

ASSIGNMENT 2 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 20: Advanced Programming		
Submission date		Date Received 1st submission	
Re-submission Date		Date Received 2nd submission	
Student Name	Ngô Hoàng Thành	Student ID	GCD18591
Class	GCD0705	Assessor name	
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.			
		Student's signature	

Grading grid

P3	P4	M3	M4	D3	D4

⚙ **Summative Feedback:**

⚙ **Resubmission Feedback:**

Grade:

Assessor Signature:

Date:

Lecturer Signature:

Table of Content

I. Build an application derived from UML class diagrams.(P3).....	4
1. Scenario and UML class diagram.	4
2. Develop a C# Application based on UML Class Diagram	6
3. UML after completing the program	19
II. Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types.(P4)	20
1. Creational pattern types	20
2. Behavioral pattern types	22
3. Structural pattern types	24
III. Reference	26

Table of Figure

Figure 1: KIINBURGER menu	5
Figure 2: UML class diagram design.....	6
Figure 3: IDrink interface	7
Figure 4: enum DrinkSize.....	7
Figure 5: Class Coca	8
Figure 6: Class Pepsi and TraditionalMilkTea	9
Figure 7: Class JellyMilkTea and PearlMilkTea	10
Figure 8: IHamburger interface	11
Figure 9: enum HamburgerSize	11
Figure 10: Extra food	12
Figure 11: SimpleHamburger class.....	13
Figure 12: ShrimpHamburger class	14
Figure 13: FlavaHamburger and BeefHamburger class.....	16
Figure 14: Class Order	17

Figure 15: Main function	18
Figure 16: Output.....	19
Figure 17: Complete UML class diagram.....	20

Table of Table

Table 1: Creational pattern.....	22
Table 2: Behavioral pattern.....	24
Table 3: Structural pattern.....	26

I. Build an application derived from UML class diagrams.(P3)

1. Scenario and UML class diagram.

KINNBURGER is a newly established hamburger shop and they asked me to create a new OMS for their store. Some of the store's requirements that need to be met by OMS are as follows:

- The store offers several types of hamburgers and some drinks (coca, pepsi and some milk tea). Each type has a specific price. The system needs to be able to represent these products.
- As for milk tea, there are 3 different types: traditional milk tea, jelly milk tea and pearl milk tea. But here jelly milk tea and pearl milk tea are both made from traditional milk tea. All drinks have 3 sizes to order: big, medium and small.
- Each hamburger has at least tomato, cheese and salad (Simple Hamburger). Can be combined with additional food (shrimp, beef ...) and can be ordered many times.
- Hamburger with certain extra food combinations are named, e.g., shrimp hamburger for the base pizza with shrimp. And all types of hamburger have 2 sizes: small and big

They also sent me a menu card for their store to better understand their menu items:



KIINBURGER			
MENU			
Hamburger		Price	
Simple Hamburger(Tomato, Cheese, Salad)		1.2	
Flava Hamburger(Tomato,Cheese,Salad,Flava Roast Chicken)		3.1	
Shrimp Hamburger(Tomato, Cheese, Salad, Shrimp)		2.9	
Beef Hambueger(Tomato, Cheese, Salad,Beef)		3.5	
Big Size		x2	
Drink	Price	Extra Food	Price
Coca	0.5	Tomato	0.6
Pepsi	0.5	Cheese	0.55
Traditional Milk Tea	1.5	Salad	0.3
Jelly Milk Tea	1.7	Flava Roast Chicken	1.55
Pearl Milk Tea	2	Shrimp	0.85
Big Size	x2	Beef	1.55
Medium Size	x1.5		

Figure 1: KIINBURGER menu

UML class diagram

Scenario analysis:

- First, we will create two interfaces, IDrink and IHamburger to general regulations on hamburgers and drinks.
- For the drink interface, we create 5 classes of 5 types of drinking water above and we have to determine that jelly milk tea and pearl milk tea are made from traditional milk tea. And we create an enum to determine the size of a drink, but only need to define for 3 types: coca, pepsi and traditional milk tea.

And jelly milk tea and pearl milk tea are made up of traditional milk tea so its size is also the size of the two types above

- For the Hamburger interface, 4 classes are also created, which are 4 types of hamburgers. Define that 3 types of Flava, Shrimp and Beef are all made up of simple hamburgers. We also have an enum to specify the size and also just create a size for the simple hamburger. But here we can add extra foods to the hamburger, so we will create an abstract class called ExtraFood and the subclasses are the corresponding extra foods to be added to the hamburger.
- Finally, we create an Order class so we can compile a list of customers ordering and print out the total price.

