

# ADVANCED PROGRAMMING

## REPORT ASSIGNMENT 2

STUDENT ID: GCD18457

Assessor Name: HOANG NHU VINH

## ASSIGNMENT 2 FRONT SHEET

<b>Qualification</b>	<b>BTEC Level 5 HND Diploma in Computing</b>		
<b>Unit number and title</b>	Unit 20: Advanced Programming		
<b>Submission date</b>		<b>Date Received 1st submission</b>	
<b>Re-submission Date</b>		<b>Date Received 2nd submission</b>	
<b>Student Name</b>	Tran Quang Huy	<b>Student ID</b>	GCD18457
<b>Class</b>	GCD0604	<b>Assessor name</b>	Hoang Nhu Vinh
<b>Student declaration</b> 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.			
		<b>Student's signature</b>	Huy

### Grading grid

P3	P4	M3	M4	D3	D4

<input type="checkbox"/> <b>Summative Feedback:</b>			<input type="checkbox"/> <b>Resubmission Feedback:</b>		
<b>Grade:</b>		<b>Assessor Signature:</b>		<b>Date:</b>	
<b>Lecturer Signature:</b>					

## Contents

<b>INTRODUCTION</b> .....	1
<b>1. Implementation</b> .....	2
1.1. Scenario .....	2
1.2. Class diagram .....	3
1.3. Structure of application .....	4
1.4. Code Snippet .....	5
1.5. Result.....	15
1.6. Compare Console application and Requirement .....	20
<b>2. Scenario Investigation</b> .....	21
2.1. A range of Design Patterns. ....	21
2.2. The most appropriate design pattern from a range with a series of given scenarios. ....	29
<b>3. Evaluation</b> .....	31
3.1. The use of Design Pattern for the given purpose specified .....	31
3.1. Evaluate a range of design patterns against the range of given scenarios with justification of your choices. ....	36
<b>CONCLUSION</b> .....	49
<b>References</b> .....	50

## TABLE OF TALBES

Table 1. Highlands Coffee Menu .....	2
Table 2. Compare console application and Requirement.....	20
Table 3. A range of Design Patterns. (viblo, n.d.) .....	28
Table 4. Compare Decorator with another Pattern.....	30
Table 5. Summary compare before and after using Decorator Pattern .....	35
Table 6. Compare some Pattern can implement to odering beverage .....	36
Table 7. Compare Decorator Pattern with related Patterns .....	48

## TABLE OF FIGURES

Figure 1. Beverage Class Diagram (Decorator Pattern) .....	3
Figure 2. Class diagram before using design pattern .....	31
Figure 3. Class Diagram after using Design Pattern .....	32
Figure 4. Beverage class .....	33
Figure 5. Four concrete components.....	33
Figure 6. Class diagram of using Builder Pattern for HighLands Coffee .....	37
Figure 7. Class diagram of using Simple Factory Pattern for HighLands Coffee .....	43

## TABLE OF PICTURES

Picture 1. Structure of application.....	4
Picture 2. Structure of class Highlands .....	4
Picture 3. Structure of class HighlandsMenu.....	4
Picture 4. Result of console application loading.....	15
Picture 5. Result of Show Coffee adn make coffee offer.....	15
Picture 6. Result after choose coffee and type 'n'.....	16
Picture 7. Result after choose coffee.....	16
Picture 8. Result after choose coffee and type 'y'.....	17
Picture 9. Result when user confirm coffee .....	17
Picture 10. Result when user want to add condiments.....	18
Picture 11. Result when user add condiments.....	18
Picture 12. Result when user finish added condiments.....	19
Picture 13. Result of print reciept .....	19
Picture 14. Result after using Builder Pattern for ordering beverage .....	41
Picture 15. Result after using Simple Factory Pattern for ordering beverage.....	47

## INTRODUCTION

In this document will show more clearly about using Design Pattern to build a real program match a real problem in real life. The Document includes 3 parts:

- **Implementation program using Design Pattern:** By using Decorator Pattern, the program will solve the problems.
- **Scenario Investigation:** To talk about 9 Design Patterns for 3 kinds of Design Pattern and how the system was using Decorator Pattern to build.
- **Evaluation:** The most appropriate design pattern from a range with a series of given scenarios. And Evaluate a range of design patterns against the range of given scenarios with justification of your choices.

