UNIVERSITY of GREENWICH

Alliance with FPT Education

ADVANCED PROGRAMMING

REPORT ASSIGNMENT 1

MICROSOFT OFFICE USER
STUDENT ID: GCD18457 & GCD17175
Assessor Name: HOANG NHU VINH

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	Tran Quang Huy Nguyen Van Hieu	**Student ID**	GCD18457 GCD17175
Class	GCD0604	**Assessor name**	Hoang Nhu Vinh

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	Huy, Hieu

Grading grid

P1	P2	M1	M2	D1	D2

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

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF PICTURES

INTRODUCTION

What does the Report aim to do?

Features of programming languages that are considered advanced are used to develop software that is efficient; it can affect the performance of an application as well as the readability and extensibility of the code, improving productivity and therefore reducing cost. Many commercial applications available today, whether for productivity or entertainment, will have used one or more design pattern in their development. A design pattern is a description of how to solve a problem that can be used in many different situations and can help deepen the understanding of object-orientated programming and help improve software design and reusability. That's why we wrote this document so readers can understand how to create effective apps, improve app performance and etc.

What are the goals of the report?

- ✓ Understand about 4 characteristics of OOP includes: Abstraction, Encapsulation, Polymorphism and Inheritance.
- ✓ Understand about OOP and Design Pattern
- ✓ Understand relationship between OOP and Design Pattern.

PART I. Examine the characteristics of the object-oriented paradigm as well as the various class relationship and Design and build class diagrams using a UML tool.

Firstly, we planning a project about create “An application about **School System**”.

1. Class and Object

Firstly, we create class **Person** includes: fields and method for application School System.

- Class Diagram of **Class and Object:**

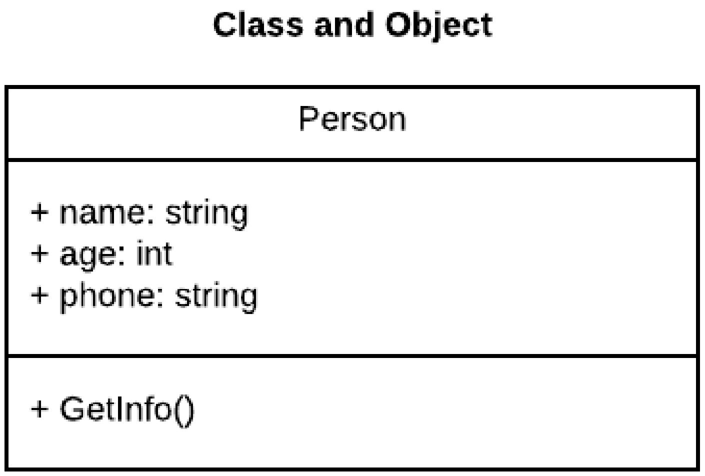


Figure 1. Class and Object class diagram

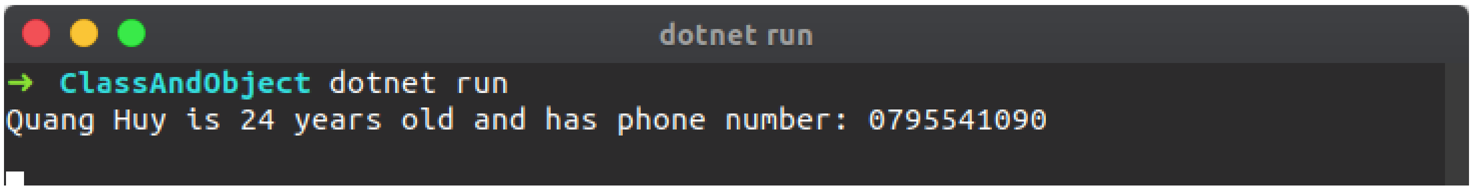
- **Source code of Class and Object:**

```
class Persons
    {
        public string name;
        public int age;

        public string phone;

        public Persons()
        {
            name = "Empty";
            age = 0;
        }

        public Persons(string Name)
        {
            name = Name;
            age = 0;
            phone = "0";
        }
        public Persons(string Name, int Age)
        {
            name = Name;
            age = Age;
            phone = "0";
        }
        public Persons(string Name, int Age, string Phone)
        {
            name = Name;
            age = Age;
            phone = Phone;
        }
       public void setName(string Name)
        {
            name = Name;
        }

        public void setAge(int Age)
        {
            age = Age;
        }

        public void setPhone(string Phone)
        {
            phone = Phone;
        }

        public void getInfo()
        {
            Console.WriteLine($"{name} is {age} years old and has phone number: {phone}");
        }
```

- **Code Main function of Class and Object:**

```
class Program
    {
        static void Main(string[] args)
        {
            Persons myPersons = new Persons();
            myPersons.setName("Quang Huy");
            myPersons.setAge(24);
            myPersons.setPhone("0795541090");
            myPersons.getInfo();

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

- **Result:**



```
dotnet run
→ ClassAndObject dotnet run
Quang Huy is 24 years old and has phone number: 0795541090
```

Picture 1. Result of Class and Object code

2. Encapsulation

In c#, Encapsulation is a process of binding the data members and member functions into a single unit.

Generally, in c# the encapsulation is used to prevent an alteration of code (data) accidentally from the outside of functions. In c#, by defining the class fields with properties we can protect the data from accidental corruption.

- **Class Diagram of Encapsulation:**

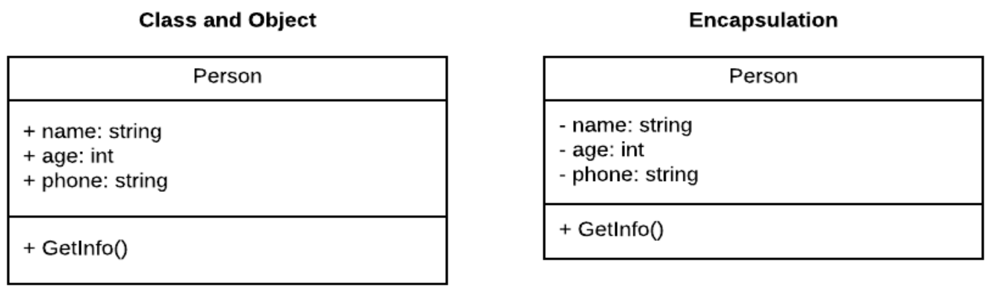


Figure 2. Encapsulation class diagram

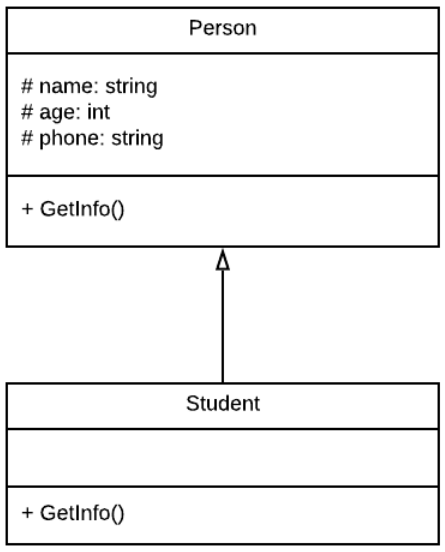


Figure 3. Protected in Encapsulation class diagram

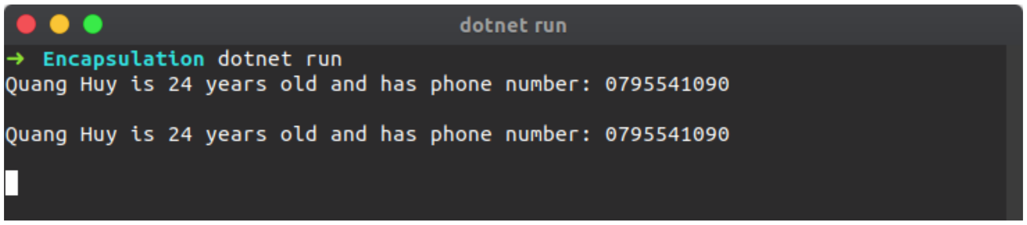
- **Source code of Encapsulation:**

```
class Persons
    {
        private string name;
        private int age;

        private string phone;

        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public int Age
        {
            get { return age; }
            set { age = value; }
        }

        public string Phone
        {
            get { return phone; }
            set { phone = value; }
        }
        public void getInfo()
        {
            Console.WriteLine($"{Name} is {Age} years old and has phone number: {Phone}");
        }
    }
```

- **Code Main function of Encapsulation:**

```
class Program
    {
        static void Main(string[] args)
        {
            Persons myPersons = new Persons();
            myPersons.Name = "Quang Huy";
            myPersons.Age = 24;
            myPersons.Phone = "0795541090";
            Console.Write($"{myPersons.Name} is {myPersons.Age} years old and has 
phone number: {myPersons.Phone}");
            Console.WriteLine();
            myPersons.getInfo();

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

- **Result:**



```
➜ Encapsulation dotnet run
Quang Huy is 24 years old and has phone number: 0795541090

Quang Huy is 24 years old and has phone number: 0795541090

```

Picture 2. Result of Encapsulation code

If you observe the above code, we defined a variable with private access modifiers and exposing those variables in public way by using properties get and set accessors. In case, if you want to make any modifications to the defined variables, then we can make it by using properties with get and set accessors.

3. Inheritance

In c#, Inheritance is a one of the primary concepts of object-oriented programming (OOP) and it is used to inherit the properties from one class (base) to another (child) class.

The inheritance will enable us to create a new class by inheriting the properties from other classes to reuse, extend and modify the behaviour of other class members based on our requirements.

In c# inheritance, the class whose members are inherited is called a base (parent) class and the class that inherits the members of base (parent) class is called a derived (child) class.

- **Multi-Level Inheritance**

Generally, c# supports only single inheritance that means a class can only inherit from one base class. However, in c# the inheritance is transitive and it allows you to define a hierarchical inheritance for a set of types and it is called a multi-level inheritance.

For example, suppose if class C is derived from class B, and class B is derived from class A, then the class C inherits the members declared in both class B and class A.

- **Multiple Inheritance**

As discussed, c# supports only single inheritance that means a class can only inherit from one base class. In case, if we try to inherit a class from multiple base classes, then we will get a compile time error.

- **Class Diagram of Inheritance:**

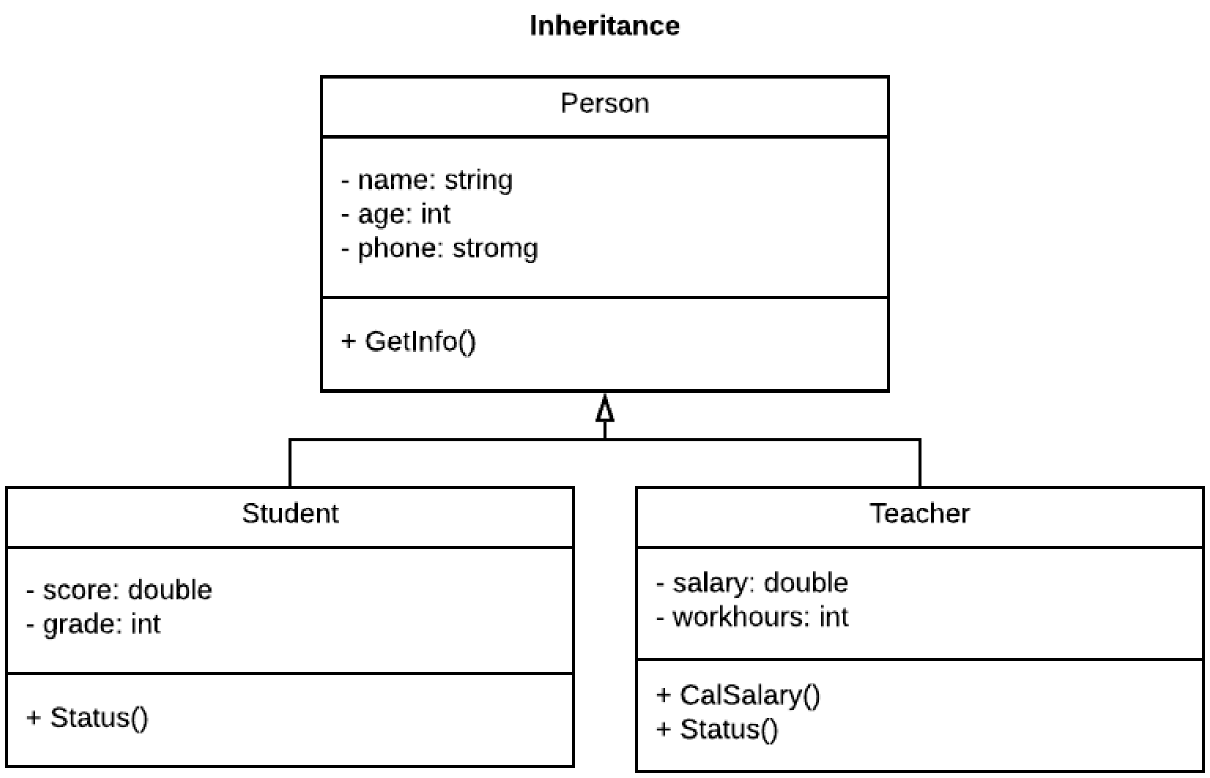


Figure 4. Inheritance class diagram

Source code of Inheritance:

```
class Persons
{
    private string name;
    private int age;

    private string phone;

    public virtual string Name
    {
        get { return name; }
        set { name = value; }
    }

    public virtual int Age
    {
        get { return age; }
        set { age = value; }
    }

    public virtual string Phone
    {
        get { return phone; }
        set { phone = value; }
    }

    public Persons(string name, int age, string phone)
    {
        this.Name = name;
        this.Age = age;
        this.Phone = phone;
    }
}
```