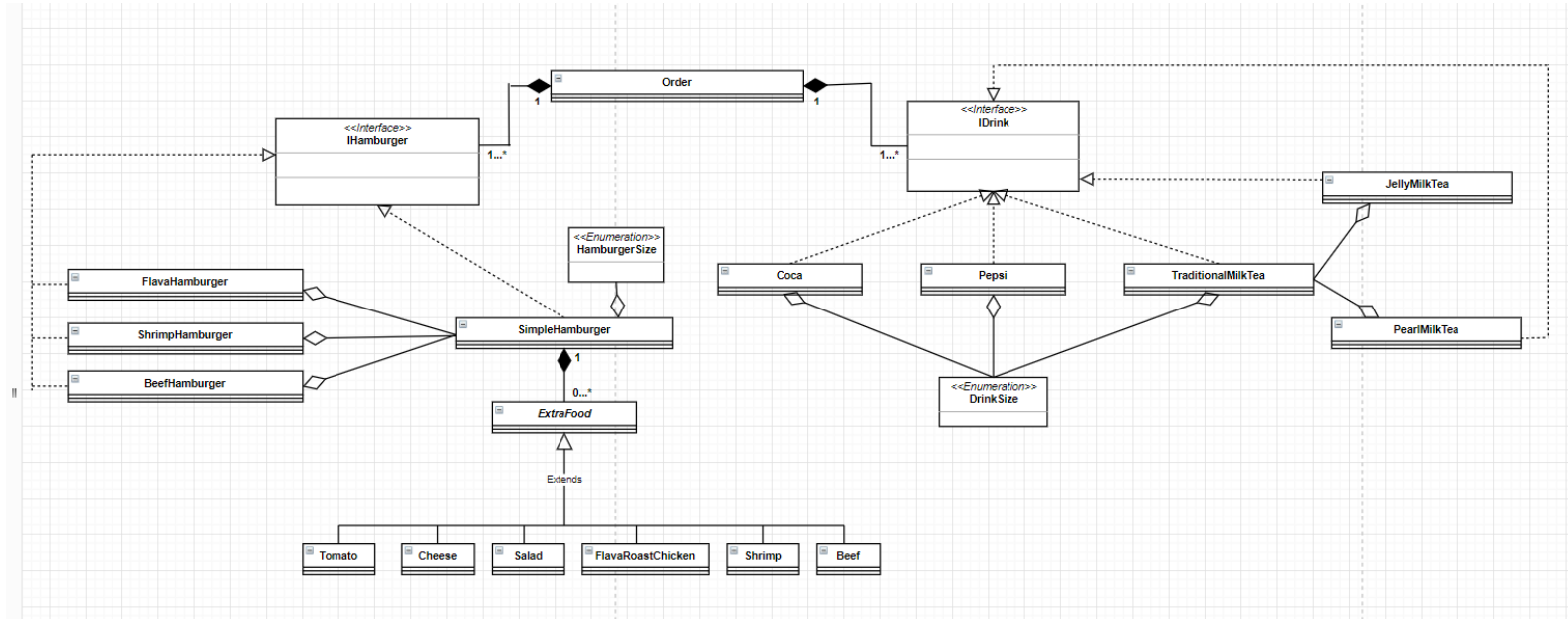


Figure 2: UML class diagram design

2. Develop a C# Application based on UML Class Diagram

Regarding the drink part, we create an IDrink interface with a method called GetPrice () to return the price of drinks.

```
8 references
interface IDrink
{
    9 references
    public double GetPrice();
}
```

Figure 3: IDrink interface

We create a DrinkSize enum to create the size of drink. Including 3 sizes are small to medium and large.

```
18 references
public enum DrinkSize
{
    Small = 1,
    Medium = 2,
    Big = 3
}
```

Figure 4: enum DrinkSize

Then we will create a Coca class that inherits the IDrink interface. Inside:

```
class Coca : IDrink
{
    private readonly double _price = 0.5;
    3 references
    public DrinkSize Size { get; private set; }
    0 references
    public Coca(DrinkSize size)
    {
        Size = size;
    }
    9 references
    public double GetPrice()
    {
        if (Size == DrinkSize.Big)
        {
            return _price * 2;
        }
        else if (Size == DrinkSize.Medium)
        {
            return _price * 1.5;
        }
        else
        {
            return _price;
        }
    }
}
```

Figure 5: Class Coca

- Create a variable `_price` and assign it the value 0.5 (which is the price of coca).
- Create a `Size` variable to determine the size of the coca.
- In the constructor, we specify that when creating a coca, we must enter the size and assign that input value to the variable `Size` above.
- Finally, the function `GetPrice ()` is the function that returns the price of coca. Here we will consider 3 cases where if the size is big, it will return the price of coca x2, if the size is medium, it will return the price of coca x1.5 and if the size is small, it will return the correct price of 0.5.