A design pattern provides a general reusable solution for the common problems occurs in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing well tested, proven development/design paradigm. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using the design patterns, you can make your code more flexible, reusable and maintainable.

After this report, I also have a lot of knowledge about OOP, specially is about using Design Pattern in a real problem to build a program. Each Design Pattern have special job and solve a difference problem.

## 1. Implementation

### 1.1. 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 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 1. Highlands Coffee Menu*

## 1.2. Class diagram

**The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

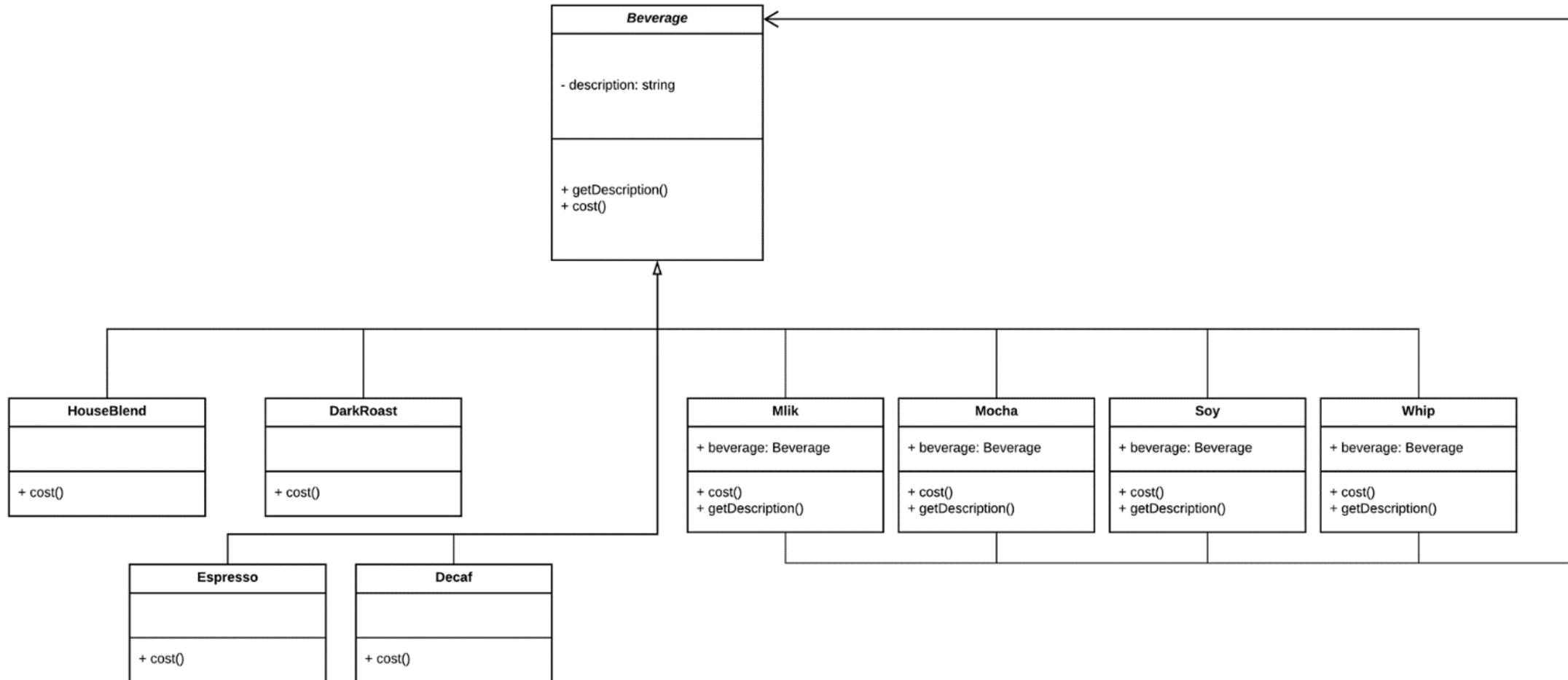
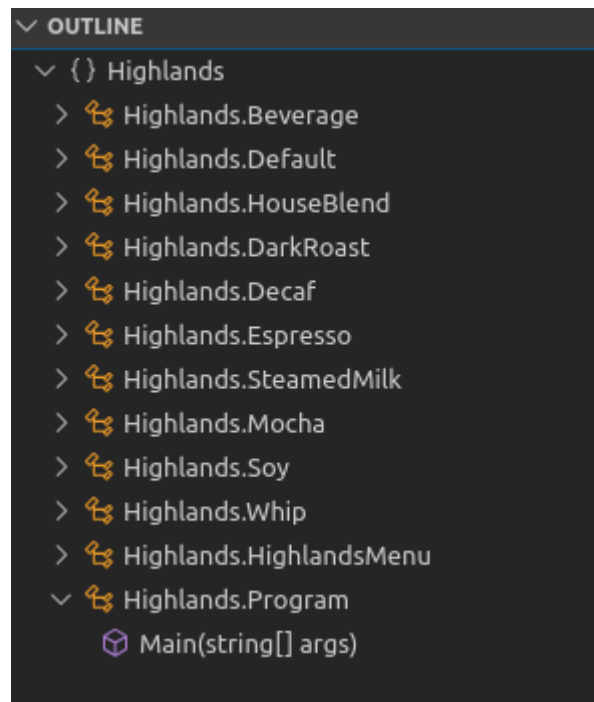