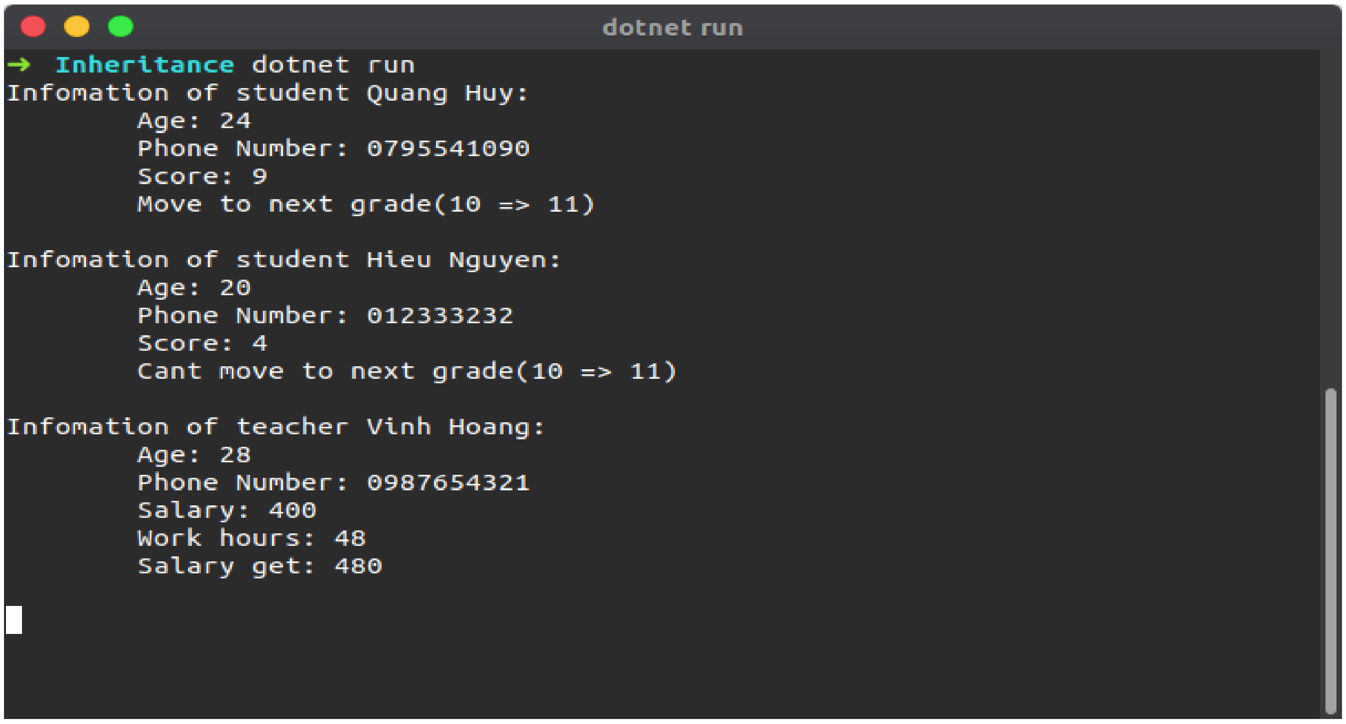
```
class Student : Persons
    {
        private double score;
        private int grade;

        public Student(string name, int age, string phone) : base(name, age, phone)
        {
            score = 0;
            grade = 0;
        }

        public Student(string name, int age, string phone, double score) : base(name, age, phone)
        {
            this.score = score;
        }

        public Student(string name, int age, string phone, double score, int grade) : base(name,
age, phone)
        {
            this.score = score;
            this.grade = grade;
        }

        public void status()
        {
            if (score >= 5)
                Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tScore: {score}\n\tMove to next grade({grade} => {grade + 1})\n\n");
            else Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tScore: {score}\n\tCant move to next grade({grade} => {grade + 1})\n\n");
        }
    }
```

```
class Teacher : Persons
    {
        private double salary;
        private int workhours;

        public Teacher(string name, int age, string phone) : base(name, age, phone)
        {
            salary = 0;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary) : base(name, age,
phone)
        {
            this.salary = salary;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary, int workhours) :
base(name, age, phone)
        {
            this.salary = salary;
            this.workhours = workhours;
        }

        public int CalSalary(int workhours)
        {
            return 10 * workhours;
        }

        public void status()
        {
            Console.Write($"Infomation of teacher {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tSalary: {salary}\n\tWork hours: {workhours}\n\tSalary get:
{CalSalary(workhours)}\n");
        }
    }
```

Code Main function of Inheritance:

```
class Program
    {
        static void Main(string[] args)
        {
            Student myStudent1 = new Student("Quang Huy", 24, "0795541090", 9, 10);
            Student myStudent2 = new Student("Hieu Nguyen", 20, "012333232", 4, 10);
            Teacher myTeacher1 = new Teacher("Vinh Hoang", 28, "0987654321", 400, 48);

            myStudent1.status();
            myStudent2.status();
            myTeacher1.status();

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

Result:



```
→ Inheritance dotnet run
Infomation of student Quang Huy:
        Age: 24
        Phone Number: 0795541090
        Score: 9
        Move to next grade(10 => 11)

Infomation of student Hieu Nguyen:
        Age: 20
        Phone Number: 012333232
        Score: 4
        Cant move to next grade(10 => 11)

Infomation of teacher Vinh Hoang:
        Age: 28
        Phone Number: 0987654321
        Salary: 400
        Work hours: 48
        Salary get: 480
```

Picture 3. Result of Inheritance code

4. Polymorphism

In c#, Polymorphism means providing an ability to take more than one form and it’s a one of the main pillar concepts of object-oriented programming, after encapsulation and inheritance.

Generally, the polymorphism is a combination of two words, one is poly and another one is morphs. Here poly means “multiple” and morphs means “forms” so polymorphism means many forms.

In c#, polymorphism provides an ability for the classes to implement a different method that are called through the same name and it also provides an ability to invoke the methods of derived class through base class reference during runtime based on our requirement

In c#, we have a two different kind of polymorphisms available, those are

- Compile Time Polymorphism
- Run Time Polymorphism

Compile Time Polymorphism

In c#, Compile Time Polymorphism means defining a multiple method with same name but with different parameters. By using compile time polymorphism, we can perform a different task with same method name by passing different parameters.

In c#, the compile time polymorphism can be achieved by using method overloading and it is also called as early binding or static binding.

```
public class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b +
c);
    }
}
```

If you observe above "Calculate" class, we defined two methods with same name (**AddNumbers**) but with different input parameters to achieve method overloading, this is called a compile time polymorphism in c#.

Run Time Polymorphism

In c#, Run Time Polymorphism means overriding a base class method in derived class by creating a similar function and this can be achieved by using override & virtual keywords along with inheritance principle.

By using run time polymorphism, we can override a base class method in derived class by creating a method with same name and parameters to perform a different task.

In c#, the run time polymorphism can be achieved by using method overriding and it is also called as late binding or dynamic binding.

```
// Base Class
public class Users
{
    public virtual void GetInfo()
    {
        Console.WriteLine("Base Class");
    }
}
// Derived Class
public class Details : Users
{
    public override void GetInfo()
    {
        Console.WriteLine("Derived Class");
    }
}
```

If you observe above code snippet, we created two classes ("Users", "Details") and the derived class (Details) is inheriting the properties from base class (Users) and we are overriding the base class method **GetInfo** in derived class by creating a same function to achieve method overriding, this is called a run time polymorphism in c#.

Here, we defined a **GetInfo** method with virtual keyword in base class to allow derived class to override that method using override keyword.

Generally, only the methods with virtual keyword in base class are allowed to override in derived class using override keyword. To know more about it, check this Method Overriding in C#.

Class Diagram of Polymorphism:

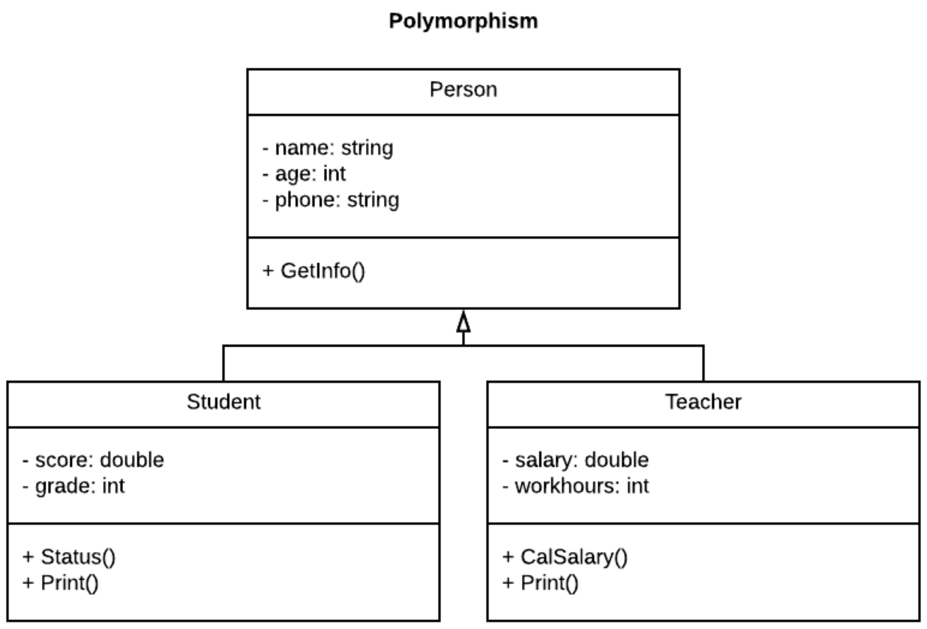


Figure 5. Polymorphism class diagram

Source code of Polymorphism:

```
class Persons
    {
        private string name;
        private int age;

        private string phone;

        public virtual string Name
        {
            get { return name; }
            set { name = value; }
        }

        public virtual int Age
        {
            get { return age; }
            set { age = value; }
        }

        public virtual string Phone
        {
            get { return phone; }
            set { phone = value; }
        }

        public Persons(string name, int age, string phone)
        {
            this.Name = name;
            this.Age = age;
            this.Phone = phone;
        }
    }
```

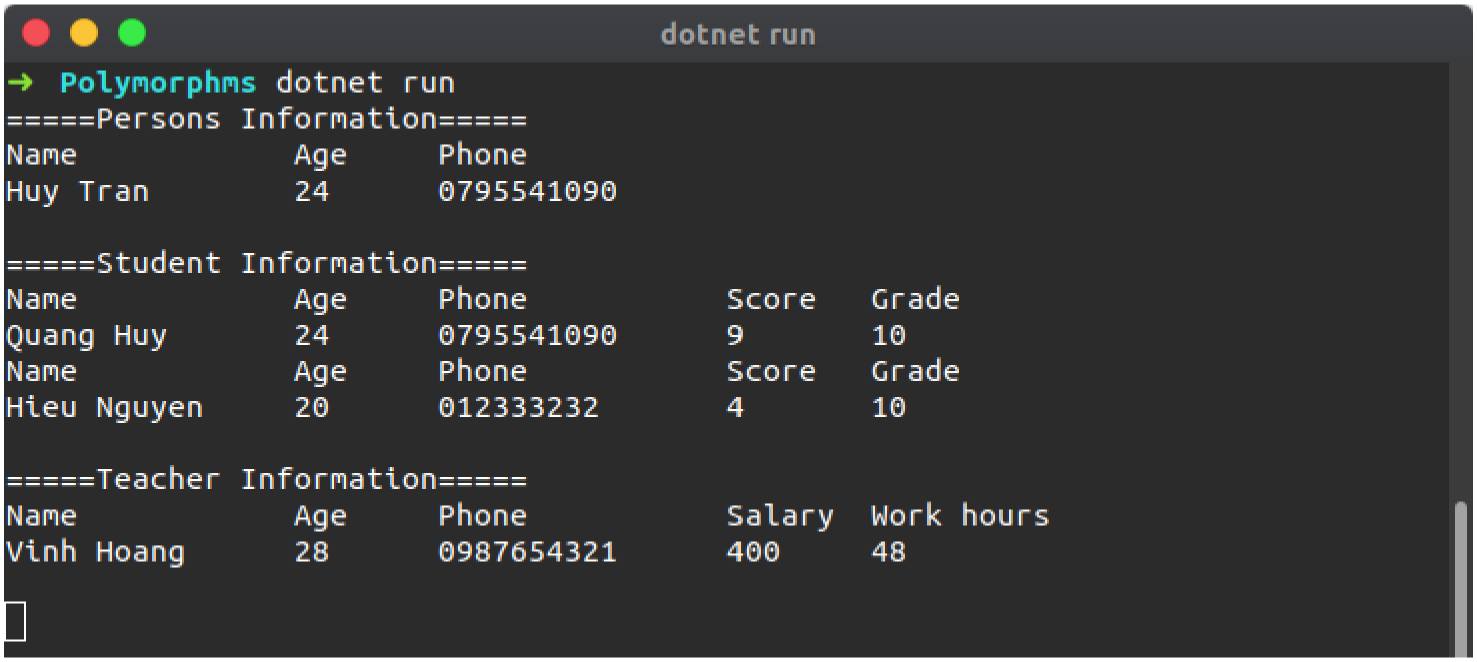
```
class Student : Persons
    {
        private double score;
        private int grade;

        public Student(string name, int age, string phone) : base(name, age, phone)
        {
            score = 0;
            grade = 0;
        }

        public Student(string name, int age, string phone, double score) : base(name, age, 
phone)
        {
            this.score = score;
        }

        public Student(string name, int age, string phone, double score, int grade) : 
base(name, age, phone)
        {
            this.score = score;
            this.grade = grade;
        }

        public void status()
        {
            if (score >= 5)
                Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone 
Number: {Phone}\n\tScore: {score}\n\tMove to next grade({grade} => {grade + 1})\n\n");
            else Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone 
Number: {Phone}\n\tScore: {score}\n\tCant move to next grade({grade} => {grade + 1})\n\n");
        }
    }
```

