPackage : BikeAccessories {
        public SportPackage(IBike bike) : base(bike) {
        }
        public override string GetDetails() { return base.GetDetails() + " + Sport Package"; }
        public override double GetPrice()   { return base.GetPrice() + 10; }
   }
   public class BikeShop {
        public static void UpgradeBike()
```

```
        {   var basicBike = new AluminiumBike();
            BikeAccessories upgraded = new SportPackage(basicBike);
            upgraded = new SecurityPackage(upgraded);
            Console.WriteLine($"Bike: '{upgraded.GetDetails()}' Cost: {upgraded.GetPrice()}");
        }
    }
}
```

3. Behavioral patterns: Iterator

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

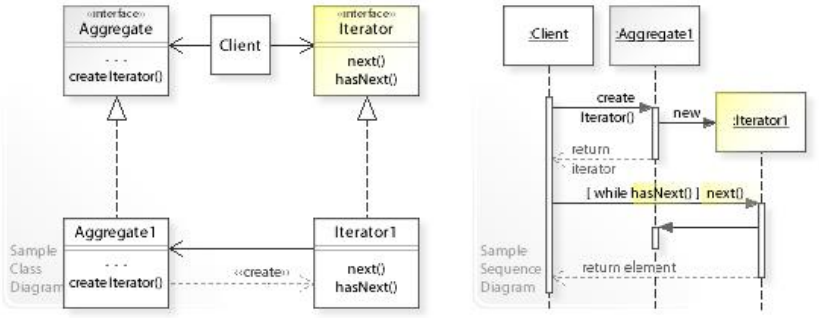

Figure 9: Structure of Iterator pattern

Example code:

```
var primes = new List<int>{ 2, 3, 5, 7, 11, 13, 17, 19 };
long m = 1;
foreach (var p in primes)
   m *= p;
```

LO2 Design a series of UML class diagrams

P2 Design and build class diagrams using a UML tool

1. Our scenario detail

A restaurant needs to manage invoices. In Bill, there must be items (food, drink) and staff.

In which, the food needs to manage additional attributes such as the food code, the name of the dish, the price of the dish. Drinks need to manage additional attributes such as drink code, drink name, drink price. Staff needs to manage additional properties such as staff code, staff name,

position. The Bill needs to manage additional attributes such as bill code, issue date, number of dishes, the number of drinks, tax, the total amount.

In addition, the restaurant bill management system needs to meet the following requirements. First, add new data (Food, Drink, Staff, Bill). Second, show the bill stored in the library. Fourth, find the bill stored in the library by code. Fifth, delete the bill stored in the library by code. Sixth, update the bill stored in the library by code. Finally, exit the system.

2. Use-case diagrams

This use-case diagram simply is stating operations the admin and the staff can perform.

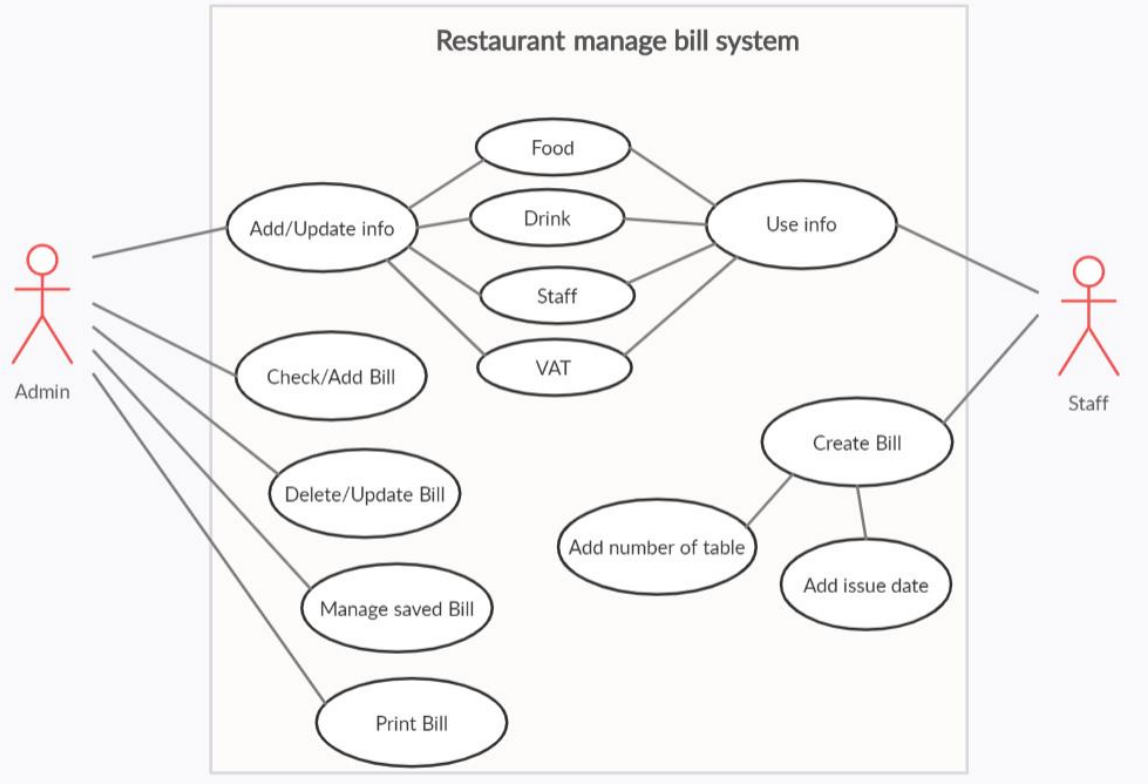

Figure 10: Use-case diagrams for restaurant management system

3. Class diagrams

The class diagram has four classes. In it, Bill is the main class used to create bills. The remaining three classes are staff, food and drink, which are the classes that manage the attributes needed for the bill class to use.

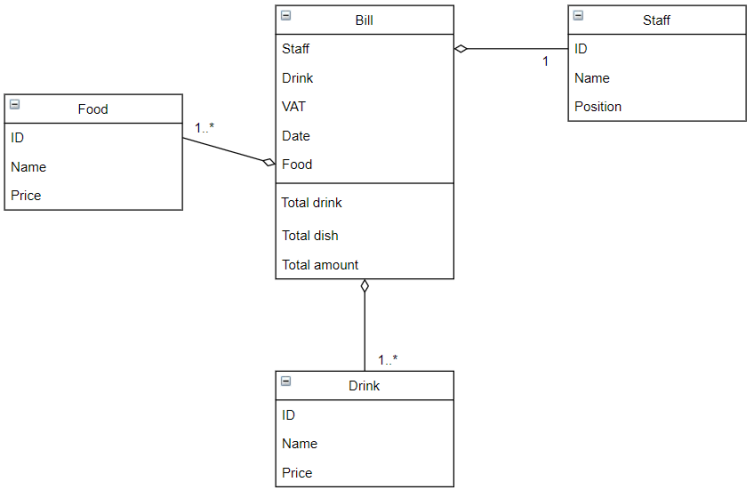

Figure 11: Class diagrams for restaurant management system

4. Pseudo-codes

```
// Food class
BEGIN
  OUTPUT ID
  OUTPUT Name
  OUTPUT Price
  OUTPUT Selection
  DISPLAY "1-Add information of new food"
        "2-Exit"
END
// Drink class
BEGIN
  OUTPUT ID
  OUTPUT Name
  OUTPUT Price
  OUTPUT Selection
  DISPLAY "1-Add information of new drink"
        "2-Exit"
```

```
END
// Staff class
BEGIN
  OUTPUT ID
  OUTPUT Name
  OUTPUT Position
  OUTPUT Selection
  DISPLAY "1-Add information of new staff"
        "2-Exit"
END
// Bill class
BEGIN
  OUTPUT Issue date
  INPUT  ID of food
     OUTPUT Name of food
     OUTPUT Price of food
  INPUT  ID of drink
     OUTPUT Name of drink
     OUTPUT Price of drink
  INPUT  ID of staff
     OUTPUT ID of staff
  SET    Total food = (price of food * total) + ...+n
     OUTPUT Total food
  SET    Total drink = (price of drink * total) + ...+n
     OUTPUT Total drink
  SET    Total amount = (Total food + Total drink) * VAT
     OUTPUT Total amount
```

```
OUTPUT Selection
        DISPLAY   "1-Find saved bill follow ID"
                  "2-Show all saved bill"
                  "3-Delete saved bill follow ID"
                  "4-Update saved bill follow ID"
                  "5-Create new bill"
                  "6-Print this bill"
                  "7-Exit"
END
```

5. Activity diagrams

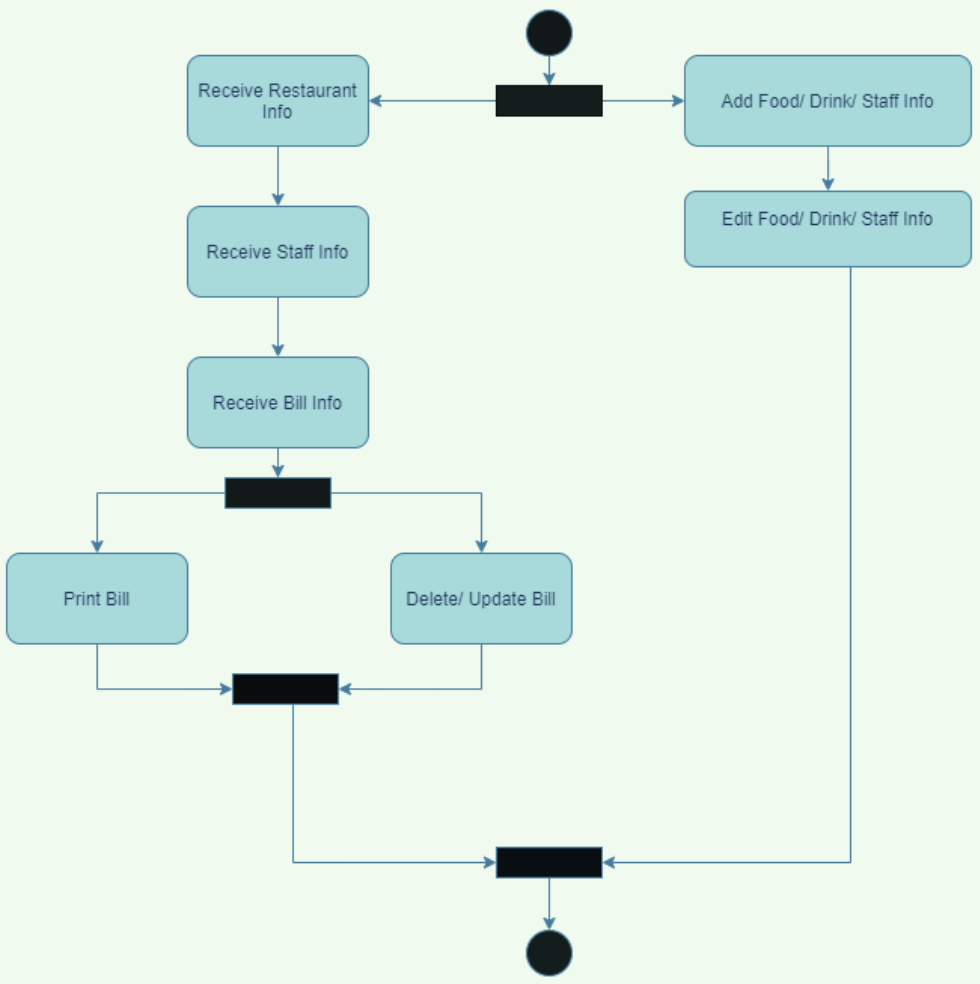

Figure 12: Activity diagram for restaurant management system

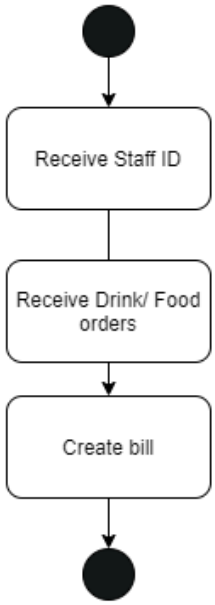

Figure 13: Activity diagram of Staff

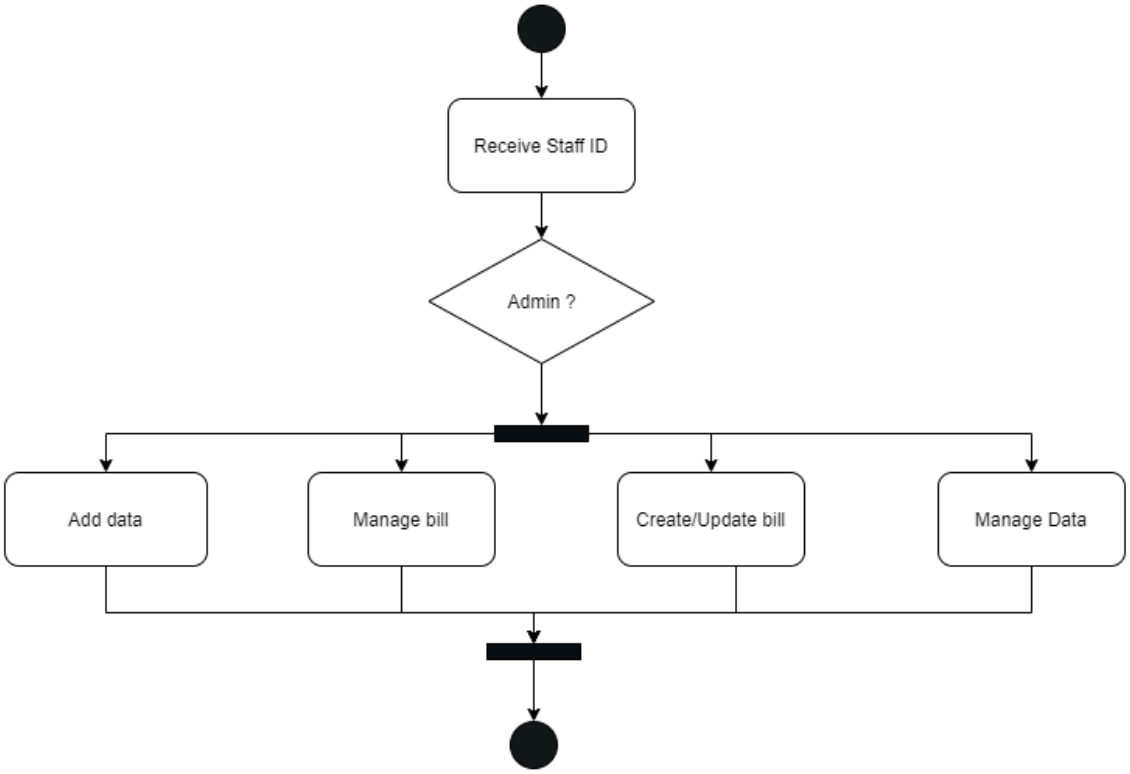

Figure 14: Activity diagram of Admin

M2 Define class diagrams for specific design patterns using a UML tool

1. Class diagram for all type in Creational Patterns

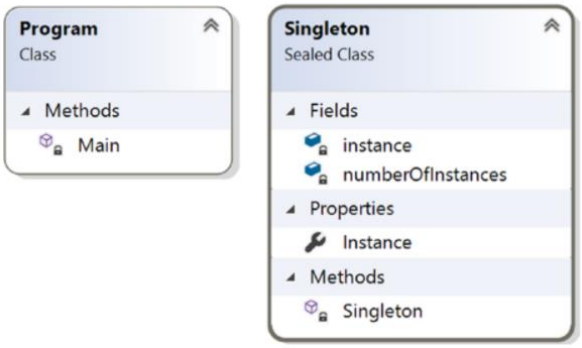

Figure 15: Class diagram for Singleton Pattern

Example code:

```
using System;
 namespace SingletonPatternEx {
 public sealed class Singleton {
 private static readonly Singleton instance=new Singleton();

 private int numberOfInstances = 0;
//Private constructor is used to prevent
//creation of instances with 'new' keyword outside this class

 private Singleton(){
       Console.WriteLine("Instantiating inside the private constructor.");
       numberOfInstances++;
       Console.WriteLine("Number of instances ={0}", numberOfInstances);
}
 public static Singleton Instance {
       get {
         Console.WriteLine("We already have an instance now.Use it.");
         return instance;
    }}}
 class Program {
 static void Main(string[] args) {Console.WriteLine("***Singleton Pattern Demo***\n");
//Console.WriteLine(Singleton.MyInt);
// Private Constructor.So,we cannot use 'new' keyword.
       Console.WriteLine("Trying to create instance s1.");
       Singleton s1 = Singleton.Instance;
       Console.WriteLine("Trying to create instance s2.");
       Singleton s2 = Singleton.Instance;
       if (s1 == s2) { Console.WriteLine("Only one instance exists."); }
       else { Console.WriteLine("Different instances exist."); } Console.Read();
  }
 }
}
```

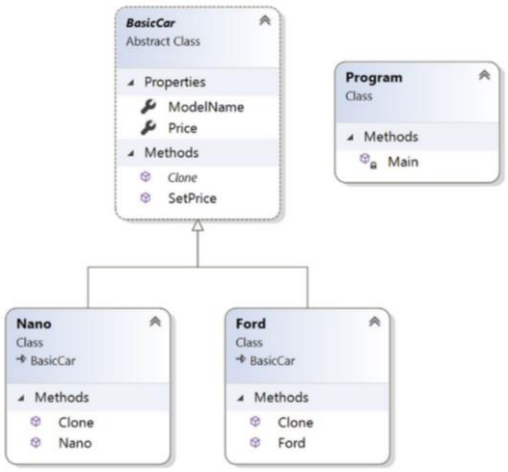

Figure 16: Class diagram for Prototype Pattern

Example code:

```
//BasicCar.cs
using System;
namespace PrototypePattern {
public abstract class BasicCar {
public string ModelName{get;set;}
public int Price {get; set;}
public static int SetPrice() { int price = 0;
Random r = new Random(); int p = r.Next(200000, 500000);
price = p; return price;
}
public abstract BasicCar Clone(); }
}
//Nano.cs
using System;
namespace PrototypePattern {
public class Nano:BasicCar {
public Nano(string m) {
ModelName = m;
}
public override BasicCar Clone() {
    return (Nano) this.MemberwiseClone();//shallow Clone
  }
 }
}
//Ford.cs
using System;
namespace PrototypePattern {
public class Ford:BasicCar {
public Ford(string m) { ModelName = m; }
public override BasicCar Clone() {
```

```
return (Ford)this.MemberwiseClone();
  }
 }
}
//Client
using System;
namespace PrototypePattern {
class Program {
static void Main(string[] args) {
Console.WriteLine("***Prototype Pattern Demo***\n");
//Base or Original Copy
BasicCar nano_base = new Nano("Green Nano") {Price = 100000};
BasicCar ford_base = new Ford("Ford Yellow") {Price = 500000};
BasicCar bc1;
//Nano
bc1 = nano_base.Clone();
bc1.Price = nano_base.Price+BasicCar.SetPrice();
Console.WriteLine("Car is: {0}, and it's price is Rs. {1} ",bc1.ModelName,bc1.Price);
//Ford
bc1 = ford_base.Clone();
bc1.Price = ford_base.Price+BasicCar.SetPrice();
Console.WriteLine("Car is: {0}, and it's price is Rs. {1}",
bc1.ModelName, bc1.Price);
Console.ReadLine();
  }
 }
}
```

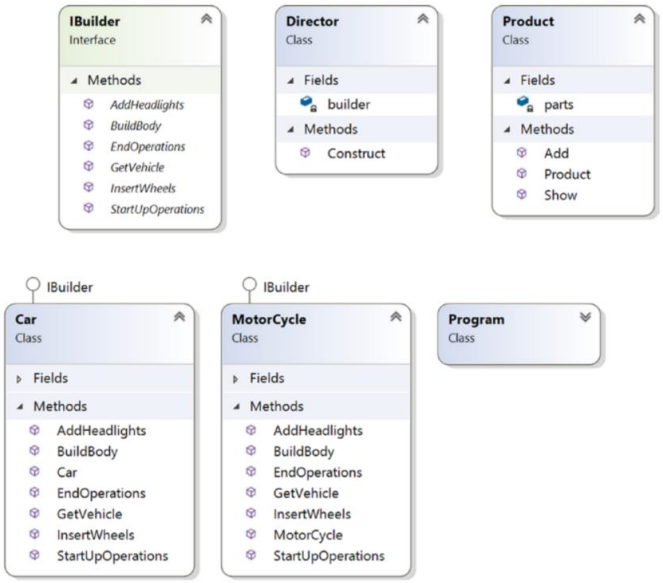

Figure 17: Class diagram for Builder Pattern

Example code:

```
using System;
using System.Collections.Generic;//For LinkedList
namespace BuilderPattern {
// Builders common interface
interface IBuilder { void StartUpOperations();
                     void BuildBody();     void InsertWheels();
                     void AddHeadlights(); void EndOperations();
Product GetVehicle();}
// ConcreteBuilder: Car
class Car : IBuilder{
private string brandName;
private Product;
public Car(string brand){
product = new Product(); this.brandName = brand;
}
public void StartUpOperations(){ //Starting with brandname
product.Add(string.Format("Car Model name :{0}",this.brandName));
}
public void BuildBody(){ product.Add("This is a body of a Car");}
public void InsertWheels(){ product.Add("4 wheels are added");}
public void AddHeadlights() { product.Add("2 Headlights are added");}
public void EndOperations() { //Nothing in this case
}
public Product GetVehicle() { return product;}
}
// ConcreteBuilder:Motorcycle
class MotorCycle : IBuilder {
private string brandName;
private Product product;
public MotorCycle(string brand) { product = new Product(); this.brandName = brand;
}
public void StartUpOperations() { //Nothing in this case
}
public void BuildBody() { product.Add("This is a body of a Motorcycle");}
public void InsertWheels() { product.Add("2 wheels are added");}
public void AddHeadlights() { product.Add("1 Headlights are added");}
public void EndOperations() {
//Finishing up with brandname
product.Add(string.Format("Motorcycle Model name :{0}", this.brandName));
}
public Product GetVehicle() { return product;}
}
// "Product"
class Product {
```

```
// We can use any data structure that you prefer e.g.List<string> etc.
private LinkedList<string> parts;
public Product() { parts = new LinkedList<string>();}
public void Add(string part) {
//Adding parts
parts.AddLast(part);
}
public void Show() {
Console.WriteLine("\nProduct completed as below :");
foreach (string part in parts) Console.WriteLine(part);}
}
// "Director"
class Director { IBuilder builder;
// A series of steps-in real life, steps are complex.
public void Construct(IBuilder builder) {
this.builder = builder; builder.StartUpOperations();
builder.BuildBody(); builder.InsertWheels();
builder.AddHeadlights(); builder.EndOperations(); }
}
class Program {
static void Main(string[] args)
{ Console.WriteLine("***Builder Pattern Demo***");
Director director = new Director();
IBuilder b1 = new Car("Ford"); IBuilder b2 = new MotorCycle("Honda");
// Making Car
director.Construct(b1); Product p1 = b1.GetVehicle(); p1.Show();
//Making MotorCycle
director.Construct(b2); Product p2 = b2.GetVehicle(); p2.Show();
Console.ReadLine();
  }
 }
}
```

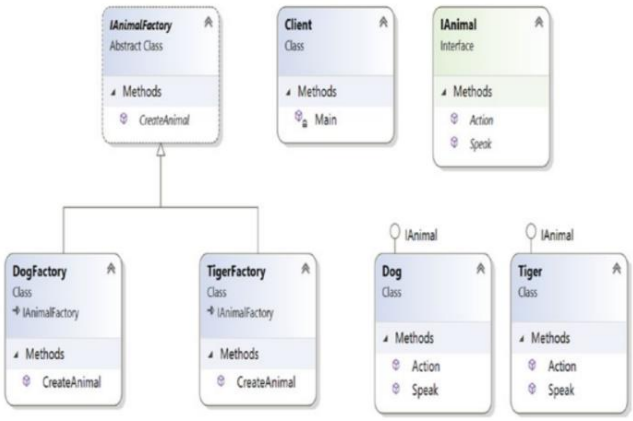

Figure 18: Class diagram for Factory Method Pattern

Example code:

```
using System;
namespace FactoryMethodPattern {
public interface IAnimal { void Speak(); void Action();}
public class Dog : IAnimal {
public void Speak()
{ Console.WriteLine("Dog says: Bow-Wow.");}
public void Action(){
Console.WriteLine("Dogs prefer barking...\n");}
}
public class Tiger : IAnimal{
public void Speak(){
Console.WriteLine("Tiger says: Halum.");}
public void Action(){
Console.WriteLine("Tigers prefer hunting...\n");}
}
public abstract class IAnimalFactory{
public abstract IAnimal CreateAnimal();}
public class DogFactory : IAnimalFactory{
public override IAnimal CreateAnimal(){
//Creating a Dog
return new Dog();}
}
public class TigerFactory : IAnimalFactory{
public override IAnimal CreateAnimal(){
//Creating a Tiger
return new Tiger();}
}
class Client{
static void Main(string[] args){
Console.WriteLine("***Factory Pattern Demo***\n");
// Creating a Tiger Factory
IAnimalFactory tigerFactory =new TigerFactory();
// Creating a tiger using the Factory Method
IAnimal aTiger = tigerFactory.CreateAnimal();
aTiger.Speak(); aTiger.Action();
// Creating a DogFactory
IAnimalFactory dogFactory = new DogFactory();
// Creating a dog using the Factory Method
IAnimal aDog = dogFactory.CreateAnimal();
aDog.Speak(); aDog.Action(); Console.ReadKey();
  }
 }
}
```

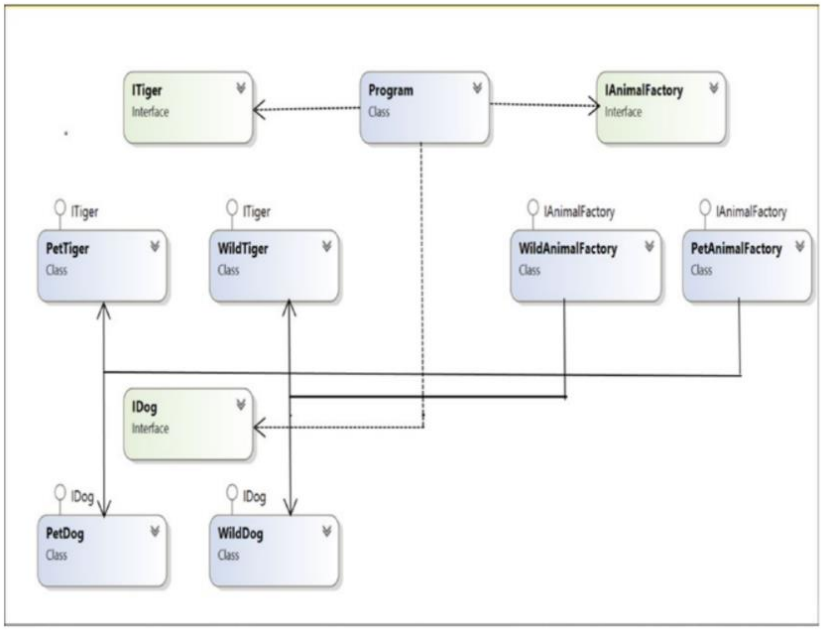

Figure 19: Class diagram for Abstract Factory Pattern

Example code:

```
using System;
namespace AbstractFactoryPattern{
public interface IDog{
void Speak(); void Action();
}
public interface ITiger{
void Speak(); void Action();
}
#region Wild Animal collections
class WildDog : IDog {
public void Speak() {
Console.WriteLine("Wild Dog says: Bow-Wow.");}
public void Action() {
Console.WriteLine("Wild Dogs prefer to roam freely in jungles.\n");}
}
class WildTiger : ITiger {
public void Speak() {
Console.WriteLine("Wild Tiger says: Halum.");}
public void Action() {
Console.WriteLine("Wild Tigers prefer hunting in jungles.\n");}
}
#endregion
#region Pet Animal collections
class PetDog : IDog {
public void Speak() {
```

```
Console.WriteLine("Pet Dog says: Bow-Wow.");}
public void Action() {
Console.WriteLine("Pet Dogs prefer to stay at home.\n");}
}
class PetTiger : ITiger {
public void Speak() {
Console.WriteLine("Pet Tiger says: Halum.");}
public void Action() {
Console.WriteLine("Pet Tigers play in an animal circus.\n");}}
#endregion
//Abstract Factory
public interface IAnimalFactory
{ IDog GetDog(); ITiger GetTiger(); }
//Concrete Factory-Wild Animal Factory
public class WildAnimalFactory : IAnimalFactory {
public IDog GetDog() {
return new WildDog(); }
public ITiger GetTiger() {
return new WildTiger(); } }
//Concrete Factory-Pet Animal Factory
public class PetAnimalFactory : IAnimalFactory {
public IDog GetDog() {
return new PetDog(); }
public ITiger GetTiger() {
return new PetTiger(); } }
class Program {
static void Main(string[] args) {
Console.WriteLine("***Abstract Factory Pattern Demo***\n");
//Making a wild dog through WildAnimalFactory
IAnimalFactory wildAnimalFactory = new WildAnimalFactory();
IDog wildDog = wildAnimalFactory.GetDog();
wildDog.Speak(); wildDog.Action();
//Making a wild tiger through WildAnimalFactory
ITiger wildTiger = wildAnimalFactory.GetTiger();
wildTiger.Speak(); wildTiger.Action();
Console.WriteLine("******************");
//Making a pet dog through PetAnimalFactory
IAnimalFactory petAnimalFactory = new PetAnimalFactory();
IDog petDog = petAnimalFactory.GetDog();
petDog.Speak(); petDog.Action();
//Making a pet tiger through PetAnimalFactory
ITiger petTiger = petAnimalFactory.GetTiger();
petTiger.Speak(); petTiger.Action(); Console.ReadLine();
  }
 }
}
```

2. Class diagram for all type in Structural Patterns

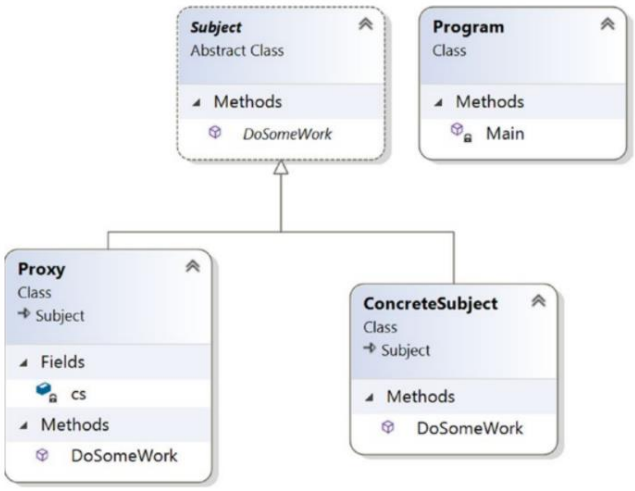

Figure 20: Class diagram for Proxy Pattern

Example code:

```
using System;
namespace ProxyPattern {
/// <summary>
/// Abstract class Subject
/// </summary>
public abstract class Subject {
public abstract void DoSomeWork();}
/// <summary>
/// ConcreteSubject class
/// </summary>
public class ConcreteSubject : Subject {
public override void DoSomeWork() { Console.WriteLine("ConcreteSubject.DoSomeWork()");}}
/// <summary>
/// Proxy class
/// </summary>
public class Proxy : Subject { Subject cs;
public override void DoSomeWork() { Console.WriteLine("Proxy call happening now...");
//Lazy initialization:We'll not instantiate until the method is
//called
if (cs == null) { cs = new ConcreteSubject();} cs.DoSomeWork();}}
class Program {
static void Main(string[] args) { Console.WriteLine("***Proxy Pattern Demo***\n");
Proxy px = new Proxy(); px.DoSomeWork(); Console.ReadKey();
  }
 }
}
```

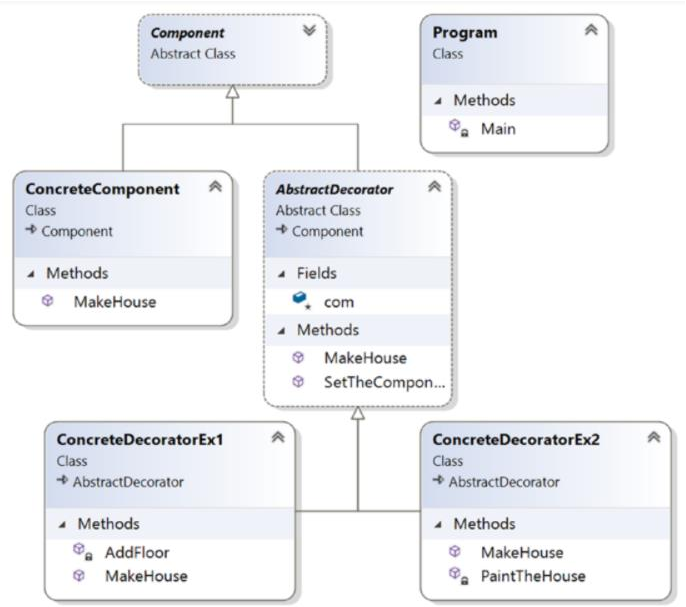

Figure 21: Class diagram for Decorator Pattern

Example code:

```
using System;
namespace DecoratorPattern {
abstract class Component {
public abstract void MakeHouse(); }
class ConcreteComponent : Component {
public override void MakeHouse()
{ Console.WriteLine("Original House is complete. It is closed for modification.");}}
abstract class AbstractDecorator : Component { protected Component com ;
public void SetTheComponent(Component c) { com = c; }
public override void MakeHouse() { if (com != null) {
com.MakeHouse();//Delegating the task
}}}
class ConcreteDecoratorEx1 : AbstractDecorator {
public override void MakeHouse() {
base.MakeHouse(); Console.WriteLine("***Using a decorator***");
//Decorating now.
AddFloor();
//You can put additional stuff as per your needs.
}
private void AddFloor()
{ Console.WriteLine("I am making an additional floor on top of it.");}}
class ConcreteDecoratorEx2 : AbstractDecorator {
public override void MakeHouse() {
Console.WriteLine(""); base.MakeHouse();
Console.WriteLine("***Using another decorator***");
//Decorating now.
```

```
PaintTheHouse();
//You can add additional stuffs as per your need
}
private void PaintTheHouse() {
Console.WriteLine("Now I am painting the house."); }}
class Program {
static void Main(string[] args) {
Console.WriteLine("***Decorator pattern Demo***\n");
ConcreteComponent cc = new ConcreteComponent();
ConcreteDecoratorEx1 decorator1 = new ConcreteDecoratorEx1();
decorator1.SetTheComponent(cc); decorator1.MakeHouse();
ConcreteDecoratorEx2 decorator2 = new ConcreteDecoratorEx2();
//Adding results from decorator1
decorator2.SetTheComponent(decorator1);
decorator2.MakeHouse(); Console.ReadKey();
  }
 }
}
```

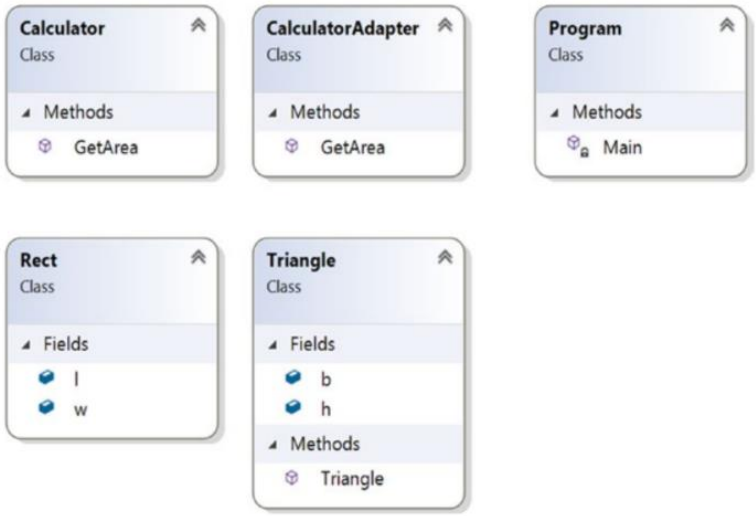

Figure 22: Class diagram for Adapter Pattern

Example code:

```
using System;
namespace AdapterPattern {
class Rect {
public double length; public double width;}
class Calculator {
public double GetArea(Rect rect) {
return rect.length * rect.width;}}
```

```
class Triangle {
public double baseT;//base
public double height;//height
public Triangle(int b, int h) {
this.baseT = b; this.height = h; }}
class CalculatorAdapter {
public double GetArea(Triangle triangle) {
Calculator c = new Calculator(); Rect rect = new Rect();
//Area of Triangle=0.5*base*height
rect.length = triangle.baseT;
rect.width = 0.5 * triangle.height;
return c.GetArea(rect); }}
class Program {
static void Main(string[] args) {
Console.WriteLine("***Adapter Pattern Demo***\n");
CalculatorAdapter cal = new CalculatorAdapter();
Triangle t = new Triangle(20, 10);
Console.WriteLine("Area of Triangle is" + cal.GetArea(t) + "Square unit");
Console.ReadKey();
  }
 }
}
```

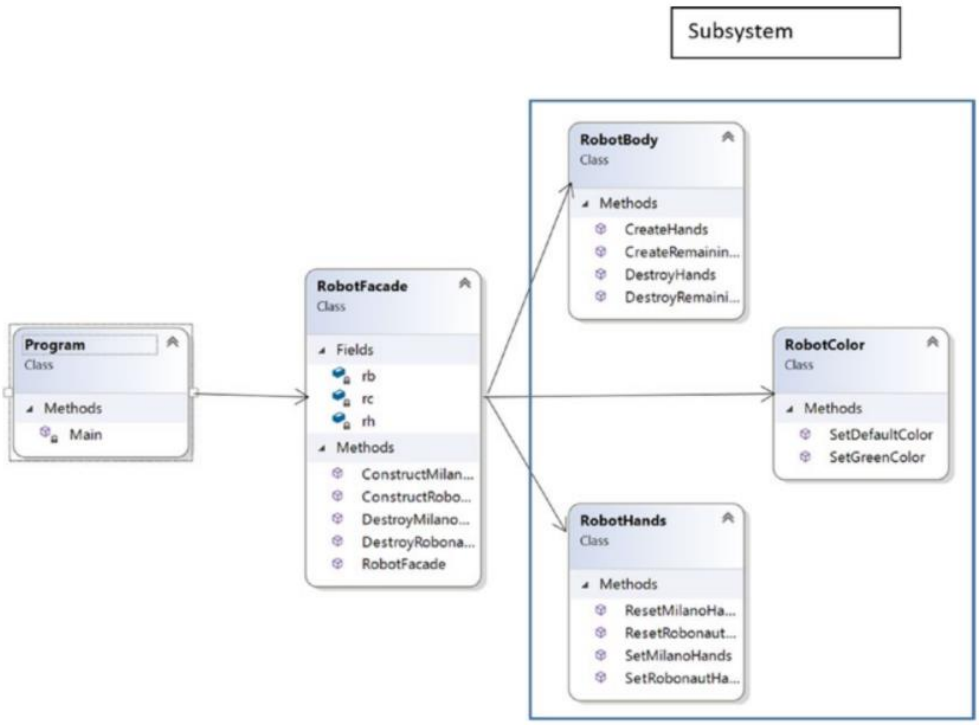

Figure 23: Class diagram for Facade Pattern

Example code:

```
// RobotBody.cs
using System;
namespace FacadePattern.RobotParts {
public class RobotBody {
public void CreateHands() { Console.WriteLine("Hands manufactured"); }
public void CreateRemainingParts()
{ Console.WriteLine("Remaining parts (other than hands) are created");}
public void DestroyHands() {
Console.WriteLine("The robot's hands are destroyed");}
public void DestroyRemainingParts() {
Console.WriteLine("The robot's remaining parts are destroyed");}}}
//RobotColor.cs
using System;
namespace FacadePattern.RobotParts{
public class RobotColor{
public void SetDefaultColor(){
Console.WriteLine("This is steel color robot.");}
public void SetGreenColor(){
Console.WriteLine("This is a green color robot.");}}}
// RobotHands.cs
using System;
namespace FacadePattern.RobotParts{
public class RobotHands{
public void SetMilanoHands(){
Console.WriteLine("The robot will have EH1 Milano hands");}
public void SetRobonautHands(){
Console.WriteLine("The robot will have Robonaut hands");}
public void ResetMilanoHands(){
Console.WriteLine("EH1 Milano hands are about to bedestroyed");}
public void ResetRobonautHands(){
Console.WriteLine("Robonaut hands are about to be destroyed");}}}
// RobotFacade.cs
using System;
using FacadePattern.RobotParts;
namespace FacadePattern{
public class RobotFacade{
RobotColor rc; RobotHands rh ; RobotBody rb;
public RobotFacade(){
rc = new RobotColor(); rh = new RobotHands(); rb = new RobotBody();}
public void ConstructMilanoRobot(){
Console.WriteLine("Creation of a Milano Robot Start");
rc.SetDefaultColor(); rh.SetMilanoHands();
rb.CreateHands(); rb.CreateRemainingParts();
Console.WriteLine("Milano Robot Creation End"); Console.WriteLine();}
public void ConstructRobonautRobot(){
Console.WriteLine("Initiating the creational process of a Robonaut Robot");
```

```
rc.SetGreenColor(); rh.SetRobonautHands();
rb.CreateHands(); rb.CreateRemainingParts();
Console.WriteLine("A Robonaut Robot is created"); Console.WriteLine();}
public void DestroyMilanoRobot()
{Console.WriteLine("Milano Robot's destruction process is started");
rh.ResetMilanoHands(); rb.DestroyHands(); rb.DestroyRemainingParts();
Console.WriteLine("Milano Robot's destruction process is over");
Console.WriteLine();}
public void DestroyRobonautRobot() {
Console.WriteLine("Initiating a Robonaut Robot's destruction process.");
rh.ResetRobonautHands(); rb.DestroyHands(); rb.DestroyRemainingParts();
Console.WriteLine("A Robonaut Robot is destroyed"); Console.WriteLine();}}}
// Program.cs
using System;
namespace FacadePattern {
class Program{
static void Main(string[] args){Console.WriteLine("***Facade Pattern Demo***\n");
//Creating Robots
RobotFacade rf1 = new RobotFacade(); rf1.ConstructMilanoRobot();
RobotFacade rf2 = new RobotFacade(); rf2.ConstructRobonautRobot();
//Destroying robots
rf1.DestroyMilanoRobot(); rf2.DestroyRobonautRobot(); Console.ReadLine();
  }
 }
}
```

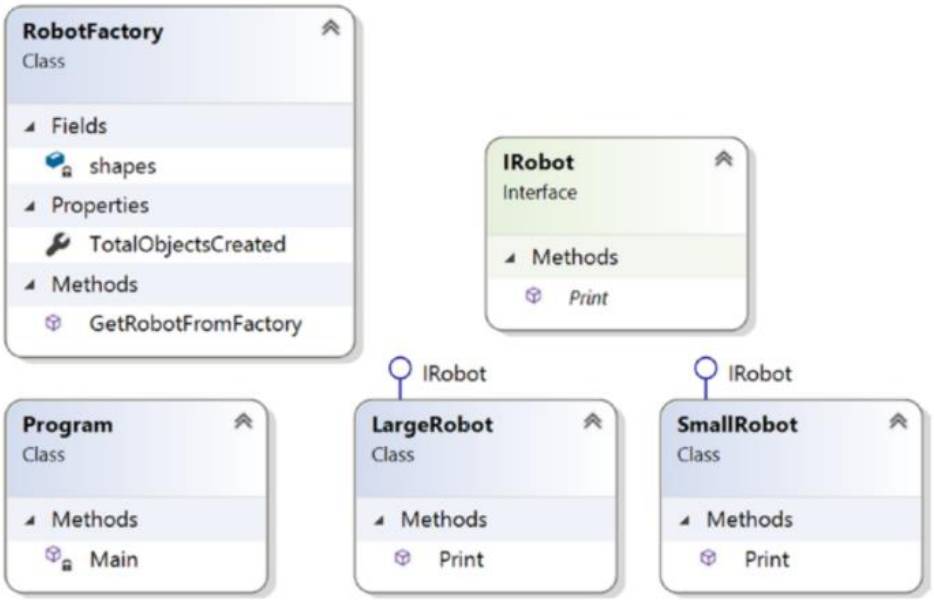

Figure 24: Class diagram for Flyweight Pattern

Example code:

```
using System;
using System.Collections.Generic;//Dictionary is used here
namespace FlyweightPattern {
/// <summary>
/// The 'Flyweight' interface
/// </summary>
interface IRobot{void Print();}
/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class SmallRobot : IRobot{
public void Print(){
Console.WriteLine("This is a small Robot");}}
/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class LargeRobot : IRobot{public void Print()
{Console.WriteLine("I am a large Robot");}}
/// <summary>
/// The 'FlyweightFactory' class
/// </summary>
class RobotFactory{
Dictionary<string, IRobot> shapes = new Dictionary<string,IRobot>();
public int TotalObjectsCreated{get {return shapes.Count;}}
public IRobot GetRobotFromFactory(string robotType)
{IRobot robotCategory = null; if (shapes.ContainsKey(robotType))
{robotCategory = shapes[robotType];}
else{ switch (robotType) {case "Small":
robotCategory = new SmallRobot();
shapes.Add("Small", robotCategory); break;
case "Large": robotCategory = new LargeRobot();
shapes.Add("Large", robotCategory); break; default:
throw new Exception("Robot Factory can create only small and large robots");}}
return robotCategory;}}
class Program{
static void Main(string[] args){
Console.WriteLine("***Flyweight Pattern Demo***\n");
RobotFactory myfactory = new RobotFactory();
IRobot shape = myfactory.GetRobotFromFactory("Small");
shape.Print(); for (int i = 0; i < 2; i++){
shape = myfactory.GetRobotFromFactory("Small"); shape.Print();}
int NumOfDistinctRobots = myfactory.TotalObjectsCreated;
Console.WriteLine("\n Now, total numbers of distinct robot objects is = {0}\n", NumOfDistinctRobots);
for (int i = 0; i < 5; i++){}
shape = myfactory.GetRobotFromFactory("Large"); shape.Print();}
int NumOfDistinctRobots = myfactory.TotalObjectsCreated;
```

```
Console.WriteLine("\n Distinct Robot objects created till Now = {0}", NumOfDistinctRobots);
Console.ReadKey();
  }
 }
}
```

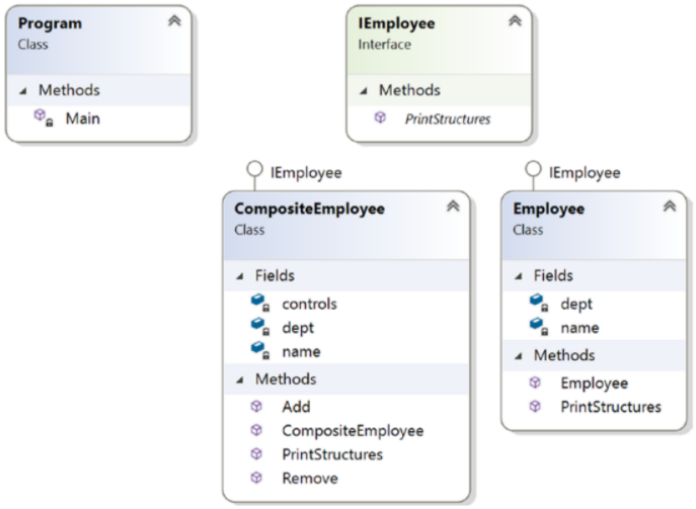

Figure 25: Class diagram for Composite Pattern

Example code:

```
using System;
using System.Collections.Generic;//For List<Employee> here
namespace CompositePattern{
interface IEmployee{
void PrintStructures();}
class CompositeEmployee : IEmployee{
private string name;
private string dept;
//The container for child objects
private List<IEmployee> controls;
// constructor
public CompositeEmployee(string name, string dept){
this.name = name; this.dept = dept;
controls = new List<IEmployee>();}
public void Add(IEmployee e){
controls.Add(e);}
public void Remove(IEmployee e){
controls.Remove(e);}
public void PrintStructures(){
Console.WriteLine("\t" + this.name + "works in" + this.dept);
```

```
foreach (IEmployee e in controls){
e.PrintStructures();
}
}
}
class Employee : IEmployee{
private string name;
private string dept;
// constructor
public Employee(string name, string dept){
this.name = name; this.dept = dept;
}
public void PrintStructures(){
Console.WriteLine("\t\t"+this.name + "works in" + this.dept);}
}
class Program{
static void Main(string[] args){
Console.WriteLine("***Composite Pattern Demo ***");
//Principal of the college
CompositeEmployee Principal = new CompositeEmployee("Dr.S.Som (Principal)","Planning-Supervising-
Managing");
//The College has 2 Head of Departments-One from Mathematics,
One from Computer Sc.CompositeEmployee hodMaths = new CompositeEmployee("Mrs.S.Das(HOD-
Maths)","Maths");
CompositeEmployee hodCompSc = new CompositeEmployee ("Mr. V.Sarcar(HOD-CSE)", "Computer Sc.");
//2 other teachers works in Mathematics department
Employee mathTeacher1 = new Employee("Math Teacher-1","Maths");
Employee mathTeacher2 = new Employee("Math Teacher-2","Maths");
//3 other teachers works in Computer Sc. department
Employee cseTeacher1 = new Employee("CSE Teacher-1","Computer Sc.");
Employee cseTeacher2 = new Employee("CSE Teacher-2", "Computer Sc.");
Employee cseTeacher3 = new Employee("CSE Teacher-3", "Computer Sc.");
//Teachers of Mathematics directly reports to HOD-Maths
hodMaths.Add(mathTeacher1);
hodMaths.Add(mathTeacher2);
//Teachers of Computer Sc. directly reports to HOD-CSE
hodCompSc.Add(cseTeacher1);
hodCompSc.Add(cseTeacher2);
hodCompSc.Add(cseTeacher3);
//Principal is on top of college
//HOD -Maths and Comp. Sc directly reports to him
Principal.Add(hodMaths);
Principal.Add(hodCompSc);
//Printing the leaf-nodes and branches in the same way.
//i.e. in each case, we are calling PrintStructures() method
Console.WriteLine("\n Testing the structure of a Principal object");
//Prints the complete structure
```

```
Principal.PrintStructures();
Console.WriteLine("\n Testing the structure of a HOD object:");
Console.WriteLine("Teachers working at Computer Science department:");
//Prints the details of Computer Sc, department
hodCompSc.PrintStructures();
//Leaf node
Console.WriteLine("\n Testing the structure of a leaf node:");
mathTeacher1.PrintStructures();
//Suppose, one computer teacher is leaving now from the
organization.hodCompSc.Remove(cseTeacher2);
Console.WriteLine("\n After CSE Teacher-2 resigned, the organization has following members:");
Principal.PrintStructures();
Console.ReadKey();
  }
 }
}
```

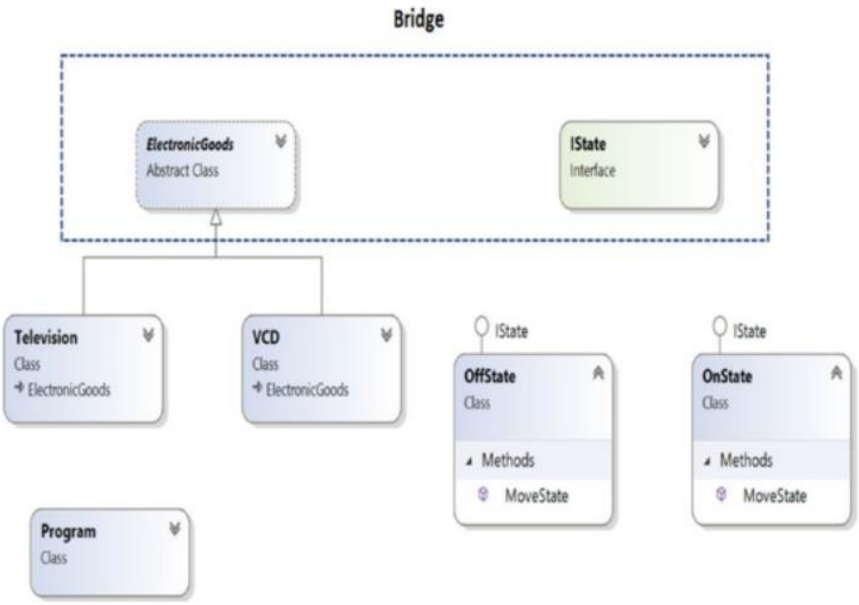

Figure 26: Class diagram for Bridge Pattern

Example code:

```
using System;
namespace BridgePattern{
//Implementor
public interface IState{
void MoveState();}
//ConcreteImplementor-1
public class OnState : IState{
public void MoveState(){
```

```
Console.Write("On State");}}
//ConcreteImplementor-2
public class OffState : IState{
public void MoveState(){
Console.Write("Off State");}}
//Abstraction
public abstract class ElectronicGoods{
//Composition - implementor
protected IState state;
//Alternative approach to properties:
//we can also pass an implementor (as input argument) inside a constructor.
//public ElectronicGoods(IState state)
//{
// this.state = state;
//}
public IState State{
get{return state;}
set{state = value;}
}
abstract public void MoveToCurrentState();}
//Refined Abstraction
public class Television : ElectronicGoods{
//public Television(IState state) : base(state)
//{
//}
/*Implementation specific:
* We are delegating the implementation to the Implementor object*/
public override void MoveToCurrentState(){
Console.Write("\n Television is functioning at : ");
state.MoveState();}}
public class VCD : ElectronicGoods{
//public VCD(IState state) : base(state)
//{
//}
/*Implementation specific:
* We are delegating the implementation to the Implementor object*/
public override void MoveToCurrentState(){
Console.Write("\n VCD is functioning at : ");
state.MoveState();}}
class Program{
static void Main(string[] args){
Console.WriteLine("***Bridge Pattern Demo***");
Console.WriteLine("\nDealing with a Television:");
//ElectronicGoods eItem = new Television(presentState);
ElectronicGoods eItem = new Television();
IState presentState = new OnState();
eItem.State = presentState;
```

```
eItem.MoveToCurrentState();
//Verifying Off state of the Television now
presentState = new OffState();
//eItem = new Television(presentState);
eItem.State = presentState;
eItem.MoveToCurrentState();
Console.WriteLine("\n \n Dealing with a VCD:");
presentState = new OnState();
//eItem = new VCD(presentState);
eItem = new VCD();
eItem.State = presentState;
eItem.MoveToCurrentState();
presentState = new OffState();
//eItem = new VCD(presentState);
eItem.State = presentState;
eItem.MoveToCurrentState();
Console.ReadLine();
  }
 }
}
```

3. Class diagram for all type in Behavioral Patterns

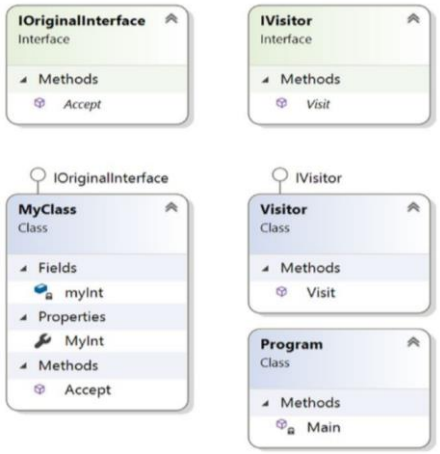

Figure 27: Class diagram for Visitor Pattern

Example code:

```
using System;
namespace VisitorPattern{
interface IOriginalInterface{
void Accept(IVisitor visitor);}
class MyClass : IOriginalInterface{
private int myInt = 5;//Initial or default value
```

```
public int MyInt{
get{return myInt;}
set{myInt = value;}}
public void Accept(IVisitor visitor){
Console.WriteLine("Initial value of the integer:{0}", myInt);
visitor.Visit(this);
Console.WriteLine("\nValue of the integer now:{0}", myInt);}}
interface IVisitor{
void Visit(MyClass myClassElement);}
class Visitor : IVisitor{
public void Visit(MyClass myClassElement){
Console.WriteLine("Visitor is trying to change the integer value.");
myClassElement.MyInt = 100;
Console.WriteLine("Exiting from Visitor.");}}
class Program{
static void Main(string[] args){
Console.WriteLine("***Visitor Pattern Demo***\n");
IVisitor visitor = new Visitor();
MyClass myClass = new MyClass();
myClass.Accept(visitor); Console.ReadLine();
  }
 }
}
```

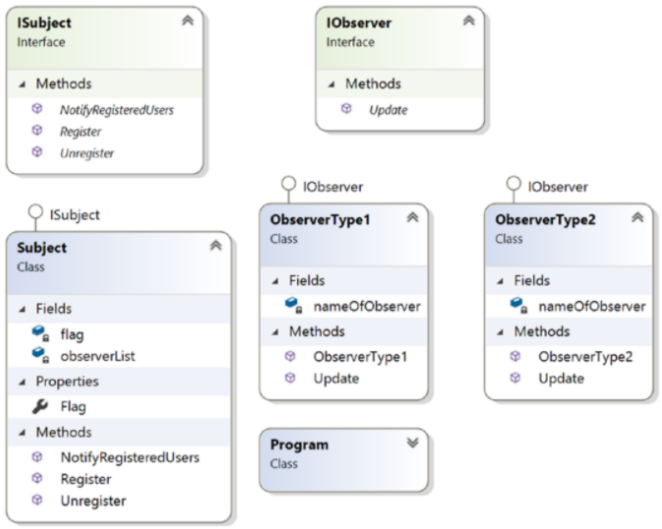

Figure 28: Class diagram for Observer Pattern

Example code:

```
using System;
using System.Collections.Generic;//We have used List<Observer> here
```

```
namespace ObserverPattern{
interface IObserver{
void Update(int i);}
class ObserverType1 : IObserver{
string nameOfObserver;
public ObserverType1(String name){
this.nameOfObserver = name;}
public void Update(int i){
Console.WriteLine("{0} has received an alert: Someone has updated myValue in Subject to: {1}", nameOf
Observer,i);
}
}
class ObserverType2 : IObserver{
string nameOfObserver;
public ObserverType2(String name){
this.nameOfObserver = name;}
public void Update(int i){
Console.WriteLine("{0} notified: myValue in Subject at present: {1}", nameOfObserver, i);
}
}
interface ISubject{
void Register(IObserver o);
void Unregister(IObserver o);
void NotifyRegisteredUsers(int i);}
class Subject:ISubject{
List<IObserver> observerList = new List<IObserver>();
private int flag; public int Flag
{get{return flag;}
set{flag = value;
//Flag value changed. So notify observer/s.
NotifyRegisteredUsers(flag);}}
public void Register(IObserver anObserver){
observerList.Add(anObserver);}
public void Unregister(IObserver anObserver){
observerList.Remove(anObserver);}
public void NotifyRegisteredUsers(int i){
foreach (IObserver observer in observerList){
observer.Update(i);}}}
class Program{
static void Main(string[] args){
Console.WriteLine("***Observer Pattern Demo***\n");
//We have 3 observers-2 of them are ObserverType1, 1 of them is of ObserverType2
IObserver myObserver1 = new ObserverType1("Roy");
IObserver myObserver2 = new ObserverType1("Kevin");
IObserver myObserver3 = new ObserverType2("Bose");
Subject subject = new Subject();
//Registering the observers-Roy,Kevin,Bose
```

```
subject.Register(myObserver1);
subject.Register(myObserver2);
subject.Register(myObserver3);
Console.WriteLine(" Setting Flag = 5 ");
subject.Flag = 5;
//Unregistering an observer(Roy))
subject.Unregister(myObserver1);
//No notification this time Roy. Since it is unregistered.
Console.WriteLine("\n Setting Flag = 50 ");
subject.Flag = 50;
//Roy is registering himself again
subject.Register(myObserver1);
Console.WriteLine("\n Setting Flag = 100 ");
subject.Flag = 100; Console.ReadKey();
  }
 }
}
```

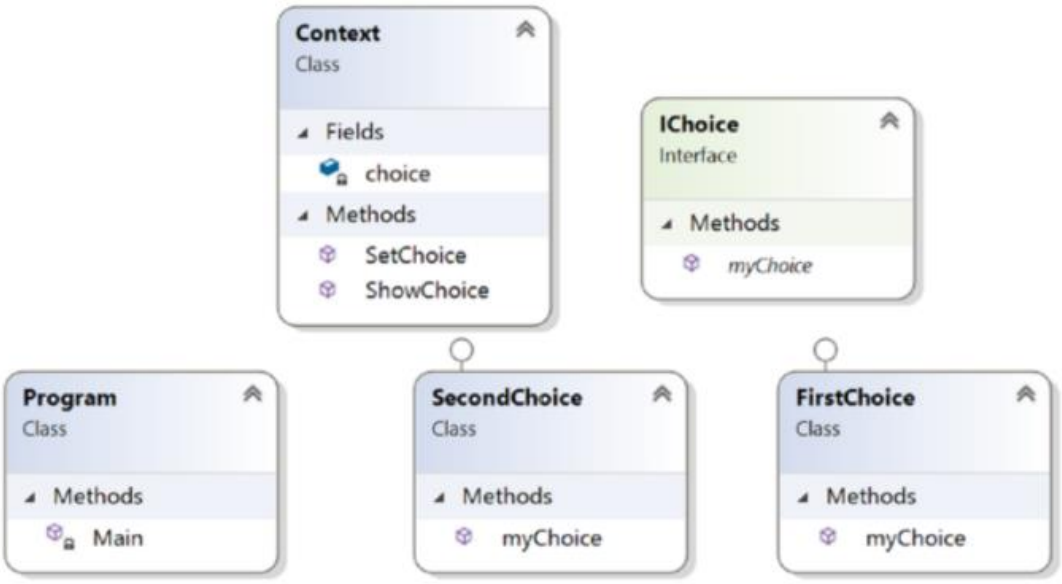

Figure 29: Class diagram for Strategy (Policy) Pattern

Example code:

```
// IChoice.cs
using System;
namespace StrategyPattern{
public interface IChoice{
void MyChoice();}}
// FirstChoice.cs
```

```
using System;
namespace StrategyPattern{
public class FirstChoice:IChoice{
public void MyChoice(){
Console.WriteLine("Traveling to Japan");}}}
// SecondChoice.cs
using System;
namespace StrategyPattern{
public class SecondChoice:IChoice{
public void MyChoice(){
Console.WriteLine("Traveling to India");}
 }
}
// Context.cs
using System;
namespace StrategyPattern{
public class Context{
IChoice choice;
/*It's our choice. We prefer to use a setter method instead of
using a constructor. We can call this method whenever we want to
change the "choice behavior" on the fly*/
public void SetChoice(IChoice choice){
this.choice = choice;}
/* This method will help us to delegate the particular
object's choice behavior/characteristic*/
public void ShowChoice(){
choice.SetChoice();}}}
// Program.cs
using System;
namespace StrategyPattern{
class Program{
static void Main(string[] args){
Console.WriteLine("***Strategy Pattern Demo***\n");
IChoice ic = null; Context cxt = new Context();
//For simplicity, we are considering 2 user inputs only.
for (int i = 1; i <= 2; i++){
Console.WriteLine("\nEnter ur choice(1 or 2)");
string c = Console.ReadLine();
if (c.Equals("1")){ ic = new FirstChoice();}
else{ ic = new SecondChoice();}
cxt.SetChoice(ic); cxt.ShowChoice();}
Console.ReadKey();
  }
 }
}
```

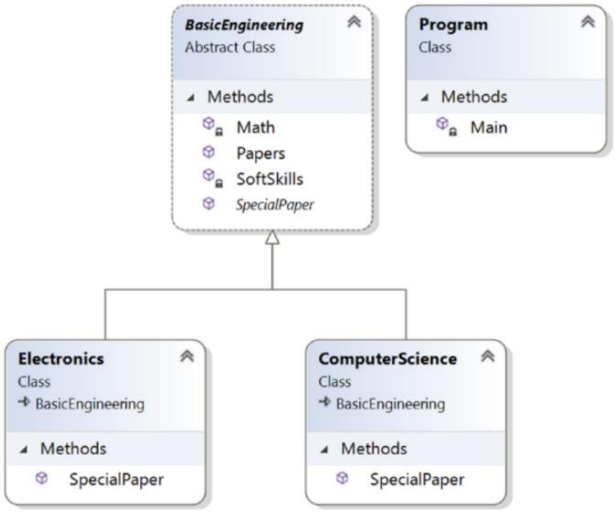

Figure 30: Class diagram for Template Method Pattern

Example code:

```
// BasicEngineering.cs
using System;
namespace TemplateMethodPattern{
public abstract class BasicEngineering{
public void Papers(){
//Common Papers:
Math(); SoftSkills();
//Specialized Paper:
SpecialPaper();}
private void Math(){
Console.WriteLine("Mathematics");}
private void SoftSkills(){
Console.WriteLine("SoftSkills");}
public abstract void SpecialPaper();}}
//ComputerScience.cs
using System;
namespace TemplateMethodPattern{
public class ComputerScience:BasicEngineering{
public override void SpecialPaper(){
Console.WriteLine("Object-Oriented Programming");}}}
//Electronics.cs
using System;
namespace TemplateMethodPattern{
public class Electronics:BasicEngineering{
public override void SpecialPaper(){
Console.WriteLine("Digital Logic and Circuit Theory");}}}
// Program.cs
using System;
```

```
namespace TemplateMethodPattern{
class Program{
static void Main(string[] args){
Console.WriteLine("***Template Method Pattern Demo***\n");
BasicEngineering bs = new ComputerScience();
Console.WriteLine("Computer Sc Papers:");
bs.Papers(); Console.WriteLine();
bs = new Electronics();
Console.WriteLine("Electronics Papers:");
bs.Papers(); Console.ReadLine();
  }
 }
}
```

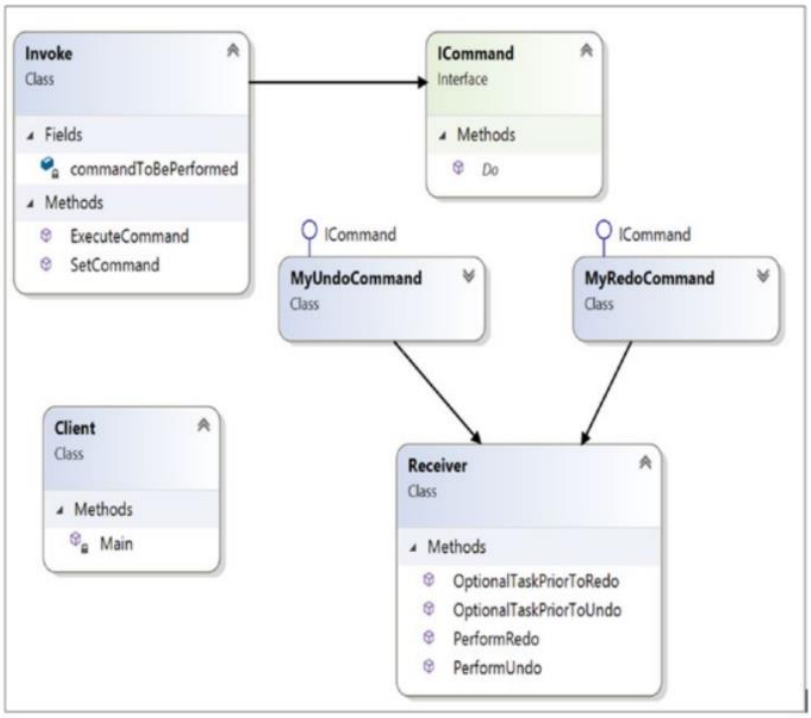

Figure 31: Class diagram for Command Pattern

Example code:

```
using System;
namespace CommandPattern{
public interface ICommand{
void Do();}
public class MyUndoCommand: ICommand{
private Receiver receiver;
public MyUndoCommand(Receiver recv){
receiver=recv;}
```

```
public void Do(){
//Perform any optional task prior to UnDo
receiver.OptionalTaskPriorToUndo();
//Call UnDo in receiver now
receiver.PerformUndo();}}
public class MyRedoCommand : ICommand{
private Receiver receiver;
public MyRedoCommand(Receiver recv){
receiver=recv;}
public void Do(){
//Perform any optional task prior to ReDo
receiver.OptionalTaskPriorToRedo();
//Call ReDo in receiver now
receiver.PerformRedo();}}
//Receiver Class
public class Receiver{
public void PerformUndo(){
Console.WriteLine("Executing-MyUndoCommand");}
public void PerformRedo(){
Console.WriteLine("Executing-MyRedoCommand");}
//Optional method-If you want to perform any prior tasks before
//Undo.
public void OptionalTaskPriorToUndo(){
Console.WriteLine("Executing-Optional Tasks prior to execute undo command");}
//Optional method-If you want to perform any prior tasks before
//Redo.
public void OptionalTaskPriorToRedo(){
Console.WriteLine("Executing-Optional Tasks prior to execute redo command");}}
//Invoker class
public class Invoke{
ICommand commandToBePerformed;
public void SetCommand(ICommand command){
this.commandToBePerformed = command;}
public void ExecuteCommand(){
commandToBePerformed.Do();}}
class Client{
static void Main(string[] args){
Console.WriteLine("***Command Pattern Demo***\n");
/*Client holds both the Invoker and Command Objects*/
Invoke invoker = new Invoke (); Receiver intendedreceiver = new Receiver();
MyUndoCommand undoCmd = new MyUndoCommand(intendedreceiver);
invoker.SetCommand(undoCmd); invoker.ExecuteCommand();
MyRedoCommand redoCmd = new MyRedoCommand(intendedreceiver);
invoker.SetCommand(redoCmd); invoker.ExecuteCommand(); Console.ReadKey();
  }
 }
}
```

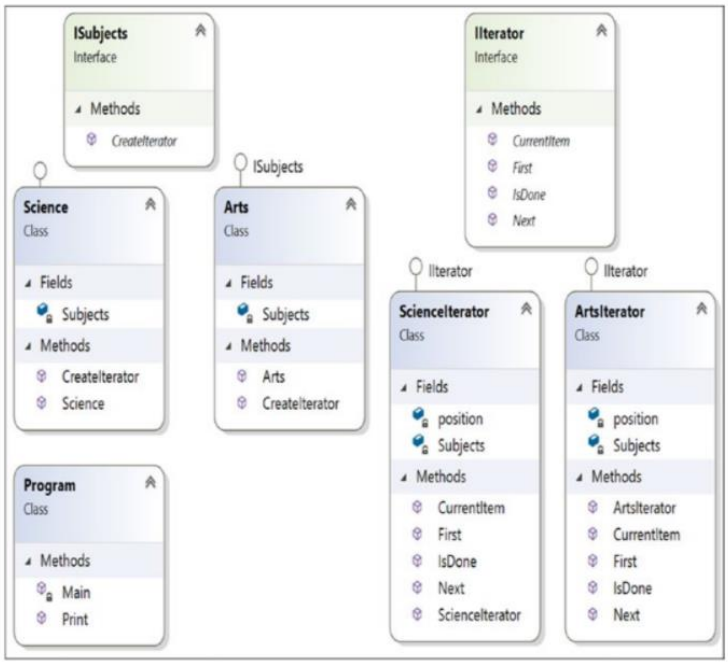

Figure 32: Class diagram for Iterator Pattern

Example code:

```
// ISubjects.cs
using System;
using IteratorPattern.Iterator;
namespace IteratorPattern.Aggregate{
public interface ISubjects{
IIterator CreateIterator();}}
// Science.cs
using System;
using IteratorPattern.Iterator;
using System.Collections.Generic;//For Linked List
namespace IteratorPattern.Aggregate{
public class Science:ISubjects{
private LinkedList<string> Subjects;
public Science(){
Subjects = new LinkedList<string>();
Subjects.AddFirst("Maths");
Subjects.AddFirst("Comp. Sc.");
Subjects.AddFirst("Physics");}
public IIterator CreateIterator(){
return new ScienceIterator(Subjects);}}}
//Arts.cs
using System;
using IteratorPattern.Iterator;
namespace IteratorPattern.Aggregate{
```

```
public class Arts:ISubjects{
private string[] Subjects;
public Arts(){
Subjects = new[] {"Bengali", "English"};}
public IIterator CreateIterator(){
return new ArtsIterator(Subjects);}}}
using System;
namespace IteratorPattern.Iterator{
public interface IIterator{
void First();//Reset to first element
string Next();//Get next element
bool IsDone();//End of collection check
string CurrentItem();//Retrieve Current Item
}
}
// ScienceIterator.cs
using System;
using System.Collections.Generic;
using System.Linq; //For: Subjects.ElementAt(position++);
namespace IteratorPattern.Iterator{
public class ScienceIterator:IIterator{
private LinkedList<string> Subjects;
private int position;
public ScienceIterator(LinkedList<string> subjects){
this.Subjects = subjects; position = 0;}
public void First(){
position = 0;}
public string Next(){
return Subjects.ElementAt(position++);}
public bool IsDone(){
if (position < Subjects.Count){
return false;}
else{
return true;}}
public string CurrentItem(){
return Subjects.ElementAt(position);}}}
// ArtsIterator.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace IteratorPattern.Iterator{
public class ArtsIterator:IIterator{
private string[] Subjects;
private int position;
public ArtsIterator(string[] subjects){
this.Subjects = subjects; position = 0;}
```

```
public void First(){
position = 0;}
public string Next(){
return Subjects[position++];}
public bool IsDone(){
if( position < Subjects.Length){
return false;}
else{
return true;}}
public string CurrentItem(){
return Subjects[position];}}}
// Program.cs
using System;
using IteratorPattern.Aggregate;
using IteratorPattern.Iterator;
namespace IteratorPattern{
class Program{
static void Main(string[] args){
Console.WriteLine("***Iterator Pattern Demo***");
ISubjects ScienceSubjects = new Science();
ISubjects ArtsSubjects = new Arts();
IIterator IteratorForScience = ScienceSubjects.CreateIterator();
IIterator IteratorForArts = ArtsSubjects.CreateIterator();
Console.WriteLine("\nScience subjects :"); Print(IteratorForScience);
Console.WriteLine("\nArts subjects :");
Print(IteratorForArts); Console.ReadLine();}
public static void Print(IIterator iterator){
while (!iterator.IsDone()){
Console.WriteLine(iterator.Next());}
  }
 }
}
```

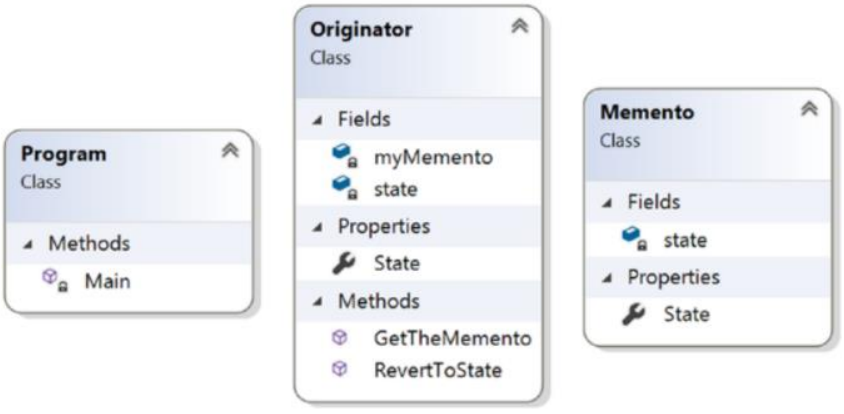

Figure 33: Class diagram for Memento Pattern

Example code:

```
using System;
namespace MementoPattern
{
/// <summary>
/// Memento class
/// </summary>
class Memento{
private string state;
public string State{
get{return state;}
set{state = value;}}}
/// <summary>
/// Originator class
/// </summary>
class Originator{
private string state;
Memento myMemento;
//public Originator()
//{
// //this.State = "Initial state";
//}
public string State{
get { return state; }
set{state = value;
Console.WriteLine("Current State : {0}", state);}}
// Originator will supply the memento in respond to caretaker's
//request
public Memento GetTheMemento(){
//Creating a memento with the current state
myMemento = new Memento();
myMemento.State = state;
return myMemento;}
// Back to old state (Restore)
public void RevertToState(Memento previousMemento){
Console.WriteLine("Restoring to previous state...");
this.state = previousMemento.State;
Console.WriteLine("Current State : {0}", state);}}
/// <summary>
/// </summary>
class Program{
static void Main(string[] args){
Console.WriteLine("***Memento Pattern Demo***\n");
//Originator is initialized with a state
Originator originatorObject = new Originator();
```