The Pepsi and TraditionalMilkTea classes are similar to the Coca class.


```
7 references
class TraditionalMilkTea : IDrink
{
    private readonly double _price = 1.5;
    7 references
    public DrinkSize Size { get; private set; }
    1 reference
    public TraditionalMilkTea(DrinkSize size)
    {
        Size = size;
    }
    9 references
    public double GetPrice()
    {
        if (Size == DrinkSize.Big)
        {
            return _price * 2;
        }
        else if (Size == DrinkSize.Medium)
        {
            return _price * 1.5;
        }
        else
        {
            return _price;
        }
    }
}
```

```
5 references
class Pepsi : IDrink
{
    private readonly double _price = 0.5;
    3 references
    public DrinkSize Size { get; private set; }
    1 reference
    public Pepsi(DrinkSize size)
    {
        Size = size;
    }
    9 references
    public double GetPrice()
    {
        if (Size == DrinkSize.Big)
        {
            return _price * 2;
        }
        else if (Size == DrinkSize.Medium)
        {
            return _price * 1.5;
        }
        else
        {
            return _price;
        }
    }
}
```

Figure 6: Class Pepsi and TraditionalMilkTea

There are 2 layers of JellyMilkTea and PearlMilkTea which will be a little different from the above 3 types because these 2 types of milk tea are made up of 1 Traditional milk tea.

```

3 references
class JellyMilkTea : IDrink
{
    private TraditionalMilkTea _traditional;
    private readonly double _price = 1.7;
    1 reference
    public JellyMilkTea(TraditionalMilkTea traditional)
    {
        _traditional = traditional;
    }

    9 references
    public double GetPrice()
    {
        if (_traditional.Size == DrinkSize.Big)
        {
            return _price * 2;
        }
        else if (_traditional.Size == DrinkSize.Medium)
        {
            return _price * 1.5;
        }
        else
        {
            return _price;
        }
    }
}

1 reference
class PearlMilkTea : IDrink
{
    private TraditionalMilkTea _traditional;
    private readonly double _price = 2;
    0 references
    public PearlMilkTea(TraditionalMilkTea traditional)
    {
        _traditional = traditional;
    }

    9 references
    public double GetPrice()
    {
        if (_traditional.Size == DrinkSize.Big)
        {
            return _price * 2;
        }
        else if (_traditional.Size == DrinkSize.Medium)
        {
            return _price * 1.5;
        }
        else
        {
            return _price;
        }
    }
}

```

Figure 7: Class JellyMilkTea and PearlMilkTea

- They also inherits the IDrink interface
- We also create a variable `_price` and assign a value to it.
- Since these 2 types are made from Traditional milk tea, we will create a variable of type TraditionalMilkTea `_traditional`.
- In the constructor, we specify to enter a TraditionMilkTea and assigning it to the `_tradition` variable created earlier.
- Although their prices are different from TraditionalHamburger, since they are made from a tradition hamburger their size is exactly the size of a tradition hamburger. So in the `GetPrice ()` function we will consider if the size of `_tradition` is big then we will return `_price x2`, if it is just then we return `_price x1.5`, and if it is small, we will only return `_price`.

Regarding the drink part, we create the IDrink interface with the following method:

- `AddExtraFood (extra: ExtraFood)`: add extra food to the hamburger
- `GetPrice ()`: Hamburger's price

- GetExtraFoodPrice (): The price of the extra food to be added
- GetTotalPrice (): The total cost of this hamburger

```
interface IHamburger
{
    7 references
    public void AddExtraFood(ExtraFood extra);
    8 references
    public double GetPrice();
    5 references
    public double GetTotalPrice();
    11 references
    public double GetExtraFoodPrice();
}
```

Figure 8: IHamburger interface

We create a HamburgerSize enum to create the size of drink

```
enum HamburgerSize
{
    small = 1,
    Big = 2
}
```

Figure 9: enum HamburgerSize

Then we create class ExtraFood has 1 variable is Price is the price of the extra feed and a constructor whose value is input is the price of the extra food and assigns it to the variable Price. The imported values will be imported from its subclasses. Corresponding to each type will have 1 different price.

```
abstract class ExtraFood
{
    2 references
    public double Price { get; private set; }
    6 references
    public ExtraFood(double price)
    {
        Price = price;
    }
}
```

```
1 reference
class Tomato : ExtraFood
{
    0 references
    public Tomato() : base(0.6)
    {
    }
}
1 reference
class Salad : ExtraFood
{
    0 references
    public Salad() : base(0.3)
    {
    }
}
1 reference
class Cheese : ExtraFood
{
    0 references
    public Cheese() : base(0.55)
    {
    }
}
1 reference
class FlavaRoastChicken : ExtraFood
{
    0 references
    public FlavaRoastChicken() : base(1.55)
    {
    }
}
1 reference
class Shrimp : ExtraFood
{
    0 references
    public Shrimp() : base(0.85)
    {
    }
}
1 reference
class Beef : ExtraFood
{
    0 references
    public Beef() : base(1.55)
    {
    }
}
```