```
class Teacher : Persons
    {
        private double salary;
        private int workhours;

        public Teacher(string name, int age, string phone) : base(name, age, phone)
        {
            salary = 0;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary) : base(name, age,
phone)
        {
            this.salary = salary;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary, int workhours) :
base(name, age, phone)
        {
            this.salary = salary;
            this.workhours = workhours;
        }

        public int CalSalary(int workhours)
        {
            return 10 * workhours;
        }

        public void status()
        {
            Console.Write($"Infomation of teacher {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tSalary: {salary}\n\tWork hours: {workhours}\n\tSalary get: {CalSalary(workhours)}\n");
        }
    }
```

Code Main function of Polymorphism:

```
class Program
    {
        static void Main(string[] args)
        {
            Student myStudent1 = new Student("Quang Huy", 24, "0795541090", 9, 10);
            Student myStudent2 = new Student("Hieu Nguyen", 20, "012333232", 4, 10);

            Teacher myTeacher1 = new Teacher("Vinh Hoang", 28, "0987654321", 400, 48);

            myStudent1.status();
            myStudent2.status();
            myTeacher1.status();

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

Result:



```
➜ Polymorphms dotnet run
=====Persons Information=====
Name            Age     Phone
Huy Tran        24      0795541090

=====Student Information=====
Name            Age     Phone           Score   Grade
Quang Huy       24      0795541090      9       10
Name            Age     Phone           Score   Grade
Hieu Nguyen     20      012333232       4       10

=====Teacher Information=====
Name            Age     Phone           Salary  Work hours
Vinh Hoang      28      0987654321      400     48
```

Picture 4. Result of Polymorphism code

5. Abstraction

- In c#, Abstraction is a principle of object-oriented programming language (OOP) and it is used to hide the implementation details and display only essential features of the object.
- In Abstraction, by using access modifiers we can hide the required details of object and expose only necessary methods and properties through the reference of object.

Class Diagram of Abstraction:

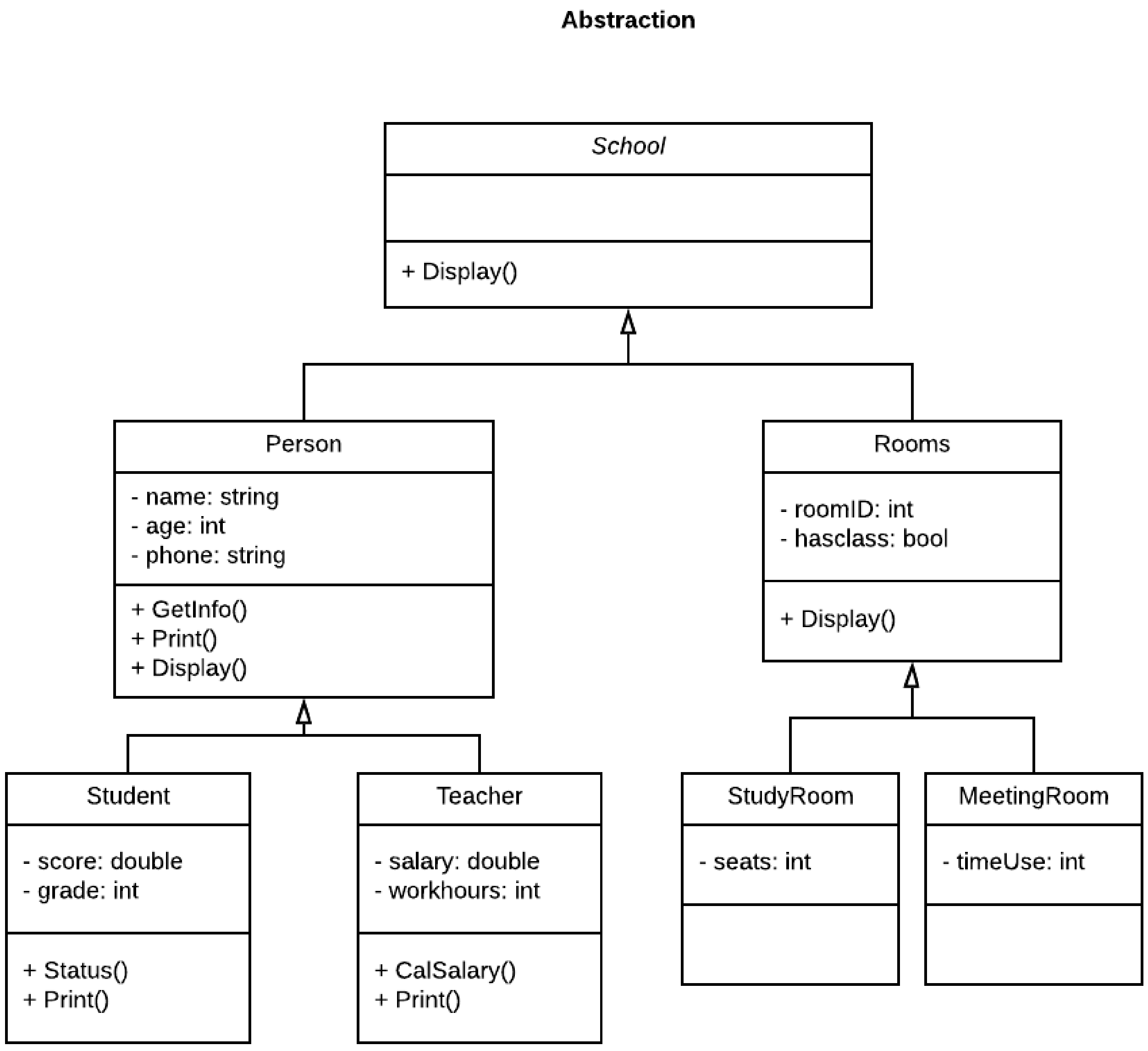


Figure 6. Abstraction class diagram

Source code of Abstraction:

```
abstract class school
    {
        public virtual void Display()
        {

        }
    }
```

```
class Persons : school
    {
        private string name;
        private int age;

        private string phone;

        public virtual string Name
        {
            get { return name; }
            set { name = value; }
        }

        public virtual int Age
        {
            get { return age; }
            set { age = value; }
        }

        public virtual string Phone
        {
            get { return phone; }
            set { phone = value; }
        }
```

```
class Room : school
    {
        private int roomID;
        private bool hasClass;

        public int roomID
        {
            get { return roomID; }
            set { roomID = value; }

        }

        public bool HasClass
        {
            get { return hasClass; }
            set { hasClass = value; }
        }

        public Room(int roomID, bool hasClass)
        {
            this.roomID = roomID;
            this.hasClass = hasClass;
        }

        public override void Display()
        {
            Console.WriteLine("==== Welcome to University of Greenwich ====");
            Console.WriteLine("====        Advance programing         ====");
            Console.WriteLine($"====        Room ID: {roomID}   \t         ====");
            Console.WriteLine($"====        Room status: {hasClass}          ====");
        }
    }
```

```
public Persons(string name, int age, string phone)
        {
            this.Name = name;
            this.Age = age;
            this.Phone = phone;
        }
        public void getInfo()
        {
            Console.WriteLine($"{name} is {age} years old and has phone number:
{phone}");
        }

        public virtual void Print()
        {
            Console.WriteLine("=====Persons Information=====");
            Console.WriteLine("Name\t\tAge\tPhone");
            Console.WriteLine($"{Name}\t{Age}\t{Phone}");
        }

        public override void Display()
        {
            Console.WriteLine("==== Welcome to University of Greenwich ====");
            Console.WriteLine("====        Advance programing         ====");
            Console.WriteLine($"====         Person Name: {Name}      ====");
            Console.WriteLine($"====         Person Age: {Age}              ====");
            Console.WriteLine($"====         Person Phone: {Phone}    ====\n");
        }
    }
```

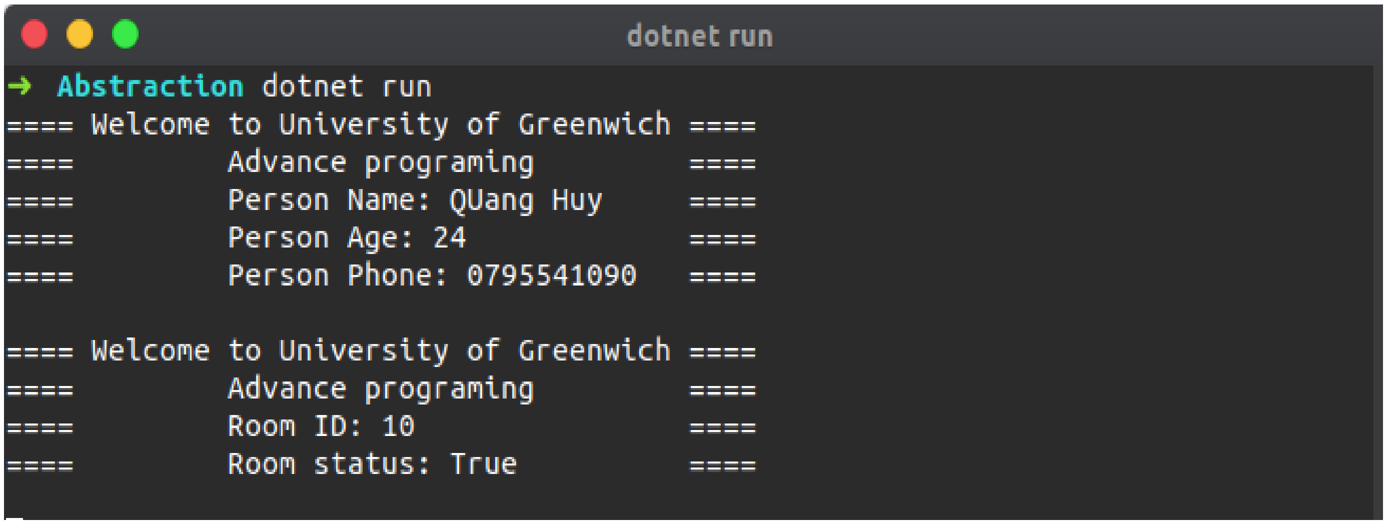
```
class Student : Persons
    {
        private double score;
        private int grade;

        public Student(string name, int age, string phone) : base(name, age, phone)
        {
            score = 0;
            grade = 0;
        }

        public Student(string name, int age, string phone, double score) : base(name, age,
phone)
        {
            this.score = score;
        }

        public Student(string name, int age, string phone, double score, int grade) :
base(name, age, phone)
        {
            this.score = score;
            this.grade = grade;
        }

        public void status()
        {
            if (score >= 5)
                Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tScore: {score}\n\tMove to next grade({grade} => {grade + 1})\n\n");
            else Console.Write($"Infomation of student {Name}:\n\tAge: {Age}\n\tPhone Number:
{Phone}\n\tScore: {score}\n\tCant move to next grade({grade} => {grade + 1})\n\n");
        }
        public override void Print()
        {
            Console.WriteLine("Name\t\tAge\tPhone\t\tScore\tGrade");
            Console.WriteLine($"{Name}\t{Age}\t{Phone}\t{score}\t{grade}");
        }

    }
```

```
class Teacher : Persons
    {
        private double salary;
        private int workhours;

        public Teacher(string name, int age, string phone) : base(name, age, phone)
        {
            salary = 0;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary) : base(name,
age, phone)
        {
            this.salary = salary;
            workhours = 0;
        }

        public Teacher(string name, int age, string phone, double salary, int
workhours) : base(name, age, phone)
        {
            this.salary = salary;
            this.workhours = workhours;
        }

        public int CalSalary(int workhours)
        {
            return 10 * workhours;
        }

        public void status()
        {
            Console.Write($"Infomation of teacher {Name}:\n\tAge: {Age}\n\tPhone
Number: {Phone}\n\tSalary: {salary}\n\tWork hours: {workhours}\n\tSalary get:
{CalSalary(workhours)}\n");
        }

        public override void Print()
        {
            Console.WriteLine("=====Teacher Information=====");
            Console.WriteLine("Name\t\tAge\tPhone\t\tSalary\tWork hours");
            Console.WriteLine($"{Name}\t{Age}\t{Phone}\t{salary}\t{workhours}");
        }
    }
```

Code Main function of Abstraction:

```
class Program
    {
        static void Main(string[] args)
        {
            school mySchool;

            mySchool = new Student("QUang Huy", 24,
"0795541090");
            mySchool.Display();
            mySchool = new Room(10, true);
            mySchool.Display();

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

Result:



```
dotnet run
➜ Abstraction dotnet run
==== Welcome to University of Greenwich ====
====         Advance programing         ====
====         Person Name: QUang Huy     ====
====         Person Age: 24             ====
====         Person Phone: 0795541090   ====