```
Memento mementoObject;
originatorObject.State = "Initial state";
mementoObject = originatorObject.GetTheMemento();
//Making a new state
originatorObject.State="Intermediary state";
// Restore to the previous state
originatorObject.RevertToState(mementoObject);
// Wait for user's input
Console.ReadKey();
  }
 }
}
```

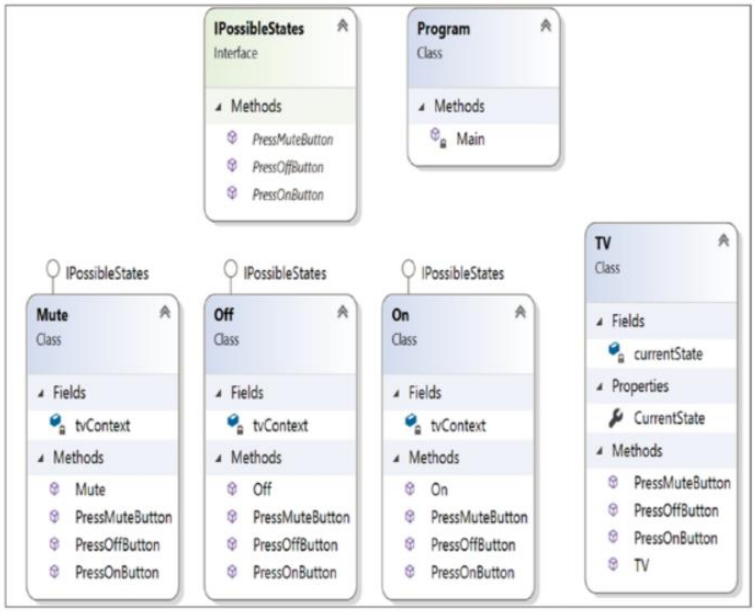

Figure 34: Class diagram for State Pattern

Example code:

```
using System;
namespace StatePattern{
interface IPossibleStates{
void PressOnButton(TV context);
void PressOffButton(TV context);
void PressMuteButton(TV context);}
class Off : IPossibleStates{ TV tvContext;
//Initially we'll start from Off state
public Off(TV context){
Console.WriteLine(" TV is Off now.");
this.tvContext = context;}
```

```
//Users can press any of these buttons at this state-On, Off or Mute
//TV is Off now, user is pressing On button
public void PressOnButton(TV context){
Console.WriteLine("You pressed On button. Going from Off to On state");
tvContext.CurrentState = new On(context);}
//TV is Off already, user is pressing Off button again
public void PressOffButton(TV context){
Console.WriteLine("You pressed Off button. TV is already in Off state");}
//TV is Off now, user is pressing Mute button
public void PressMuteButton(TV context){
Console.WriteLine("You pressed Mute button. TV is already in Off state, so Mute operation will not wo
rk.");
}
}
class On : IPossibleStates{
TV tvContext;
public On(TV context){
Console.WriteLine("TV is On now.");
this.tvContext = context;}
//Users can press any of these buttons at this state-On, Off or Mute
//TV is On already, user is pressing On button again
public void PressOnButton(TV context){
Console.WriteLine("You pressed On button. TV is already in On state.");}
//TV is On now, user is pressing Off button
public void PressOffButton(TV context){
Console.WriteLine("You pressed Off button. Going from On to Off state.");
tvContext.CurrentState = new Off(context);}
//TV is On now, user is pressing Mute button
public void PressMuteButton(TV context){
Console.WriteLine("You pressed Mute button. Going from On to Mute mode.");
tvContext.CurrentState = new Mute(context);}}
class Mute : IPossibleStates{
TV tvContext;
public Mute(TV context){
Console.WriteLine("TV is in Mute mode now.");
this.tvContext = context;}
//Users can press any of these buttons at this state-On, Off or Mute
//TV is in mute, user is pressing On button
public void PressOnButton(TV context){
Console.WriteLine("You pressed On button. Going from Mute mode to On state.");
tvContext.CurrentState = new On(context);}
//TV is in mute, user is pressing Off button
public void PressOffButton(TV context){
Console.WriteLine("You pressed Off button. Going to Mute mode to Off state.");
tvContext.CurrentState = new Off(context);}
//TV is in mute already, user is pressing mute button again
public void PressMuteButton(TV context){
```

```
Console.WriteLine(" You pressed Mute button. TV is already in Mute mode.");}}
class TV{
private IPossibleStates currentState;
public IPossibleStates CurrentState{
//get;set;//Not working as expected
get{return currentState;}
/*Usually this value will be set by the class that
implements the interface "IPossibleStates"*/
set{currentState = value;}}
public TV(){
this.currentState = new Off(this);}
public void PressOffButton(){
currentState.PressOffButton(this);//Delegating the state
}
public void PressOnButton(){
currentState.PressOnButton(this);//Delegating the state
}
public void PressMuteButton(){
currentState.PressMuteButton(this);//Delegating the state
}}
class Program{
static void Main(string[] args){
Console.WriteLine("***State Pattern Demo***\n");
//Initially TV is Off
TV tv = new TV();
Console.WriteLine("User is pressing buttons in the following sequence:");
Console.WriteLine("Off->Mute->On->On->Mute->Mute->Off\n");
//TV is already in Off state
tv.PressOffButton();
//TV is already in Off state, still pressing the Mute button
tv.PressMuteButton();
//Making the TV on
tv.PressOnButton();
//TV is already in On state, pressing On button again
tv.PressOnButton();
//Putting the TV in Mute mode
tv.PressMuteButton();
//TV is already in Mute, pressing Mute button again
tv.PressMuteButton();
//Making the TV off
tv.PressOffButton();
// Wait for user
Console.Read();
  }
 }
}
```

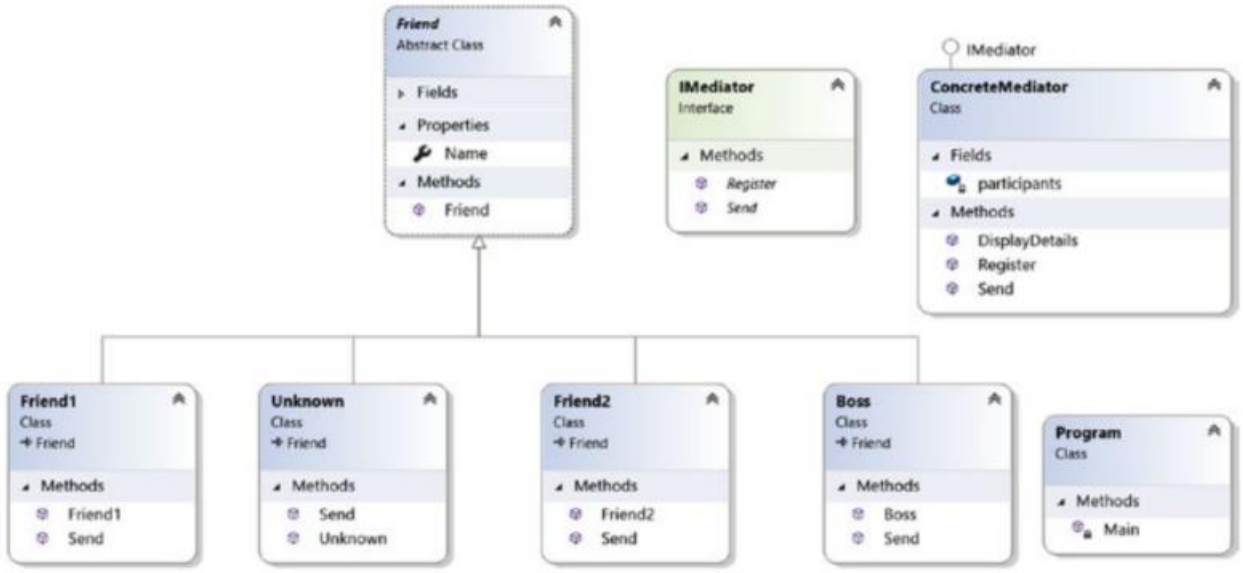

Figure 35: Class diagram for Mediator Pattern

Example code:

```
using System;
using System.Collections.Generic;
namespace MediatorPattern{
interface IMediator{
void Register(Friend friend);
void Send(Friend friend, string msg);}
// ConcreteMediator
class ConcreteMediator : IMediator{
//private Friend friend1,friend2,boss;
List<Friend> participants = new List<Friend>();
public void Register(Friend friend){
participants.Add(friend);}
public void DisplayDetails(){
Console.WriteLine("At present, registered Participants are:");
foreach (Friend friend in participants)
{Console.WriteLine("{0}", friend.Name);}}
public void Send(Friend friend, string msg){
if (participants.Contains(friend)){
Console.WriteLine(String.Format("[{0}] posts: {1} Last message posted {2}", friend.Name, msg, DateTim
e.Now));
System.Threading.Thread.Sleep(1000);}
else{
Console.WriteLine("An outsider named {0} trying to send some messages", friend.Name);
}}}
// Friend
abstract class Friend{
```

```
protected IMediator mediator;
private string name;
public string Name{
get { return name; }
set { name = value; }}
// Constructor
public Friend(IMediator mediator){
this.mediator = mediator;}}// Friend1-first participant
class Friend1 : Friend{
public Friend1(IMediator mediator, string name) : base(mediator){
this.Name = name;}
public void Send(string msg){
mediator.Send(this,msg);}}
// Friend2-Second participant
class Friend2 : Friend{
// Constructor
public Friend2(IMediator mediator, string name) : base(mediator){
this.Name = name;}
public void Send(string msg){
mediator.Send(this, msg);}}
/* Friend3-Third Participant.He is the boss.*/
class Boss : Friend{
// Constructor
public Boss(IMediator mediator, string name) : base(mediator){
this.Name = name;}
public void Send(string msg){
mediator.Send(this, msg);}}
class Unknown: Friend{
// Constructor
public Unknown(IMediator mediator, string name): base(mediator){
this.Name = name;}
public void Send(string msg){
mediator.Send(this, msg);}}
class Program{
static void Main(string[] args){
Console.WriteLine("***Mediator Pattern Demo***\n");
ConcreteMediator mediator = new ConcreteMediator();
Friend1 Amit = new Friend1(mediator, "Amit");
Friend2 Sohel = new Friend2(mediator, "Sohel");
Boss Raghu = new Boss(mediator, "Raghu");
//Registering participants
mediator.Register(Amit);
mediator.Register(Sohel);
mediator.Register(Raghu);
//Displaying the participant's list
mediator.DisplayDetails();
Console.WriteLine("Communication starts among participants...");
```

```
Amit.Send("Hi Sohel,can we discuss the mediator pattern?");
Sohel.Send("Hi Amit,Yup, we can discuss now.");
Raghu.Send("Please get back to work quickly.");
//An outsider/unknown person tries to participate
Unknown unknown = new Unknown(mediator, "Jack");
unknown.Send("Hello Guys..");
// Wait for user
Console.Read();
  }
 }
}
```

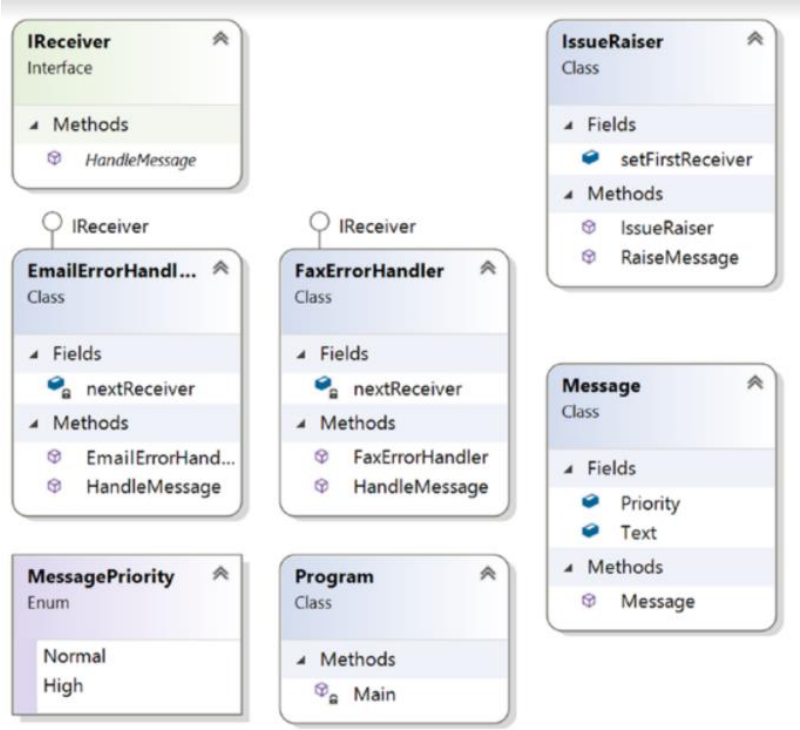

Figure 36:Class diagram for Chain of Responsibility Pattern

Example code:

```
using System;
namespace ChainOfResponsibilityPatternModified{
public enum MessagePriority
{Normal,High}
public class Message{
public string Text;
public MessagePriority Priority;
public Message(string msg, MessagePriority p){
Text = msg; this.Priority = p;}}
```