Figure 10: Extra food

Next we will create the hamburgers for the restaurant. First we will create SimpleHamburger because this is the base hamburger, the other types of hamburgers are all made from this type. And it inherits the IHamburger interface

```
class SimpleHamburger : IHamburger
{
    private readonly double _price = 1.2;
    5 references
    public HamburgerSize Size { get; private set; }
    private List<ExtraFood> ExtraFoods;
    1 reference
    public SimpleHamburger(HamburgerSize size)
    {
        ExtraFoods = new List<ExtraFood>();
        Size = size;
    }

    7 references
    public void AddExtraFood(ExtraFood extra)
    {
        ExtraFoods.Add(extra);
    }

    8 references
    public double GetPrice()
    {
        if (Size == HamburgerSize.Big)
        {
            return _price * 2;
        }
        else return _price;
    }

    11 references
    public double GetExtraFoodPrice()
    {
        return ExtraFoods.Sum(t => t.Price);
    }

    5 references
    public double GetTotalPrice()
    {
        return GetPrice() + GetExtraFoodPrice();
    }
}
```

Figure 11: SimpleHamburger class

- We will create a Size variable to determine the size of the cake. And a list of extra foods is the variable Extrafoods, if we add more food, it will update to this list.
- In the constructor when we create the object, the input value is 1 size of the hamburger and the variable Size above equals that value. Then create a new ExtraFoods list. Then in the AddExtraFood () function, if you want to add this extra food, just Add it to the ExtraFoods list.

- In the GetPrice () function we will consider if the cake size is a big size then _price x2 will be returned and if the size is small, _price will be returned.
- In the GetExtraFoodPrice () function, we calculate and return the total amount of extra food added to the hamburger.
- Finally, the function GetTotalPrice () is the sum of the above two functions, ie the total value of the hamburger ordered including the extra food.

Then from the SimpleHamburger ships we will create Flavahamburger, ShrimpHamburger and BeefHamburger. Below are examples of ShrimpHamburger.

```
class ShrimpHamburger : IHamburger
{
    private SimpleHamburger _simple;
    private readonly double Price = 2.9;

    0 references
    public ShrimpHamburger(SimpleHamburger simple)
    {
        _simple = simple;
    }

    7 references
    public void AddExtraFood(ExtraFood extra)
    {
        _simple.AddExtraFood(extra);
    }

    11 references
    public double GetExtraFoodPrice()
    {
        return _simple.GetExtraFoodPrice();
    }

    8 references
    public double GetPrice()
    {
        if (_simple.Size == HamburgerSize.Big)
        {
            return Price * 2;
        }
        else return Price;
    }

    5 references
    public double GetTotalPrice()
    {
        return GetPrice() + GetExtraFoodPrice();
    }
}
```

Figure 12: ShrimpHamburger class

- It also inherits the IDrink interface
- We also create a variable `_price` and assign a value to it.
- Since this type is made from simple hamburgers, we'll create a type variable of type SimpleHamburger `_simple`.
- In the constructor, we specify to import a SimpleHamburger and assigning it to the `_simple` variable we created earlier.
- Although their prices are different from traditional hamburgers, since they are made from traditional hamburgers, their size is exactly the same as that of a traditional hamburger. So in the `GetPrice ()` function we will consider if the size of `_simple` is large then we will return `_price x2`, and if it is small, we will only return `_price`.
- Because the variable `_simple` is a SimpleHamburger cake, we can add extra food with the `AddExtraFood ()` function and calculate them with the `GetExtraFoodPrice ()` function.
- Finally, we calculate the total amount of the product using `GetTotalPrice ()` which includes `GetPrice ()` and `GetExtraFoodPrice ()` function.

Similar to the other 2 types of cakes are FlavaHamburger and BeefHamburger.

```
class BeefHamburger : IHamburger
{
    private SimpleHamburger _simple;
    private readonly double Price = 3.5;

    1 reference
    public BeefHamburger(SimpleHamburger simple)
    {
        _simple = simple;
    }

    7 references
    public void AddExtraFood(ExtraFood extra)
    {
        _simple.AddExtraFood(extra);
    }

    11 references
    public double GetExtraFoodPrice()
    {
        return _simple.GetExtraFoodPrice();
    }

    8 references
    public double GetPrice()
    {
        if (_simple.Size == HamburgerSize.Big)
        {
            return Price * 2;
        }
        else return Price;
    }

    5 references
    public double GetTotalPrice()
    {
        return GetPrice() + GetExtraFoodPrice();
    }
}
```

```
class FlavaHamburger : IHamburger
{
    private SimpleHamburger _simple;
    private readonly double Price = 3.1;

    1 reference
    public FlavaHamburger(SimpleHamburger simple)
    {
        _simple = simple;
    }

    7 references
    public void AddExtraFood(ExtraFood extra)
    {
        _simple.AddExtraFood(extra);
    }

    11 references
    public double GetExtraFoodPrice()
    {
        return _simple.GetExtraFoodPrice();
    }

    8 references
    public double GetPrice()
    {
        if (_simple.Size == HamburgerSize.Big)
        {
            return Price * 2;
        }
        else return Price;
    }

    5 references
    public double GetTotalPrice()
    {
        return GetPrice() + GetExtraFoodPrice();
    }
}
```