==== Welcome to University of Greenwich ====
====         Advance programing         ====
====         Room ID: 10                ====
====         Room status: True          ====
```

Picture 5. Result of Abstraction code

The different between Encapsulation and Abstraction in C#

Encapsulation	Abstraction
It is used to bind data members and member functions into single unit to prevent outsiders to access it directly.	It is used to hide unwanted data and show only required properties and methods.

Table 1. The different between Encapsulation and Abstraction

PART II. Determine a design pattern from each of the creational, structural and behavioral pattern types and Define class diagrams for specific design patterns using a UML tool.

1. Introduction about design pattern

Based on (Linkedin, 2017), Design pattern is useful as an abstraction tool for solving design problems in every discipline of engineering and architecture. A design pattern is a template for an object or class design that solves a recurring problem. Design patterns are helpful to document the creational, structural, and behavioral characteristics of the software. Separating the user interface from the logic, for example, offers these three distinct advantages:

- Interfaces are better defined and more specific to a device
- Apps become more adaptable to changing requirements
- Objects are reusable

Types of Design Patterns

In Software Engineering there are mainly 3 categories of Design Patterns.

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Design Patterns

The creational patterns aim to separate a system from how its objects are created, composed, and represented.

They increase the system's flexibility in terms of the what, who, how, and when of object creation

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.

There are mainly 5 Creational Patterns as below

- Singleton
- Abstract Factory
- Builder
- Factory Method
- Prototype

Structural Design Patterns

Structural design patterns ease the design by identifying a simple way to realize relationships between entities.

Patterns aim to separate a system from how its objects are created, composed, and represented.

There are mainly 7 Structural Patterns as below

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Design Patterns

Behavioral design patterns identify common communication patterns between objects and realize these patterns.

Behavioral design patterns increase flexibility in carrying out the communication between objects.

There are mainly 11 Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Based on the Design Patterns listed above, I will choose three types of design patterns representing three different types: Builder (Creational Pattern), Decorator (Structural Pattern) and Command (Behavioral Pattern)

2. Builder Pattern (Creational Pattern)

- **Define**

Based on (Wikipedia, n.d.) The Builder is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation. It is one of the Gang of Four design patterns.

- Separate construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.
- Difference between Builder and Abstract Factory
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects - simple or complex.
- Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

- **Brief**

Imagine you want to build a house, you have several options in how your house is made an example: How many floors you want? What types of the room would you like? What swimming pool would you want? In such cases, builder pattern comes to the rescue.

- **Why use it**

In some common cases, we often create constructions class to build individual houses with distinct properties. If there is a way to re-explain the initialization steps, the builder pattern will help us build easier. When there could be several flavors of an object and to avoid the constructor telescoping. The key difference from the factory pattern is that; factory pattern is to be used when the creation is a one step process while builder pattern is to be used when the creation is a multi-step process.

- **Class Diagram**

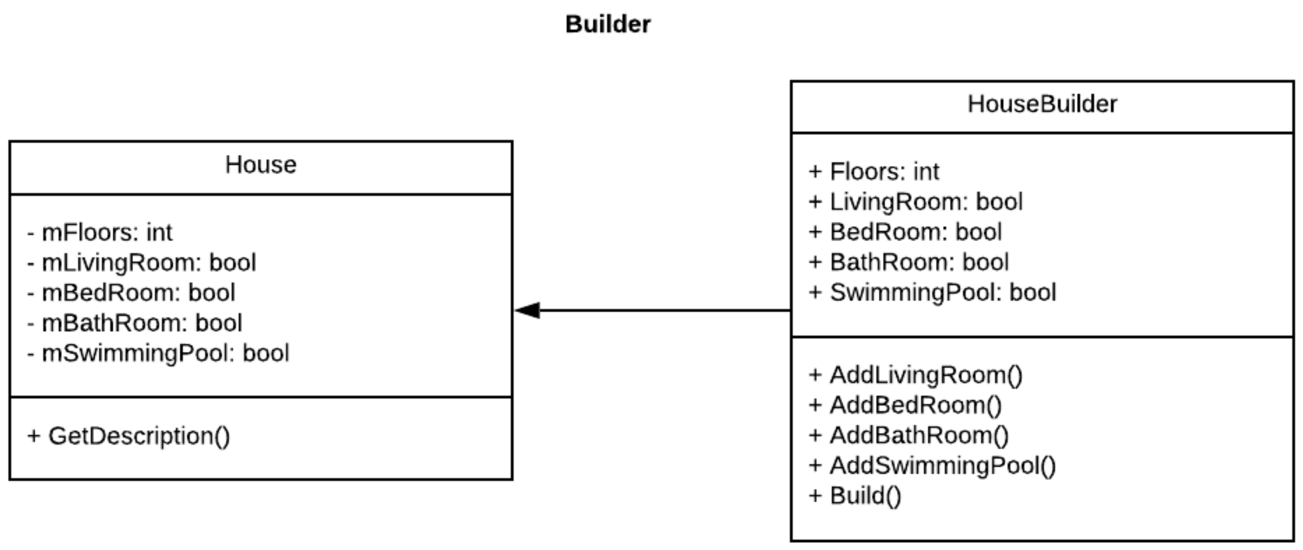


Figure 7. Builder class diagram

- **Source code of Builder design pattern:**

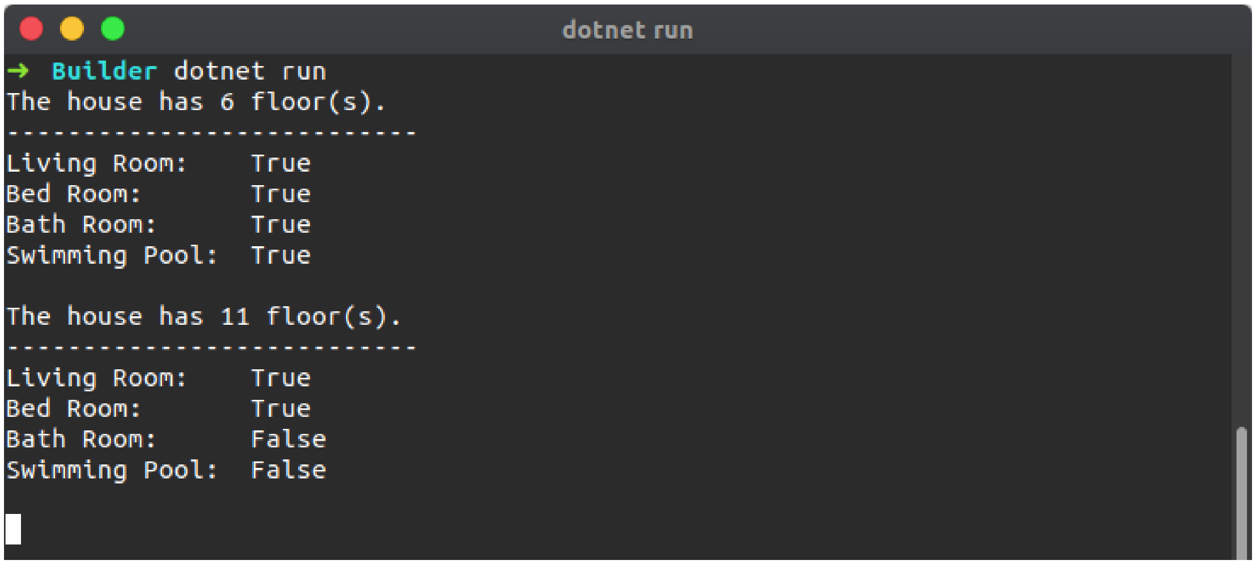
```
class House
  {
      private int mFloors;
      private bool mLivingRoom;
      private bool mBedRoom;
      private bool mBathRoom;
      private bool mSwimmingPool;

      public House(HouseBuilder builder)
      {
          this.mFloors = builder.Floors;
          this.mLivingRoom = builder.LivingRoom;
          this.mBedRoom = builder.BedRoom;
          this.mBathRoom = builder.BathRoom;
          this.mSwimmingPool = builder.SwimmingPool;
      }

      public string GetDescription()
      {
          var sb = new StringBuilder();
          sb.Append($"The house has {this.mFloors} floor(s).\n");
          sb.Append("---------------------------\n");
          sb.Append($"Living Room: \t{this.mLivingRoom}\n");
          sb.Append($"Bed Room: \t{this.mBedRoom}\n");
          sb.Append($"Bath Room: \t{this.mBathRoom}\n");
          sb.Append($"Swimming Pool: \t{this.mSwimmingPool}");
          return sb.ToString();
      }
  }
```

```
class HouseBuilder
    {
        public int Floors;
        public bool LivingRoom;
        public bool BedRoom;
        public bool BathRoom;
        public bool SwimmingPool;
        public HouseBuilder(int floor)
        {
            this.Floors = floor;
        }

        public HouseBuilder AddLivingRoom()
        {
            this.LivingRoom = true;
            return this;
        }

        public HouseBuilder AddBedRoom()
        {
            this.BedRoom = true;
            return this;
        }

        public HouseBuilder AddBathRoom()
        {
            this.BathRoom = true;
            return this;
        }

        public HouseBuilder AddSwimmingPool()
        {
            this.SwimmingPool = true;
            return this;
        }

        public House Build()
        {
            return new House(this);
        }
    }
```

- **Code Main function:**

```
    static void Main(string[] args)
         {
             var myHouse = new HouseBuilder(3).AddBathRoom()
                                              .AddBedRoom()
                                              .AddLivingRoom()
                                              .AddSwimmingPool()
                                              .Build();

              var myHouseOfHieu = new HouseBuilder(3).AddBedRoom()
                                                     .AddLivingRoom()
                                                     .Build();


             Console.WriteLine(myHouse.GetDescription());
              Console.WriteLine(myHouseOfHieu.GetDescription());


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

- **Result:**



dotnet run

```
→ Builder dotnet run
The house has 6 floor(s).
---------------------------
Living Room:    True
Bed Room:       True
Bath Room:      True
Swimming Pool:  True

The house has 11 floor(s).
---------------------------
Living Room:    True
Bed Room:       True
Bath Room:      False
Swimming Pool:  False

```

Picture 6. Result of Builder Pattern

3. Decorator Pattern (Structural Pattern):

- **Define**

Based on (Wikipedia, n.d.), in object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern. The decorator pattern is structurally nearly identical to the chain of responsibility pattern, the difference being that in a chain of responsibility, exactly one of the classes handles the request, while for the decorator, all classes handle the request.

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Client-specified embellishment of a core object by recursively wrapping it.

Wrapping a gift, putting it in a box, and wrapping the box

You want to add behavior or state] to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

- **Brief**

Imagine you run a technology service shop offering multiple services such as sell Macbook. Now how do you calculate the bill to be charged? You pick one configure and dynamically keep adding to it the prices for the provided option till you get the final cost. Here each type of service is a decorator.

- **Why use it**

Decorator is used in this case to change the value of the properties of the Macbook without adding or deleting any properties of the Macbook. Choosing a Macbook and then upgrading options will be easier with Decorator. In other words, subclasses inherit the old properties of a product and then change the value of those properties to create a new product based on the old product whose properties remain the same.

- **Class Diagram:**

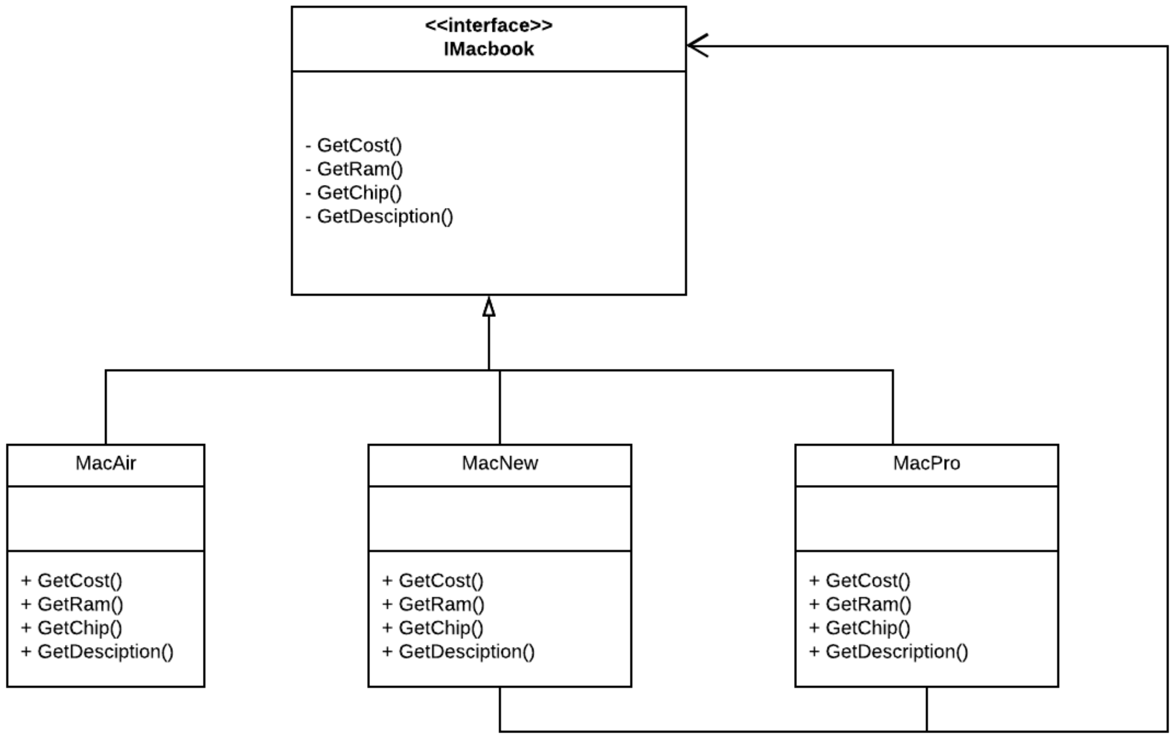


Figure 8. Decorator class diagram

- **Source code of Decorator pattern:**

```
interface IMacbook
    {
        int GetCost();
        int GetRam();
        string GetChip();
        string GetDescription();
    }

    class MacAir : IMacbook
    {
        public int GetCost()
        {
            return 720;
        }

        public int GetRam()
        {
            return 4;
        }

        public string GetChip()
        {
            return "i3";
        }

        public string GetDescription()
        {
            return $"Macbook Air: {GetChip()}\t\tRam: {GetRam()}\t\tCost: {GetCost()}";
        }
    }
```