```
public interface IReceiver{
bool HandleMessage(Message message);}
public class IssueRaiser{
public IReceiver setFirstReceiver;
public IssueRaiser(IReceiver firstReceiver){
this.setFirstReceiver = firstReceiver;}
public void RaiseMessage(Message message){
if (setFirstReceiver != null)
setFirstReceiver.HandleMessage(message);}}
public class FaxErrorHandler : IReceiver{
private IReceiver nextReceiver;
public FaxErrorHandler(IReceiver nextReceiver){
this.nextReceiver = nextReceiver;}
public bool HandleMessage(Message message){
if (message.Text.Contains("Fax")){
Console.WriteLine("FaxErrorHandler processed {0} priority issue: {1}", message.Priority, message.Text
);
return true;}else{
if (nextReceiver != null)
nextReceiver.HandleMessage(message);}
return false;}}
public class EmailErrorHandler : IReceiver{
private IReceiver nextReceiver;
public EmailErrorHandler(IReceiver nextReceiver){
this.nextReceiver = nextReceiver;}
public bool HandleMessage(Message message){
if (message.Text.Contains("Email")){
Console.WriteLine("EmailErrorHandler processed {0} priority issue: {1}", message.Priority, message.Te
xt);
return true;}else{
if (nextReceiver != null) nextReceiver.HandleMessage(message);}
return false;}}
class Program{
static void Main(string[] args)
{Console.WriteLine("\n ***Chain of Responsibility Pattern Demo***\n");
IReceiver faxHandler, emailHandler;
//End of chain
emailHandler = new EmailErrorHandler(null);
//fax handler is placed before email handler
faxHandler = new FaxErrorHandler(emailHandler);
//Starting point:IssueRaiser will raise issues and set the first //handler
IssueRaiser raiser = new IssueRaiser(faxHandler);
Message m1 = new Message("Fax is reaching late to the destination.", MessagePriority.Normal);
Message m2 = new Message("Emails are not raching to destinatinations.", MessagePriority.High);
Message m3 = new Message("In Email, CC field is disabled always.", MessagePriority.Normal);
Message m4 = new Message("Fax is not reaching destination.",
MessagePriority.High); raiser.RaiseMessage(m1);
```

```
raiser.RaiseMessage(m2); raiser.RaiseMessage(m3);
raiser.RaiseMessage(m4); Console.ReadKey();
  }
 }
}
```

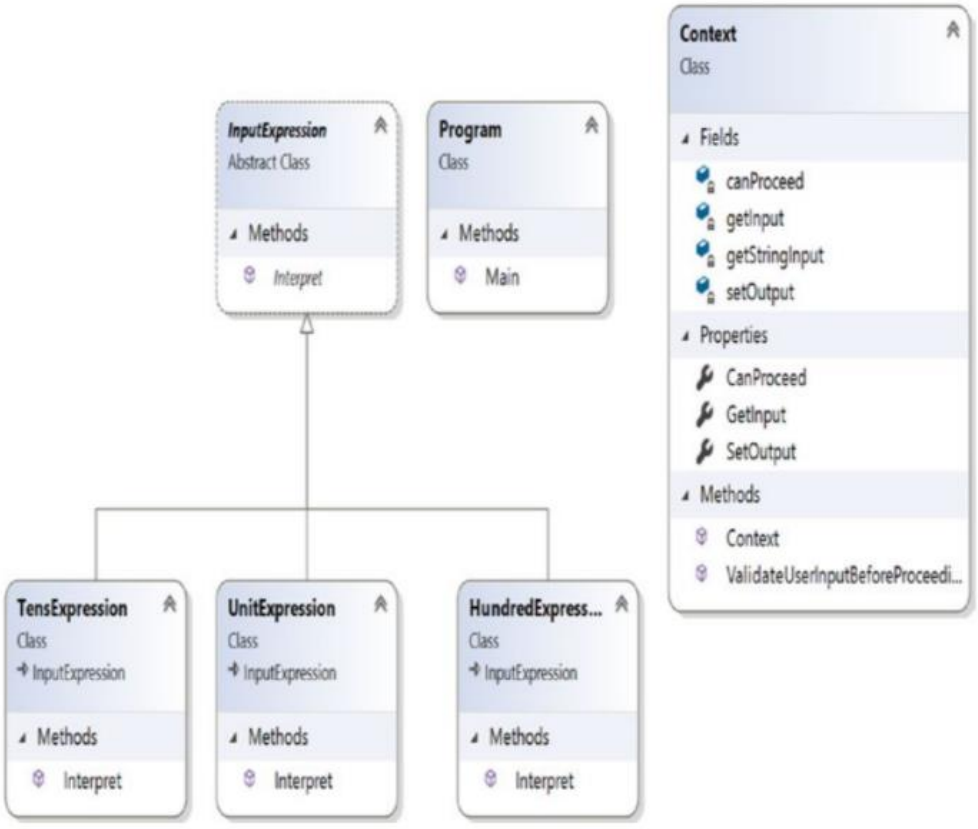

Figure 37: Class diagram for Interpreter Pattern

Example code:

```
using System;
using System.Collections.Generic;
namespace InterpreterPattern{
public class Context{
//We will interpret an integer
private int getInput;
private string getStringInput;
//We are printing it in the word form i.e. in String representation
private string setOutput;
//Flag-whether it is a valid input or not
private bool canProceed = false;
public bool CanProceed{
```

```
get {return canProceed;}}
//Using properties to get the input(readonly)
public int GetInput{get {return getInput;}
//set {input = value;}
}
//Using properties to get and set output
public string SetOutput{get {return setOutput;} set {setOutput = value;}}
//Our constructor
public Context(string input){this.getStringInput = input;}
//We'll check whether it is a valid input that lies between 100 and
//999
public int ValidateUserInputBeforeProceedings(string inputString){
if (int.TryParse(inputString, out getInput)){
Console.WriteLine("You have entered {0}", getInput);
//Some basic validations
if ((getInput < 100) || (getInput > 999)){
Console.WriteLine("Please enter a number between 100 and 999 and try again.");
//Just returning a 4-digit negative number to indicate a wrong input
return -9999;}}
canProceed = true; return getInput;}}
//abstract class-will hold the common code
abstract class InputExpression{
public abstract void Interpret(Context context);}
class HundredExpression : InputExpression{
public override void Interpret(Context context){
if (context.CanProceed){int hundreds = context.GetInput / 100;
switch (hundreds){
case 1:context.SetOutput += "One Hundred";break;
case 2:context.SetOutput += "Two Hundred";break;
case 3:context.SetOutput += "Three Hundred";break;
case 4:context.SetOutput += "Four Hundred";break;
case 5:context.SetOutput += "Five Hundred";break;
case 6:context.SetOutput += "Six Hundred";break;
case 7:context.SetOutput += "Seven Hundred";break;
case 8:context.SetOutput += "Eight Hundred";break;
case 9:context.SetOutput += "Nine Hundred";break;
default:context.SetOutput += "*";break;}}}}
class TensExpression : InputExpression{
public override void Interpret(Context context){
if (context.CanProceed){int tens = context.GetInput % 100;
//Process further by dividing it by 10
tens = tens / 10;switch (tens)
{case 1:context.SetOutput += "One Ten and";break;
case 2:context.SetOutput += "Twenty";break;
case 3:context.SetOutput += "Thirty";break;
case 4:context.SetOutput += "Forty";break;
case 5:context.SetOutput += "Fifty";break;
```

```
case 6:context.SetOutput += "Sixty";break;
case 7:context.SetOutput += "Seventy";break;
case 8:context.SetOutput += "Eighty";break;
case 9:context.SetOutput += "Ninety";break;
default:context.SetOutput += String.Empty;break;}}}}
class UnitExpression : InputExpression{
public override void Interpret(Context context){if (context.CanProceed){
int units = context.GetInput % 100;
//Process further to get the unit digit
units = units % 10;switch (units)
{case 1:context.SetOutput += "One";break;
case 2:context.SetOutput += "Two";break;
case 3:context.SetOutput += "Three";break;
case 4:context.SetOutput += "Four";break;
case 5:context.SetOutput += "Five";break;
case 6:context.SetOutput += "Six";break;
case 7:context.SetOutput += "Seven";break;
case 8:context.SetOutput += "Eight";break;
case 9:context.SetOutput += "Nine";break;
default:context.SetOutput += String.Empty;break;}}}}
//Client Class
class Program{
public static void Main(String[] args){
Console.WriteLine("***Interpreter Pattern Demo***\n");
string inputString;
//int userInput;
Console.WriteLine("Enter a 3 digit number only (i.e. 100 to 999)");
inputString = Console.ReadLine();
//Context context = new Context(userInput);
Context context = new Context(inputString);
//Some basic validations before we proceed
//Checking whether we can parse the string as an integer
if (context.ValidateUserInputBeforeProceedings(inputString) != -9999){
// Build the 'parse tree'
List<InputExpression> expTree = new List<InputExpression>();
expTree.Add(new HundredExpression());
expTree.Add(new TensExpression());
expTree.Add(new UnitExpression());
// Interpret the valid input
foreach (InputExpression inputExp in expTree){
inputExp.Interpret(context);}
Console.WriteLine("Original Input {0} is interpreted as {1}", context.GetInput, context.SetOutput);
} Console.ReadLine();
  }
 }
}
```

REFERENCES

[1] Dan Clark, 2020. *Beginning C# Object-Oriented Programming*. 2nd ed. [ebook] Available at: <https://drive.google.com/drive/folders/11Ad6JnmaMWjmAgen0d4fsqWnUQSIqrus> [Accessed 28 November 2020].

[2] Tam Phan, 2020. *Unit 20: Advanced Programming (1651)*. [lecture] Available at: <https://drive.google.com/drive/folders/1LWDEnxIwM0Ru6NS3nOyQ1rs90ekD3G38> [Accessed 28 November 2020]

[3] Manish Agrahari, 2020. *Introduction To Object Oriented Programming Concepts In C#*. [online] C-sharpcorner.com. Available at: <https://www.c-sharpcorner.com/UploadFile/mkagrahari/introduction-to-object-oriented-programming-concepts-in-C-Sharp/#:~:text=A%20class%20is%20the%20core,perform%20operations%20on%20the%20data.> [Accessed 28 November 2020].

[4] W3schools.com. 2020. *C# Classes And Objects*. [online] Available at: <https://www.w3schools.com/cs/cs_classes.asp> [Accessed 28 November 2020].

[5] K9, 2017. *[Khóa Học Lập Trình Hướng Đối Tượng C#] - Bài 2: Class | Howkteam*. [video] Available at: <https://www.youtube.com/watch?v=pXcMdI3LVEE&list=PL33lvabfss1zRgaWBcC__Bnt5AOSRfU71&index=2> [Accessed 28 November 2020].

[6] GeeksforGeeks. 2020. *C# | Class And Object - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-class-and-object/?ref=lbp> [Accessed 28 November 2020].

[7] GeeksforGeeks. 2020. *C# | Inheritance - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-inheritance/?ref=lbp> [Accessed 28 November 2020].

[8] GeeksforGeeks. 2020. *C# | Encapsulation - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-encapsulation/?ref=lbp> [Accessed 28 November 2020].

[9] GeeksforGeeks. 2020. *C# | Abstraction - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-abstraction/?ref=lbp> [Accessed 28 November 2020].

[10] K9, 2017. *[Khóa Học Lập Trình Hướng Đối Tượng C#] - Bài 5: Kế Thừa | Howkteam*. [video] Available at: <https://www.youtube.com/watch?v=tgxT3xRdfvU&list=PL33lvabfss1zRgaWBcC__Bnt5AOSRfU71&index=5> [Accessed 28 November 2020].

[11] K9, 2017. *[Khóa Học Lập Trình Hướng Đối Tượng C#] - Bài 6: Đa Hình | Howkteam*. [video] Available at: <https://www.youtube.com/watch?v=SfFAk5LEW0I&list=PL33lvabfss1zRgaWBcC__Bnt5AOSRfU71&index=6> [Accessed 28 November 2020].

[12] Www-acad.sheridanc.on.ca. 2020. *Class Relationships*. [online] Available at: <http://www-acad.sheridanc.on.ca/~jollymor/prog24178/oop6.html> [Accessed 28 November 2020].

[13] Muhammad Arshad, 2019. *Type Of Relationships In Object Oriented Programming (OOP)*. [online] marshad.pw. Available at: <https://www.marshad.pw/relationships-oop/> [Accessed 28 November 2020].

[14] Visual-paradigm.com. 2020. *UML Association Vs Aggregation Vs Composition*. [online] Available at: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/#:~:text=Aggregation%20and%20Composition%20are%20subsets,exist%20independently%20of%20the%20parent.> [Accessed 28 November 2020].

[15] Mahesh Alle, 2020. *Singleton Design Pattern In C#*. [online] C-sharpcorner.com. Available at: <https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/> [Accessed 4 December 2020].

[16] En.wikipedia.org. 2020. *Decorator Pattern*. [online] Available at: <https://en.wikipedia.org/wiki/Decorator_pattern#Structure> [Accessed 4 December 2020].

[17] En.wikipedia.org. 2020. *Iterator Pattern*. [online] Available at: <https://en.wikipedia.org/wiki/Iterator_pattern#Structure> [Accessed 4 December 2020].

[18] Wrapper, 2020. *Decorator*. [online] Refactoring.guru. Available at: <https://refactoring.guru/design-patterns/decorator#:~:text=Decorator%20is%20a%20structural%20design,objects%20that%20contain%20the%20behaviors.> [Accessed 4 December 2020].

[19] Vaskaran Sarcar and Priya Shimanthoor, 2018. *Design Patterns In C#*. [ebook] Available at: <https://drive.google.com/drive/folders/11Ad6JnmaMWjmAgen0d4fsqWnUQSIqrus> [Accessed 4 December 2020].