Figure 13: FlavaHamburger and BeefHamburger class

Finally, we create an Order class to create a multi-dish menu that combines and sums it all up.


```

class Order
{
    private List<IHamburger> _hamburgers;
    private List<IDrink> _drinks;
    1 reference
    public Order()
    {
        _hamburgers = new List<IHamburger>();
        _drinks = new List<IDrink>();
    }

    3 references
    public void AddHamburger(IHamburger hamburger)
    {
        _hamburgers.Add(hamburger);
    }

    3 references
    public void AddDrink(IDrink drink)
    {
        _drinks.Add(drink);
    }

    1 reference
    public double GetTotalPrice()
    {
        double sum = 0;
        foreach (var item in _hamburgers)
        {
            sum = item.GetTotalPrice() + sum;
        }
        foreach (var item in _drinks)
        {
            sum = item.GetPrice() + sum;
        }
        return sum;
    }
}

```

Figure 14: Class Order

- Finally, we create an Order class to create a multi-dish menu that combines and sums it all up.
- First we create 2 object lists of 1 hamburger list and 1 drink list. As soon as we initialize the order object, we will create 2 lists of hamburger and drink.
- Then we will create a function AddHamburger to add hamburgers and the AddDrink function to add drink to the list. Finally, we calculate the total amount in the 2 lists with the GetTotalPrice () function.

In the main function we will create hamburgers and drinks and print the price of each item then add them to the list and print out the total price.

```
static void Main(string[] args)
{
    Pepsi pepsi = new Pepsi(DrinkSize.Big);
    Console.WriteLine("Pepsi: " + pepsi.GetPrice());

    Coca coca = new Coca(DrinkSize.Small);
    Console.WriteLine("Coca: " + coca.GetPrice());

    TraditionalMilkTea traditional = new TraditionalMilkTea(DrinkSize.Small);
    Console.WriteLine("Traditional milk tea: " + traditional.GetPrice());

    JellyMilkTea jelly = new JellyMilkTea(traditional);
    Console.WriteLine("Jelly milk tea: " + jelly.GetPrice());

    SimpleHamburger simpleHamburger = new SimpleHamburger(HamburgerSize.Big);
    Console.WriteLine("Simple Hamburger: " + simpleHamburger.GetTotalPrice());
    FlavaHamburger flavaHamburger = new FlavaHamburger(new SimpleHamburger(HamburgerSize.small));
    Console.WriteLine("Flava Hamburger: " + flavaHamburger.GetTotalPrice());
    BeefHamburger beefHamburger = new BeefHamburger(new SimpleHamburger(HamburgerSize.small));
    Console.WriteLine("Beef Hamburger: " + beefHamburger.GetTotalPrice());

    Order order = new Order();


    order.AddHamburger(simpleHamburger);
    order.AddHamburger(flavaHamburger);
    order.AddHamburger(beefHamburger);
    order.AddDrink(jelly);
    order.AddDrink(coca);
    order.AddDrink(pepsi);
    order.AddDrink(traditional);

    Console.WriteLine("Order total price: " + order.GetTotalPrice());

    Console.ReadKey();
}
```

Figure 15: Main function

Output:

 F:\Assignment thầy vĩnh\A2\HamburgerStore\Ha

```
Pepsi: 1
Coca: 0.5
Traditional milk tea: 1.5
Jelly milk tea: 1.7
Simple Hamburger: 2.4
Flava Hamburger: 3.1
Beef Hamburger: 3.5
Order total price: 13.7
```