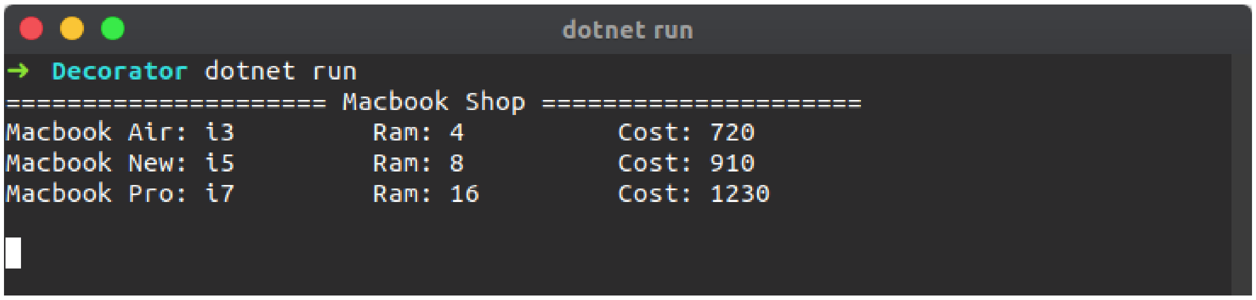
```
class MacNew : IMacbook
    {
        private readonly IMacbook mMacbook;

        public MacNew(IMacbook macbook)
        {
            mMacbook = macbook;
        }

        public int GetCost()
        {
            return mMacbook.GetCost() + 190;
        }

        public int GetRam()
        {
            return mMacbook.GetRam() + 4;
        }

        public string GetChip()
        {
            return "i5";
        }

        public string GetDescription()
        {
            return $"Macbook New: {GetChip()}\t\tRam: {GetRam()}\t\tCost: {GetCost()}";
        }
    }
```

```
class MacPro : IMacbook
{
    private readonly IMacbook mMacbook;

    public MacPro(IMacbook macbook)
    {
        mMacbook = macbook;
    }

    public int GetCost()
    {
        return mMacbook.GetCost() + 320;
    }

    public int GetRam()
    {
        return mMacbook.GetRam() + 8;
    }

    public string GetChip()
    {
        return "i7";
    }

    public string GetDescription()
    {
        return $"Macbook Pro: {GetChip()}\t\tRam: {GetRam()}\t\tCost: {GetCost()}";
    }

}
```

- **Code Main function of Decorator pattern:**

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("===================== Macbook Shop =====================");
            var myMacAir = new MacAir();
            Console.WriteLine(myMacAir.GetDescription());

            var myMacNew = new MacNew(myMacAir);
            Console.WriteLine(myMacNew.GetDescription());

            var myMacPro = new MacPro(myMacNew);
            Console.WriteLine(myMacPro.GetDescription());

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

- **Result:**



```
dotnet run
→ Decorator dotnet run
===================== Macbook Shop =====================
Macbook Air: i3         Ram: 4          Cost: 720
Macbook New: i5         Ram: 8          Cost: 910
Macbook Pro: i7         Ram: 16         Cost: 1230
```

Picture 7. Result of Decorator Pattern

4. Command Pattern (Behavioral Pattern):

- **Define:**

In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Command means command. The commanding officer is called a commander, who does not do it but orders others to do it. As such, there must be a person who receives and executes the order.

Command - provides a class that encapsulates commands

- **Brief:**

A generic example would be you using the garage at your home. You or Client use the remote (**Invoker**) to open or close (**Command**) and remote simply forwards the request to your Garage (**Receiver**) what has the ways of what and how to open or close.

- **Why use it**

Command pattern can also be used to implement a transaction-based system. Where you keep maintaining the history of commands as soon as you execute them. If the final command is successfully executed, all good otherwise just iterate through the history and keep executing the undo on all the executed commands.

- **Class Diagram:**

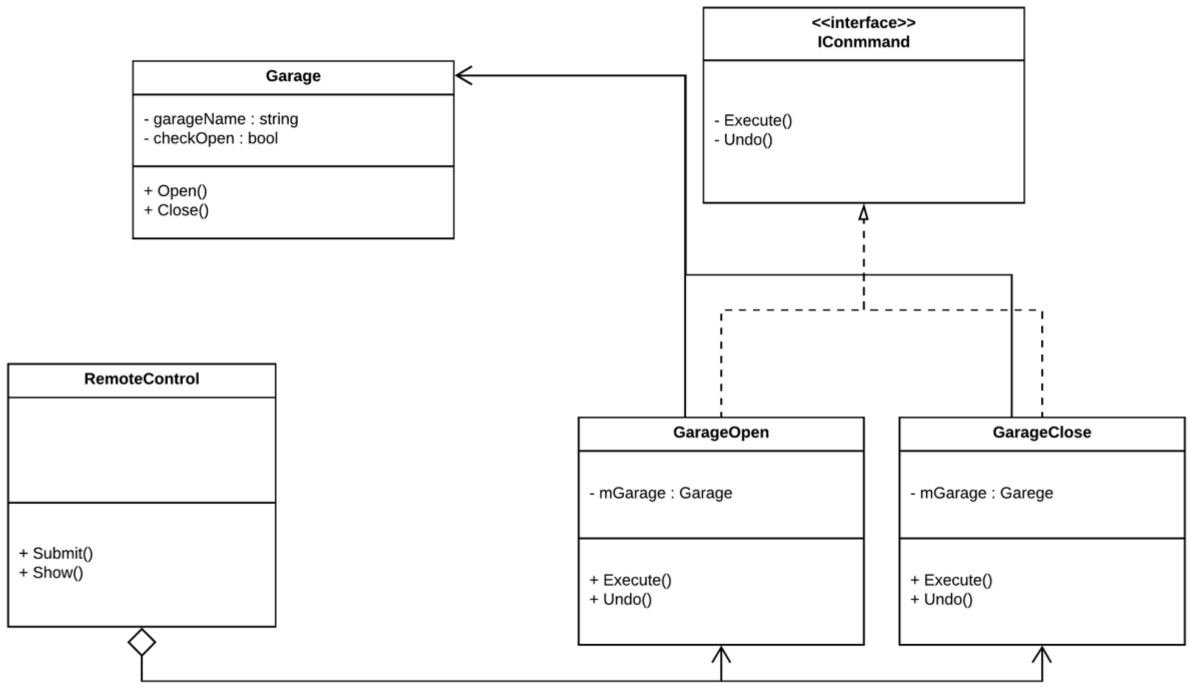


Figure 9. Command class diagram

- **Source code:**

```
class Garage
    {
        private string garageName;
        public Garage(string garageName)
        {
            this.garageName = garageName;
        }

        bool checkedOpen = false;

        public void Open()
        {
            if (checkedOpen == true)
            {
                Console.WriteLine($"{garageName} --> Garage is alredy Opened!");
            }
            else
            {
                Console.WriteLine($"{garageName} --> Garage Opened!");
                checkedOpen = true;
            }

        }

        public void Close()
        {
            if (checkedOpen == true)
            {
                Console.WriteLine($"{garageName} --> Garage Closed!");
                checkedOpen = false;
            }
            else
            {
                Console.WriteLine($"{garageName} --> Garage is alredy Closed!");
            }

        }
    }
```

```
interface ICommand
    {
        void Execute();
        void Undo();
    }

    class GarageOpen : ICommand
    {
        private Garage myGarage;
        public GarageOpen(Garage garage)
        {
            myGarage = garage;
        }
        public void Execute()
        {
            myGarage.Open();
        }
        public void Undo()
        {
            myGarage.Close();
        }
    }
    class GarageClose : ICommand
    {
        private Garage myGarage;

        public GarageClose(Garage garage)
        {
            myGarage = garage;
        }
        public void Execute()
        {
            myGarage.Close();
        }

        public void Undo()
        {
            myGarage.Open();
        }
    }
```

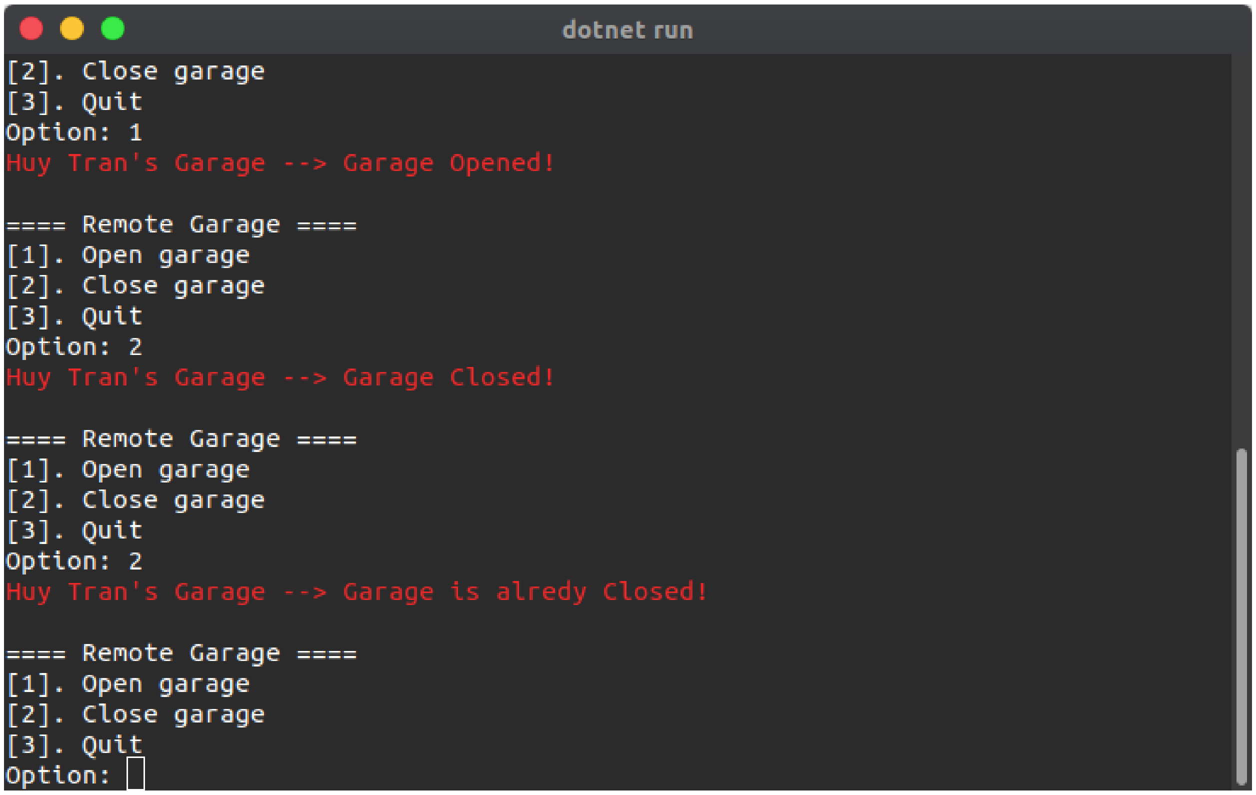
```
class RemoteControl
    {
        public void Submit(ICommand command)
        {
            command.Execute();
        }

        public void Show()
        {
            Console.WriteLine("==== Remote Garage ====");
            Console.WriteLine("[1]. Open garage");
            Console.WriteLine("[2]. Close garage");
            Console.WriteLine("[3]. Quit");
            Console.Write("Option: ");
        }
    }
```

```
class RemoteControl
    {
        public void Submit(ICommand command)
        {
            command.Execute();
        }

        public void Show()
        {
            Console.WriteLine("==== Remote Garage ====");
            Console.WriteLine("[1]. Open garage");
            Console.WriteLine("[2]. Close garage");
            Console.WriteLine("[3]. Quit");
            Console.Write("Option: ");
        }
    }
```

- **Code Main function of Command pattern:**

```
        static void Main(string[] args)
        {
            var myGarage = new Garage("Huy Tran's Garage");

            var openGarage = new GarageOpen(myGarage);
            var CloseGarage = new GarageClose(myGarage);

            var myRemote = new RemoteControl();

            int choose;

            myRemote.Show();
            choose = Int32.Parse(Console.ReadLine());
            while (choose != 3)
            {
                switch (choose)
                {
                    case 1:
                        Console.ForegroundColor = ConsoleColor.Red;
                        myRemote.Submit(openGarage);
                        Console.ResetColor();
                        break;
                    case 2:
                        Console.ForegroundColor = ConsoleColor.Red;
                        myRemote.Submit(CloseGarage);
                        Console.ResetColor();
                        break;
                    case 3:
                        break;
                }
                if (choose == 3) break;
                Console.WriteLine();
                myRemote.Show();
                choose = Int32.Parse(Console.ReadLine());
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
```

- **Result:**



```
[2]. Close garage
[3]. Quit
Option: 1
Huy Tran's Garage --> Garage Opened!

==== Remote Garage ====
[1]. Open garage
[2]. Close garage
[3]. Quit
Option: 2
Huy Tran's Garage --> Garage Closed!

==== Remote Garage ====
[1]. Open garage
[2]. Close garage
[3]. Quit
Option: 2
Huy Tran's Garage --> Garage is alredy Closed!

==== Remote Garage ====
[1]. Open garage
[2]. Close garage
[3]. Quit
Option: 
```

Picture 8. Result of Command Pattern

PART III. Analyze the relationship between the Object-Oriented paradigm and design patterns

1. Why was the design pattern born because OOP's problems?

Base on: Design Pattern: Elements of Reusable Object-Oriented:

- Software that object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.
- Yet **experienced object-oriented designers** do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what **good object-oriented design** is all about.
- One thing expert designer know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more **flexible**, **elegant**, and **ultimately reusable**. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to **design problems without having to rediscover them**.
- **Design patterns** make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

2. How design patterns solve OOP's problems?

The concept of Design Pattern is closely associated with Object Oriented Programing (OOP). Design patterns solve many of the problems object-oriented. There are several of these problems and how design patterns solve them:

Finding Appropriate Objects:

- **OOP Problem:**
  Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client.