Figure 16: Output

3. UML after completing the program

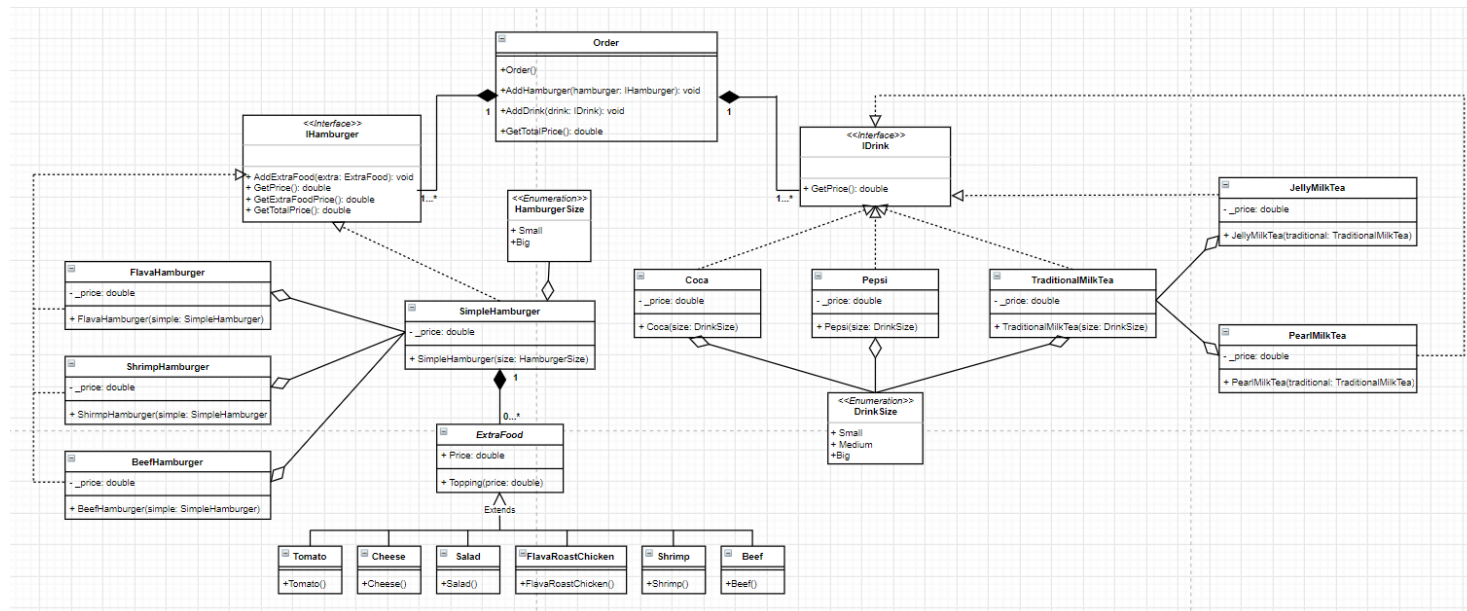
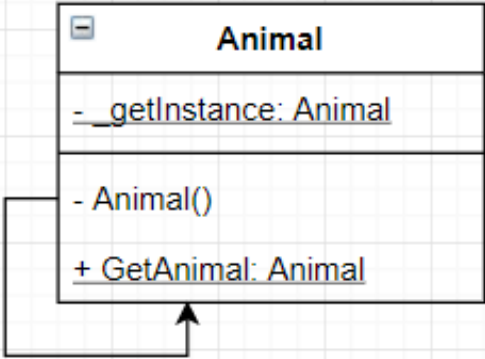


Figure 17: Complete UML class diagram

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

1. Creational pattern types

Pattern Name	Class Diagram	Short Explanation / Explain how the pattern works
Singleton Pattern	Because the constructor of this Animal class is declared private, from the outside we cannot create objects directly from this Animal class, but must create through GetAnimal () function. And in GetAnimal function will process code so that only one object can be created from this class. if trying to create more then just return that object.	A singleton pattern is a design pattern that only allows the creation of a single object from itself and grants access to the created instance. The Singleton design pattern is used when the user wants to create only one object from that class.

	 <pre> classDiagram class Animal { -_getInstance: Animal - Animal() + GetAnimal: Animal } Animal --> Animal </pre> <p>The diagram shows a class named 'Animal'. It has a private attribute <code>- _getInstance: Animal</code>, a private method <code>- Animal()</code>, and a public method <code>+ GetAnimal: Animal</code>. A self-association arrow points from the bottom of the class box back to the top.</p>	
Builder Pattern	<p>Below are the same implementation steps that are inherited from the IBuilder interface, but if we put in different materials, we will produce different products. If you put in wood and coconut, you will pay for that cottage, and if it is stone, you will give a stone house</p>	<p>Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.</p>

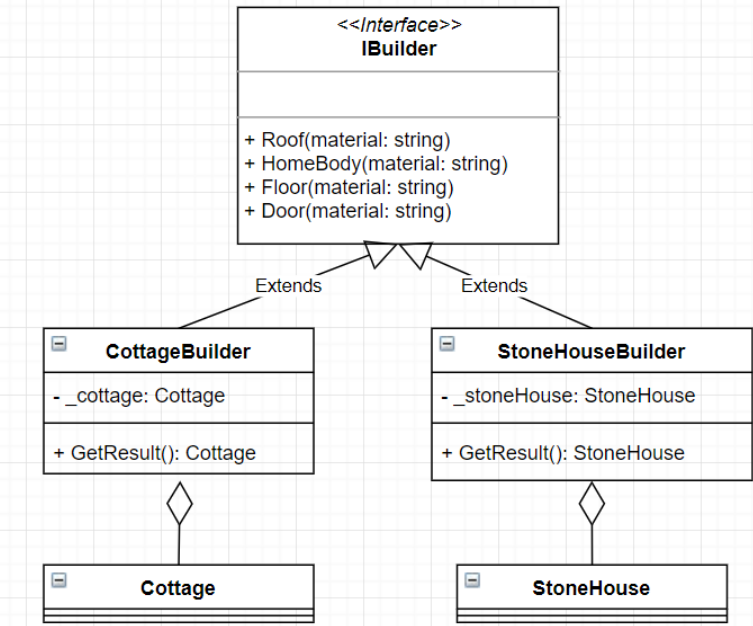
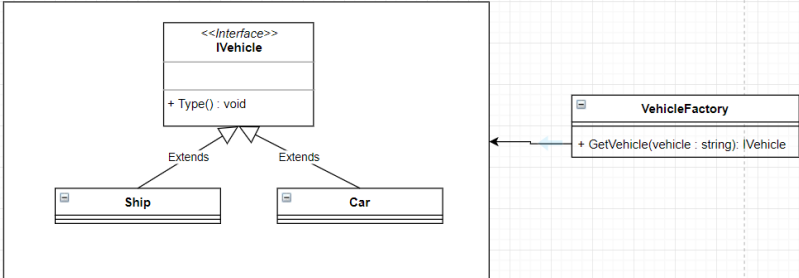
	 <pre> classDiagram class IBuilder { <<interface>> +Roof(material: string) +HomeBody(material: string) +Floor(material: string) +Door(material: string) } class CottageBuilder { -_cottage: Cottage +GetResult(): Cottage } class StoneHouseBuilder { -_stoneHouse: StoneHouse +GetResult(): StoneHouse } class Cottage class StoneHouse IBuilder < -- CottageBuilder IBuilder < -- StoneHouseBuilder CottageBuilder o-- Cottage StoneHouseBuilder o-- StoneHouse </pre>	
<p>Factory Method Pattern</p>	<p>Here we use factory methods to create objects such as car or ship with VehicleFactory class, regardless of the class to create of that object.</p>  <pre> classDiagram class IVehicle { <<interface>> +Type(): void } class Ship class Car class VehicleFactory { +GetVehicle(vehicle : string): IVehicle } IVehicle < -- Ship IVehicle < -- Car VehicleFactory --> IVehicle </pre>	<p>In class-based programming, Factory Method is a design pattern that uses factory methods to solve the problem of object creation without having to specify the exact class of the object to be created.</p>

Table 1: Creational pattern

2. Behavioral pattern types

Pattern Name	Class Diagram	Short Explanation / Explain how the pattern works
Command Pattern	<p>We have created a Stock class that will give the specific requirement to buy and sell. We have specific order classes BuyStock and SellStock that implement the Order interface that will perform actual order processing. A generated Broker class acts as an invoker object. It can receive and order.</p> <pre> classDiagram class Stock { +Name: string +Quantity: int +Buy(): void +Sell(): void } class Program { +main(): void } class IOrder { <<interface>> +Execute(): void } class BuyStock { +stock: Stock +BuyStock() } class SellStock { +stock: Stock +SellStock() } class Broker { +Orders: list<IOrder> +TakeOrder(): void +PlaceOrder(): void } Program --> Stock : Uses Program --> Broker : Uses Broker --> IOrder : Uses BuyStock -- > IOrder : Extends SellStock -- > IOrder : Extends BuyStock --> Stock SellStock --> Stock </pre>	<p>Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.</p>
Strategy Pattern	<p>Let's say we want to transport passengers from somewhere to the airport. There are a number of options such as driving, taking a taxi, the city bus or other car services. For some airports, metro and helicopters are also available as a means of transportation to the airport. Any transport method will have a traveler arrive at the airport, and we can be used interchangeably. Tourists choose a strategy (choice) based on a balance of cost, convenience and time to suit themselves.</p>	<p>Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.</p>

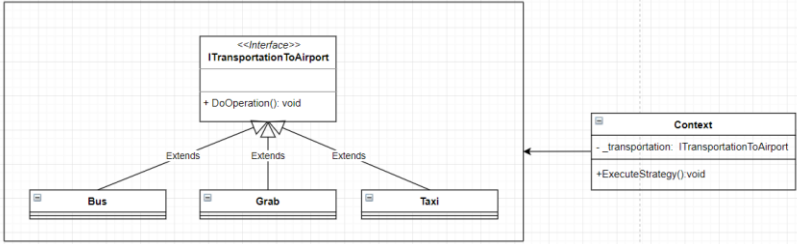
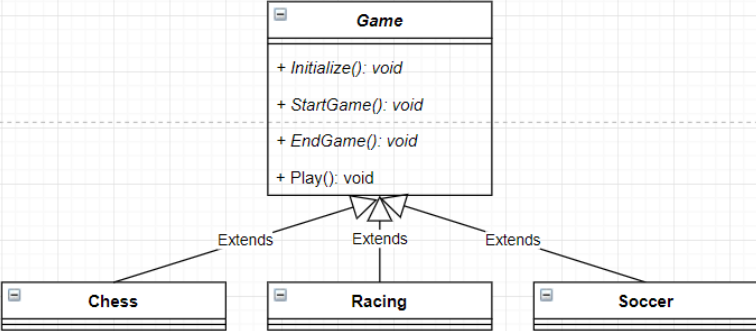
	 <pre> classDiagram class ITransportationToAirport { <<interface>> +DoOperation(): void } class Bus class Grab class Taxi class Context { -_transportation: ITransportationToAirport +ExecuteStrategy(): void } ITransportationToAirport < -- Bus ITransportationToAirport < -- Grab ITransportationToAirport < -- Taxi Context --> ITransportationToAirport </pre>	
Template Method	<p>Create an abstract class that defines the framework of abstract methods for the child classes to inherit and implement corresponding to each type.</p>  <pre> classDiagram class Game { +Initialize(): void +StartGame(): void +EndGame(): void +Play(): void } class Chess class Racing class Soccer Game < -- Chess Game < -- Racing Game < -- Soccer </pre>	<p>The Template Method is a behavioral design pattern that defines the framework of an algorithm in the superclass is usually an abstract class, and subclasses override the algorithm's specific steps without changing its structure.</p>