  The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, re-usability, and on and on. They all influence the decomposition, often in conflicting ways.

  Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays, others are much higher-level.

- **Using design pattern:**
  Design patterns help you identify less-obvious abstractions and the objects that can capture them.

  For example: Objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs.

◦ **The Strategy pattern** describes how to implement interchangeable families of algorithms.
◦ **The State pattern** represents each state of an entity as an object.
These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Determining Object Granularity:

- **OOP Problem:**
  Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

- Using design pattern:
  - **The Facade pattern:** describes how to represent complete subsystems as objects
  - **The Flyweight pattern:** describes how to support huge numbers of objects at the finest granularities.

  Other design patterns describe specific ways of decomposing an object into smaller objects:
  - **Abstract Factory and Builder:** yield objects whose only responsibilities are creating other objects.
  - **Visitor and Command:** yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces:

- **OOP Problem:**
  Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

- **Using design pattern:**
  - Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface.

  - **The Memento pattern** describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. The pattern stipulates that Memento objects must define two interfaces: a restricted one that lets clients hold and copy mementos, and a privileged one that only the original object can use to store and retrieve state in the memento.

  - **Decorator and Proxy pattern** require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects.

  - **The Visitor pattern** interface must reflect all classes of objects that visitors can visit.

Class versus Interface Inheritance

- **OOP Problem:**
  It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another. Although most programming languages don't support the distinction between interface and implementation inheritance, people make the distinction in practice.

  For example: objects in a "Chain of Responsibility" must have a common type, but usually they don't share a common implementation.

- **Using design pattern:**
  - **The Composite pattern:** Component defines a common interface, but Composite often defines a common implementation.
  - **The Command, Observer, State, and Strategy patterns:** Often implemented with abstract classes that are pure interfaces.

Programming to an Interface, not an Implementation

- **OOP Problem:**
  When inheritance is used carefully, all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. All sub classes can then respond to the requests in the interface of this abstract class, making them all sub types of the abstract class.

  Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class. You have to instantiate concrete classes that specify a particular implementation somewhere in your system.

- **Using design pattern:**
  - The creational patterns such as **Abstract Factory**, **Builder**, **Factory Method**, **Prototype** and **Singleton**: By abstracting the process of object creation, these patterns give you different ways to associate an interface with its implementation transparently at instantiation. Creational patterns ensure that your system is written in terms of interfaces, not implementations.

Here are some common causes of redesign along with the design pattern:

- **Creating an object by specifying a class explicitly.**
  Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.

  Design patterns: Abstract Factory, Factory Method, Prototype.

- **Dependence on specific operations.**
  When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.

  Design patterns: Chain of Responsibility, Command.

- **Dependence on hardware and software platform.**
  External operating system interfaces and application programming interfaces are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms. It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies.

  Design patterns: Abstract Factory, Bridge.

- **Dependence on object representations or implementations.**
  Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.

  Design patterns: Abstract Factory, Bridge, Memento, Proxy.

- **Algorithmic dependencies.**
  Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore, algorithms that are likely to change should be isolated.

  Design patterns: Builder, Iterator, Strategy, Template Method, Visitor.

- **Tight coupling.**
  Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain.

  Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.

- **Inability to alter classes conveniently.**
  Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it as may be the case with a commercial class library or maybe any change would require modifying lots of existing subclasses. Design patterns offer ways to modify classes in such circumstances.

  Design patterns: Adapter, Decorator, Visitor.

3. Summary of relationship between OOP and Design pattern

Firstly, we need to know what is Object-Oriented programming: is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types. (wikipedia, n.d.)

Secondly OOP is a programming paradigm and we should know what programming paradigm is and what Design Pattern is:

Design pattern: In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. (wikipedia, n.d.)

Programming paradigm: Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Thirdly, what is Difference between Programming Paradigm and Design Pattern:

(qoura, n.d.)

Programming Paradigm	Design Pattern
Programming paradigm, is a method, a way, a principle of programming. It describes the programming process, which is the way programs are made. It explains core structure of program written in certain paradigm, everything that program consists of and its components. I know of following programming paradigms: Procedural paradigm, functional paradigm, object-oriented paradigm, logical paradigm, language-oriented programming, event-driven programming, modular programming, and structured paradigm.	In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. (GoF definition) Design patterns are formalized solutions to common programming problems. They mostly refer to object-oriented programming, but some of solutions can be applied in various paradigms.
Now a design pattern is a tried and tested solution to a common programming pattern. It could be considered a best practice. If you approach a program using an Object-Oriented paradigm, there are a number of design patterns you can then draw on to solve specific problems. (allthingsjavascript, n.d.)	

Table 2. The difference between Programming Paradigm and Design Pattern

Why are design patterns and OOP treated separately?

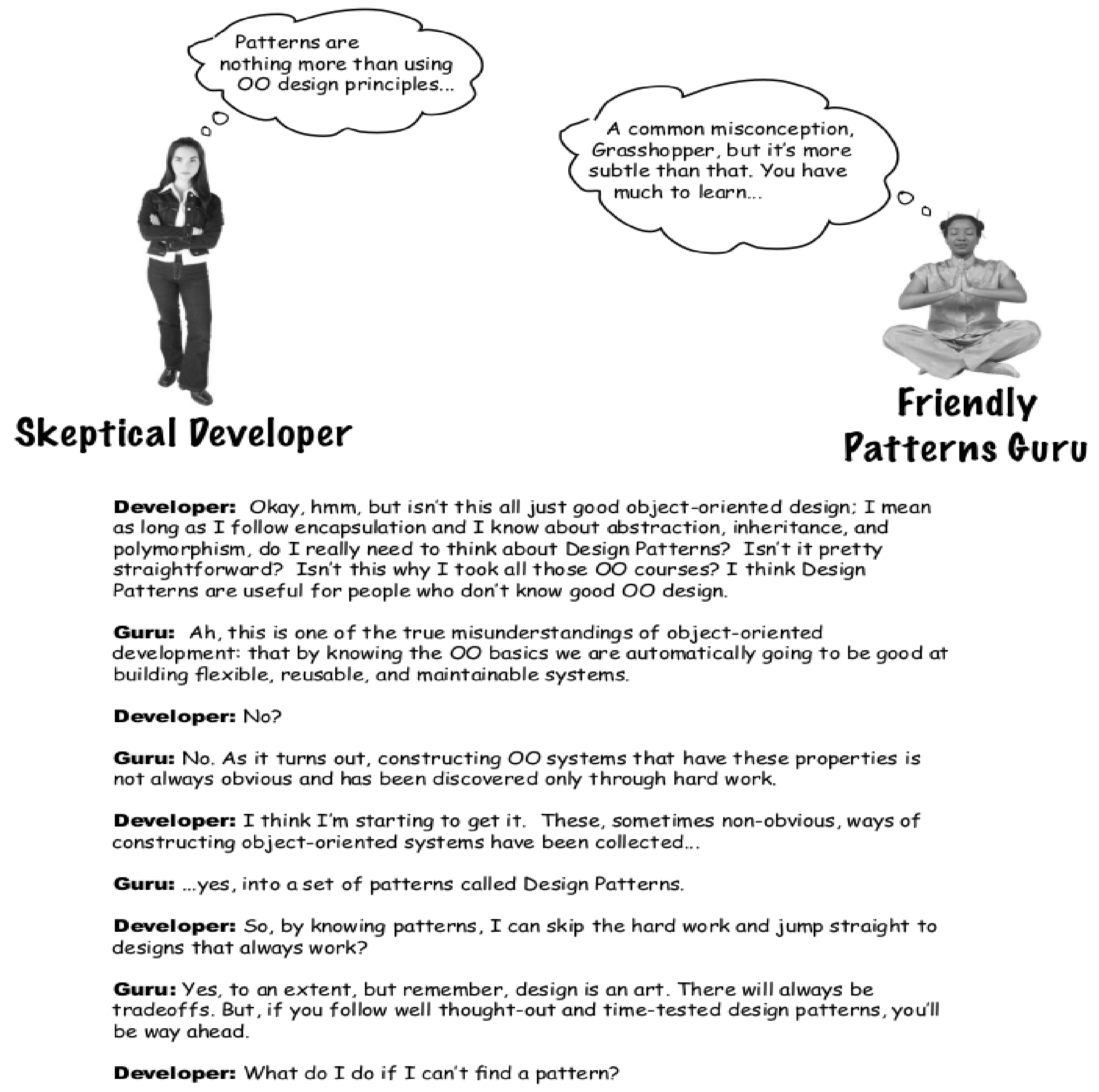
Because they are different subjects. In general, you learn to program, then you learn how to think about programming.

Base on stack overflow (stackoverflow, n.d.) Talk about how is OOP and Design Patterns related? I found that Design patterns and OOP are not the same. A design pattern is a proven, battle-tested way of solving some problem.

Base on Book: Design Patterns: Elements of Reusable Object-Oriented Software.

Design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

Base on Story in Book: Head First Design Patterns.


Patterns are nothing more than using OO design principles...
A common misconception, Grasshopper, but it's more subtle than that. You have much to learn...
Skeptical Developer
Friendly Patterns Guru

Developer: Okay, hmm, but isn't this all just good object-oriented design; I mean as long as I follow encapsulation and I know about abstraction, inheritance, and polymorphism, do I really need to think about Design Patterns? Isn't it pretty straightforward? Isn't this why I took all those OO courses? I think Design Patterns are useful for people who don't know good OO design.

Guru: Ah, this is one of the true misunderstandings of object-oriented development: that by knowing the OO basics we are automatically going to be good at building flexible, reusable, and maintainable systems.

Developer: No?

Guru: No. As it turns out, constructing OO systems that have these properties is not always obvious and has been discovered only through hard work.

Developer: I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected...

Guru: ...yes, into a set of patterns called Design Patterns.

Developer: So, by knowing patterns, I can skip the hard work and jump straight to designs that always work?

Guru: Yes, to an extent, but remember, design is an art. There will always be tradeoffs. But, if you follow well thought-out and time-tested design patterns, you'll be way ahead.

Developer: What do I do if I can't find a pattern?

Picture 9. Story in Head First Design Pattern book

A Developer can programming in OOP without using Design Patterns, but if Developer want to be good at building flexible, reusable or maintainable system. They have to find out what is **Design Patterns**.

Base on Jeremy Gibons, Computer Scientist and Professor of Computing at the University of Oxford proposed a design pattern called ‘ORIGAMI’:The pattern deals with higher level programming, one of which is ‘origami’ programming paradigm, in which the “structure of each program reflects that of the datatype it traverses.” This means Design Patterns might not be completely free from object orientation, especially in higher level programming.

In fact, 16 out of 23 of the Gang of Four Design Patterns would “either be simpler or invisible” in Functional Programming paradigm (programming language in Lisp or Dylan) – found by Peter Norvig. Which means **Design Patterns do not need to be Object-Oriented**.

Ralf L¨ammelg and Joost Visser had also formally documented 6 basic and 7 advanced design patterns for Functional Strategic Programming. This is not to mention the existence of ‘monad’ - a design pattern from Haskell programming language.

So, in my opinion **OOP is not a design pattern**. OOP is basic. Design Patterns (GoF23) are approaches to implement OOP concept. Design Pattern is a technique in object-oriented programming, it is quite important and every good programmer must know. Used frequently in OOP languages. It will give you "designs", solutions to solve common problems, common in programming. The problems you encounter may come up with your own solutions, but they may not be optimal. Design Pattern helps you solve problems in the most optimal way, providing you with solutions in OOP programming.

Design Patterns are not specific languages at all. It can be implemented in most programming languages, such as Java, C #, Javascript or any other programming language.

PART IV. Define/ refine class diagrams derived from a given code scenario using a UML tool.

Making easy to understand Define/refine class diagrams derived from a give code scenario using a UML tool from previous class diagrams and code in OOP, I will make a new scenario and using UML tool to draw class diagrams step by step with code.

1. A scenario:

The Highlands Coffee has made for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

Because they've grown so quickly, they're scrambling to **update** their **ordering systems** to match their **beverage offerings**.

The manager of Highlands coffee has defined several requirements that need to be fulfilled the order system:

- The coffee shop serves some kinds of coffee (**House Blend, Dark Roast**, **Decaf**, **Espresso**).
- The coffee can be combined with additional Condiments like steamed milk, soy, mocha (Chocolate) and have it all topped off with whipped milk.
- Highlands charges a bit for each of these, so they really need to get them built into their order system.

The Coffee shop has a menu card include coffee and condiments like table below:

Highlands Coffee	
Coffees	
House Blend	1.2
Dark Roast	1.4
Decaf	1.7
Espresso	1.9
Condiments	
Steamed milk	.20
Mocha	.30
Soy	.25
Whip	.20

Table 3. Highlands Coffee menu

2. Class diagram

Here's the first attempt for class diagram: Each cost method computes the cost of the coffee along with the other condiments in the order.

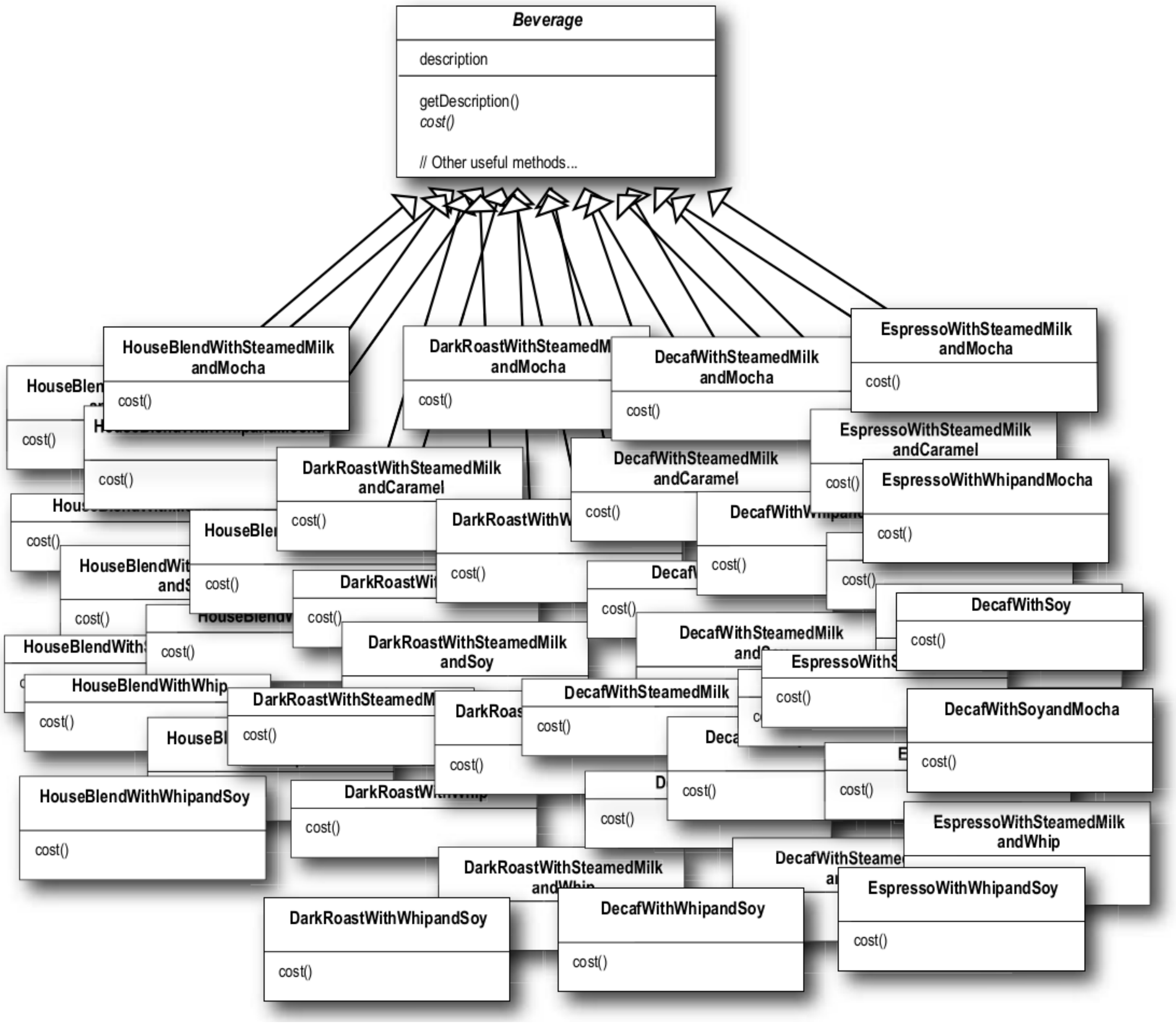

Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
HouseBlendWithWhip
DecafWithSoyandMocha
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()

Figure 10. The begin class diagram of Highlands

It's pretty obvious that **Highlands** has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

So, after that, let’s start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha and whip…

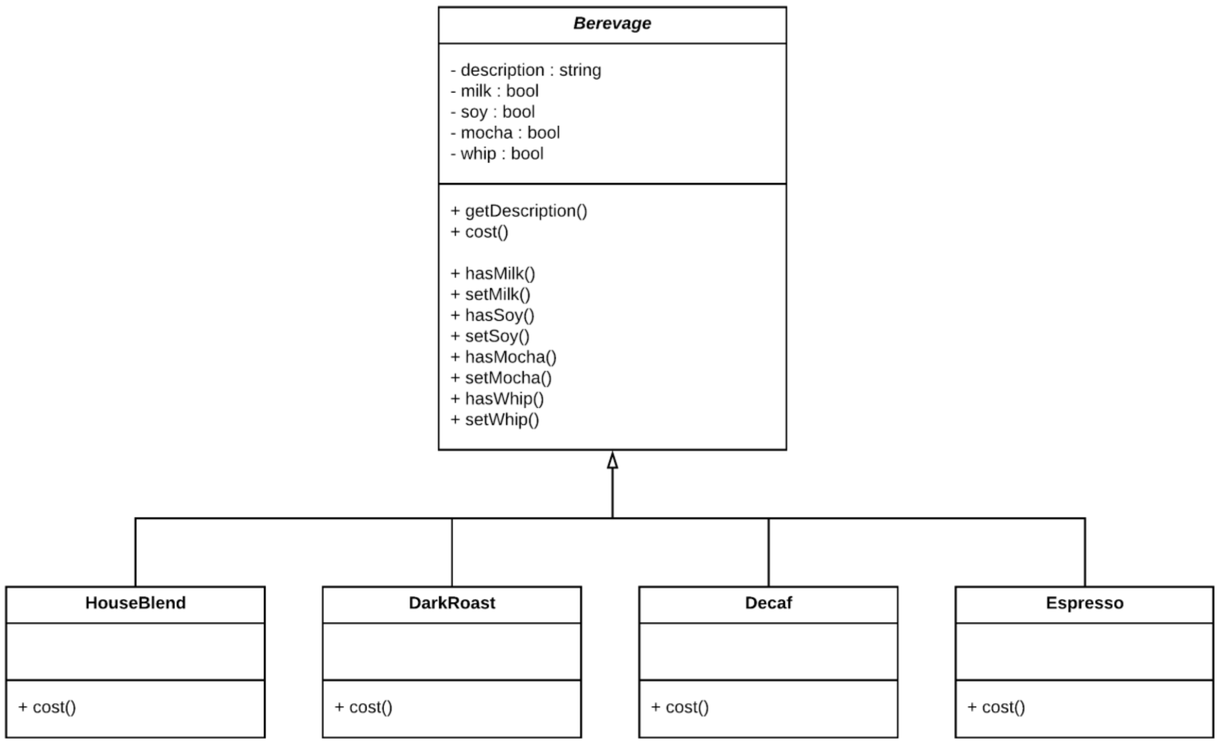


Figure 11. Optimize OOP in Highlands's Class diagram

In the subclasses, one for each beverage on the menu, The superclass **Cost()** will calculate the costs for all of the condiments, while the overridden cost() in the subclass will extend that functionality to include costs for that specific beverage type.

Each **cost()** method need to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

What requirements or other factors might change that will impact this design?

- Price changes for condiments will force us to alter existing code.
- New condiments will force us to add new methods and alter the cost method in the superclass.
- We may have new beverages. For some of these beverages like **iced tea**, the condiments may not be appropriate, the **Tea** subclass will still inherit method like **hasMilk()**.
- What if a customer wants a double **mocha**?

As we saw in previous class diagrams so that is very bad idea! So, that is enough for the **Object-Oriented Design**. Now we will start with a **beverage** and **decorate** it with the condiments at runtime.

For example: Customer wants a Dark Roast with Mocha and Whip

- Take a **DarkRoast** object
- Decorate it with a **Mocha** object
- Decorate it with a **Whip** object
- Call the **cost()** method and rely on delegation to add on the condiment costs

Constructing a drink order with Decorators

- We start with our DarkRoast object.

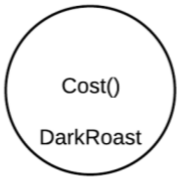


Figure 12. Dark Roast object

- The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

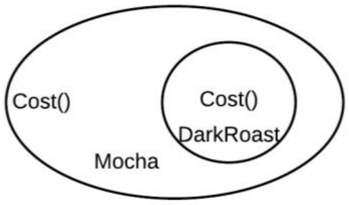


Figure 13. Mocha object

- The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

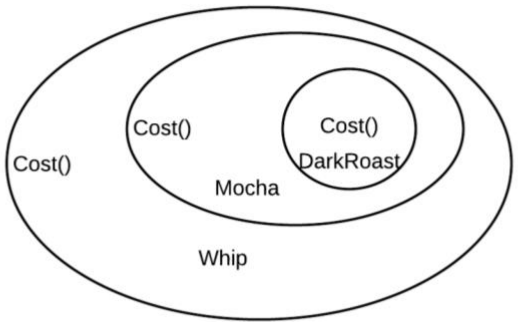


Figure 14. Whip object

Now it’s time to compute the cost for the customer. We do this by calling **cost()** on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.

Decorating our Beverages:

The Decorator Pattern attaches additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to sub classing for extending functionality.

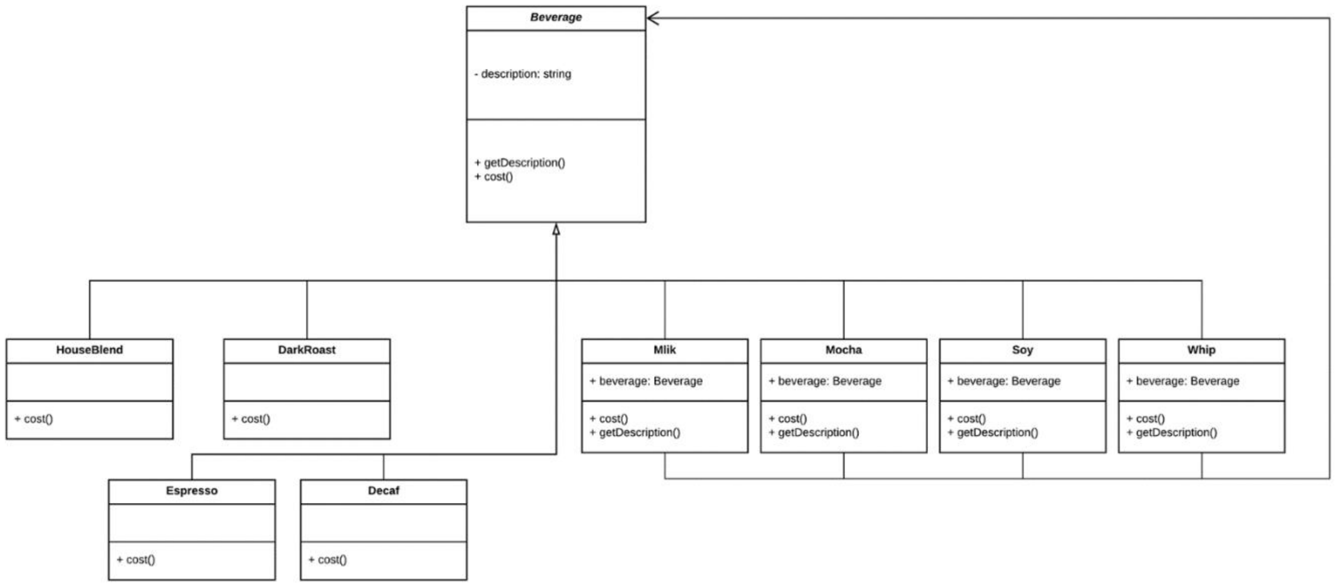


Figure 15. Using decorator to design Highlands class diagram

Explanation:

Beverage acts as our abstract component:

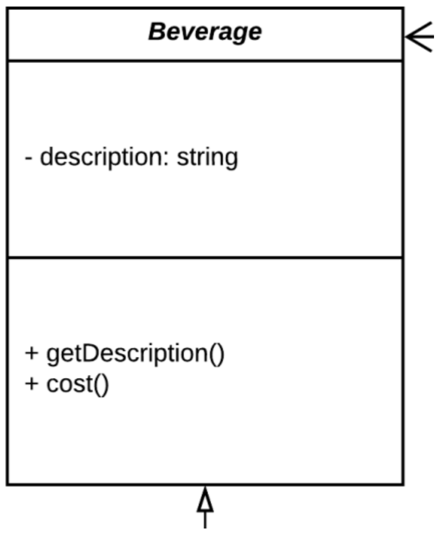


Figure 16. Abstract class Beverage

The four concrete components, one per coffee type.