Table 2: Behavioral pattern

3. Structural pattern types

Pattern Name	Class Diagram	Short Explanation / Explain how the pattern works
Adapter Pattern	<p>Here we have an IBird Interface which will have an AnimalSound function to make noise for objects created from its subclasses and an IToyDuck interface also has a Squeak () function to create a separate call for the objects created from. subclass of. Here we have 2 separate interfaces and have separate calls. But if the customer wants to create a ToyDuck but the Bird</p>	<p>Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.</p>

	<p>calls, we will use an AdapterToyDuck class to do this. It creates a ToyDuck but when created it will create a bird in it and its Squeak () function will return the sound of a bird that the user requested.</p> <pre> classDiagram class IBird { <<Interface>> +Fly(): void +MakeASound(): void } class Sparrow { } class AdapterToyDuck { -_bird: IBird +AdapterToyDuck(bird: IBird) } class PlasticToyDuck { } class IToyDuck { <<Interface>> +Squeak(): void } IBird < -- Sparrow IBird < -- AdapterToyDuck IBird < -- PlasticToyDuck AdapterToyDuck o-- IBird AdapterToyDuck < -- IToyDuck PlasticToyDuck < -- IToyDuck </pre>	
Decorator Pattern	<p>Here we use the RedShape class to add the details we want to the subclasses of the IShape interface, here are the Circle and Rectangle classes without changing their structure.</p> <pre> classDiagram class IShape { <<Interface>> +Draw(): void } class Circle { } class Rectangle { } class RedShape { -_shape: IShape +RedShape(shape: IShape) +Draw(): void } IShape < -- Circle IShape < -- Rectangle IShape < -- RedShape RedShape o-- IShape </pre>	Decorator pattern allows a user to add new functionality to an existing object without altering its structure
Bridge Pattern	<p>Suppose you have an interface called IShape and have 2 subclasses, Triangle and Square. Now you want to add a color, say red or blue, to those two shapes. To get around that, just create a few more classes such as RedSquare, RedTriangle, BlueSquare and BlueStriangle. But assuming there are more shapes or more colors, the number of subclasses will create a lot. So to solve the above problem, we will create an additional</p>	Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be

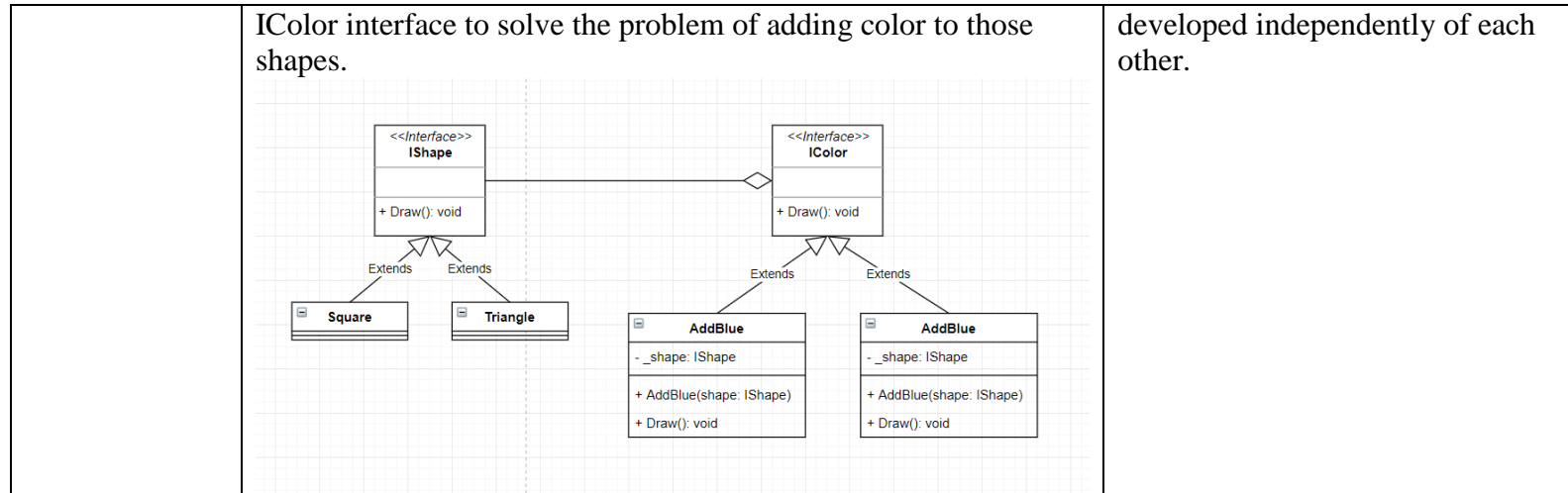


Table 3: Structural pattern

III. Reference