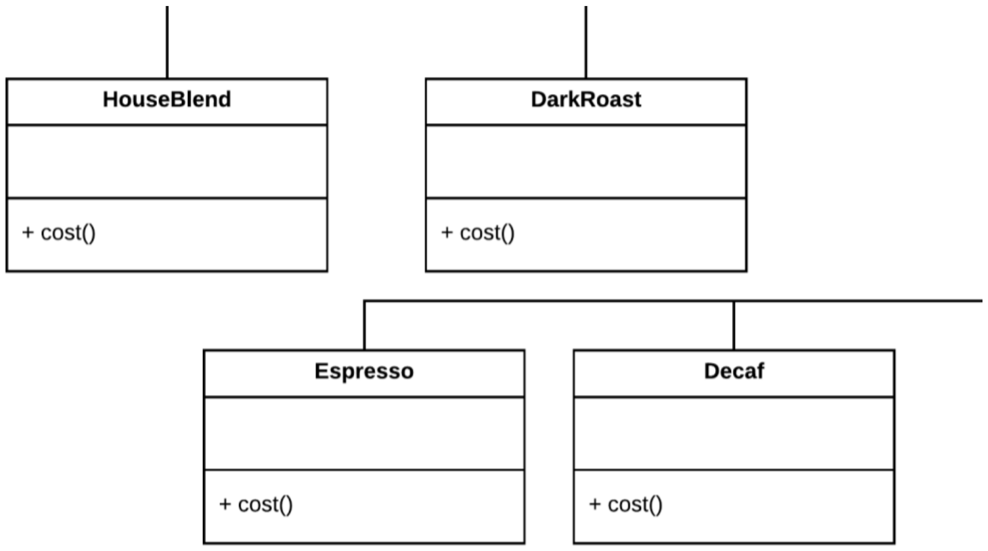


Figure 17. Four concrete components

Here are our condiment decorators, notice they need to implement not only **cost()** but also **getDescription().**

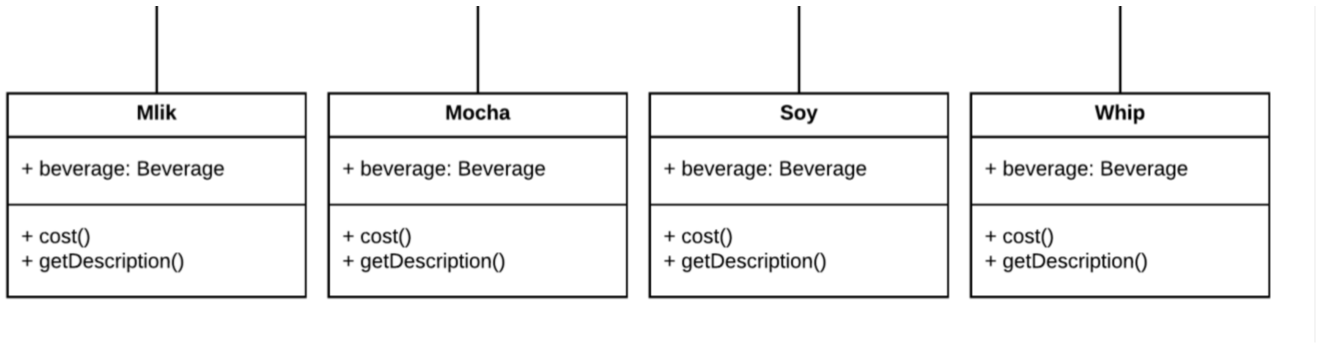


Figure 18. Decorator class diagram

- When we compose a **decorator** with a component, we are adding **new behavior.** We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together. So, we're subclassing the abstract class Beverage in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as other decorators.

- If we rely on inheritance, then our behavior can only be determined statically at compile time. In other words, we get only whatever behavior the superclass gives us or that we override. With composition, we can mix and match decorators any way we like at **runtime**.

3. Writing the Highlands Coffee code:

- The **Beverage** class:

Beverage is an abstract class with the two methods **getDescription()** and **cost()**

```
public abstract class Beverage
    {
        protected string description = "Unknow Beverage";

        public string getDescription()
        {
            return description;
        }

        public abstract double cost();
    }
```

```
public abstract class Beverage
    {
        protected string description = "Unknow Beverage";

        public string getDescription()
        {
            return description;
        }

        public abstract double cost();
    }
```

- Now that we've got our base classes out of the way, let's implement some beverages. We'll start with **HouseBlend**, **DarkRoast**, **Decaf** and **Expresso**. Remember, we need to set a description for the specific beverage and also implement the **cost()** method.

Firstly, we extend the Beverage class, since this is a beverage.

To take care of the description, we set this in the constructor for the class.

We need to compute the cost of each **HouseBlend**, **DarkRoast**, **Decaf** and **Expresso.** We don't need to worry about adding in condiments in this class, we just need to return the price of each beverage.

```
public class HouseBlend : Beverage
    {
        public HouseBlend()
        {
            description = "House Blend";
        }

        public override double cost()
        {
            return 1.2;
        }
    }

    public class DarkRoast : Beverage
    {
        public DarkRoast()
        {
            description = "Dark Roast";
        }

        public override double cost()
        {
            return 1.4;
        }
    }
```

```
public class HouseBlend : Beverage
    {
        public HouseBlend()
        {
            description = "House Blend";
        }

        public override double cost()
        {
            return 1.2;
        }
    }

    public class DarkRoast : Beverage
    {
        public DarkRoast()
        {
            description = "Dark Roast";
        }

        public override double cost()
        {
            return 1.4;
        }
    }
```

```
public class Decaf : Beverage
    {
        public Decaf()
        {
            description = "Decaf";
        }

        public override double cost()
        {
            return 1.7;
        }
    }


    public class Espresso : Beverage
    {
        public Espresso()
        {
            description = "Espresso";
        }

        public override double cost()
        {
            return 1.9;
        }
    }
```

```
public class Decaf : Beverage
    {
        public Decaf()
        {
            description = "Decaf";
        }

        public override double cost()
        {
            return 1.7;
        }
    }


    public class Espresso : Beverage
    {
        public Espresso()
        {
            description = "Espresso";
        }

        public override double cost()
        {
            return 1.9;
        }
    }
```

- Looking back at the Decorator Pattern class diagram, we'll see we've now written our abstract component (**Beverage**), we have our concrete components (**HouseBlend**), and we have our abstract decorator (**Beverage**). Now it's time to implement the concrete decorators.

```
public class SteamedMilk : Beverage
    {
        Beverage myBeverage;

        public SteamedMilk(Beverage beverage)
        {
            myBeverage = beverage;
        }

        public override string getDescription()
        {
            return $"{myBeverage.getDescription()}, Steamed Milk";
        }

        public override double cost()
        {
            return myBeverage.cost() + 0.20;
        }
    }
```

```
public class Mocha : Beverage
    {
        Beverage myBeverage;

        public Mocha(Beverage beverage)
        {
            myBeverage = beverage;
        }

        public override string getDescription()
        {
            return myBeverage.getDescription() + ", Mocha";
        }

        public override double cost()
        {
            return myBeverage.cost() + 0.30;
        }
    }



public class Mocha : CondimentDecorator
    {
        Beverage myBeverage;

        public Mocha(Beverage beverage)
        {
            myBeverage = beverage;
        }

        public override string getDescription()
        {
            return myBeverage.getDescription() + ", Mocha";
        }

        public override double cost()
        {
            return myBeverage.cost() + 0.30;
        }
    }
```

```
public class Soy : Beverage
    {
        Beverage beverage;

        public Soy(Beverage beverage)
        {
            this.beverage = beverage;
        }

        public override string getDescription()
        {
            return beverage.getDescription() + ", Soy";
        }

        public override double cost()
        {
            return beverage.cost() + 0.25;
        }
    }



public class Soy : CondimentDecorator
    {
        Beverage beverage;

        public Soy(Beverage beverage)
        {
            this.beverage = beverage;
        }

        public override string getDescription()
        {
            return beverage.getDescription() + ", Soy";
        }

        public override double cost()
        {
            return beverage.cost() + 0.25;
        }
    }
```

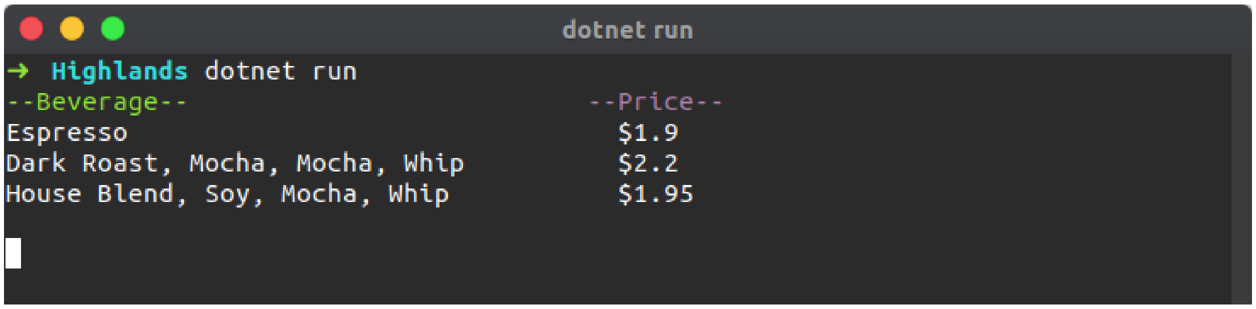
```
public class Whip : Beverage
    {
        Beverage myBeverage;

        public Whip(Beverage beverage)
        {
            myBeverage = beverage;
        }

        public override string getDescription()
        {
            return myBeverage.getDescription() + ", Whip";
        }

        public override double cost()
        {
            return myBeverage.cost() + 0.20;
        }
    }



public class Whip : CondimentDecorator
    {
        Beverage myBeverage;

        public Whip(Beverage beverage)
        {
            myBeverage = beverage;
        }

        public override string getDescription()
        {
            return myBeverage.getDescription() + ", Whip";
        }

        public override double cost()
        {
            return myBeverage.cost() + 0.20;
        }
    }
```

- Finally, we need to run a code, here is some test code to make order:

```
static void Main(string[] args)
        {
            Beverage myBeverage = new Espresso();
            Console.WriteLine($"{myBeverage.getDescription()}: ${myBeverage.cost()}");

            Beverage myBeverage2 = new DarkRoast();
            myBeverage2 = new Mocha(myBeverage2);
            myBeverage2 = new Mocha(myBeverage2);
            myBeverage2 = new Whip(myBeverage2);
            Console.WriteLine($"{myBeverage2.getDescription()}:
${myBeverage2.cost()}");

            Beverage myBeverage3 = new HouseBlend();
            myBeverage3 = new Soy(myBeverage3);
            myBeverage3 = new Mocha(myBeverage3);
            myBeverage3 = new Whip(myBeverage3);
            Console.WriteLine($"{myBeverage3.getDescription()}:
${myBeverage3.cost()}");

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

```
static void Main(string[] args)
        {
            Beverage myBeverage = new Espresso();
            Console.WriteLine($"{myBeverage.getDescription()}: ${myBeverage.cost()}");

            Beverage myBeverage2 = new DarkRoast();
            myBeverage2 = new Mocha(myBeverage2);
            myBeverage2 = new Mocha(myBeverage2);
            myBeverage2 = new Whip(myBeverage2);
            Console.WriteLine($"{myBeverage2.getDescription()}:
${myBeverage2.cost()}");

            Beverage myBeverage3 = new HouseBlend();
            myBeverage3 = new Soy(myBeverage3);
            myBeverage3 = new Mocha(myBeverage3);
            myBeverage3 = new Whip(myBeverage3);
            Console.WriteLine($"{myBeverage3.getDescription()}:
${myBeverage3.cost()}");

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

Result:



dotnet run

```
➜ Highlands dotnet run
--Beverage--                           --Price--
Espresso                                 $1.9
Dark Roast, Mocha, Mocha, Whip           $2.2
House Blend, Soy, Mocha, Whip            $1.95
```

Picture 10. Result of using Decorator for ordering beverage

CONCLUSION

- ✓ After completion content of the report, we are achieved about: 4 characteristic of OOP include: Define, Brief, Class Diagram for each characteristics and Result of each characteristics.
- ✓ Understood about Design Pattern include: Creational, Structural and Behavioral.
- ✓ Understood relationship between OOP and Design Pattern.

Bibliography

Linkedin, 2017. *linkedin.* [Online]
Available at: https://www.linkedin.com/pulse/design-patterns-summary-pradeep-misra/

qoura, n.d. [Online]
Available at: https://www.quora.com/What-is-the-difference-between-a-programming-paradigm-and-a-design-pattern

stackoverflow, n.d. [Online]
Available at: https://stackoverflow.com/questions/478773/how-is-oop-and-design-patterns-related?

wikipedia, n.d. [Online]
Available at: https://en.wikipedia.org/wiki/Object-oriented_programming

wikipedia, n.d. [Online]
Available at: https://en.wikipedia.org/wiki/Software_design_pattern

Wikipedia, n.d. [Online]
Available at: https://en.wikipedia.org/wiki/Decorator_pattern

Wikipedia, n.d. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Builder_pattern