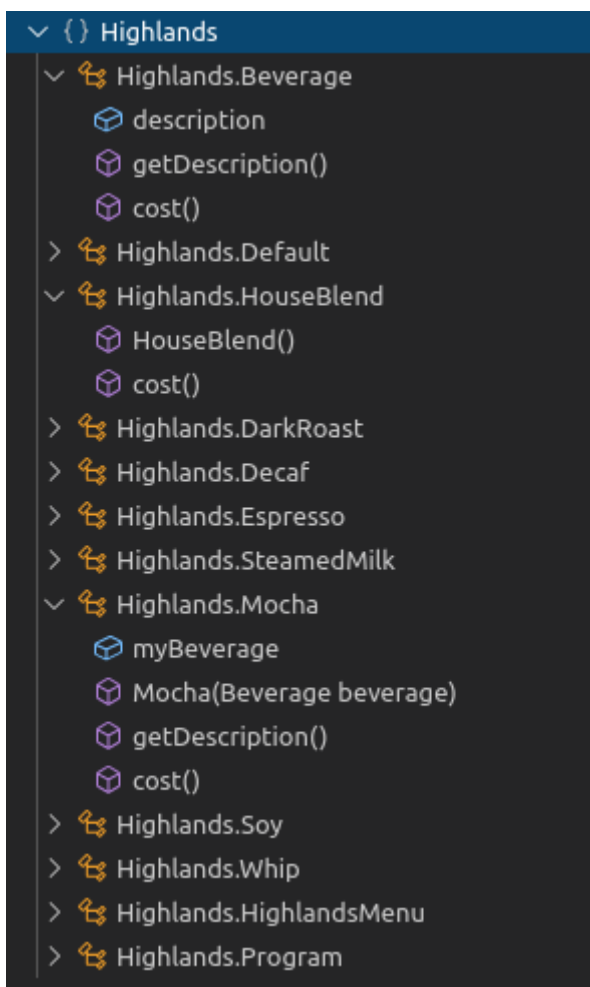
Figure 1. Beverage Class Diagram (Decorator Pattern)



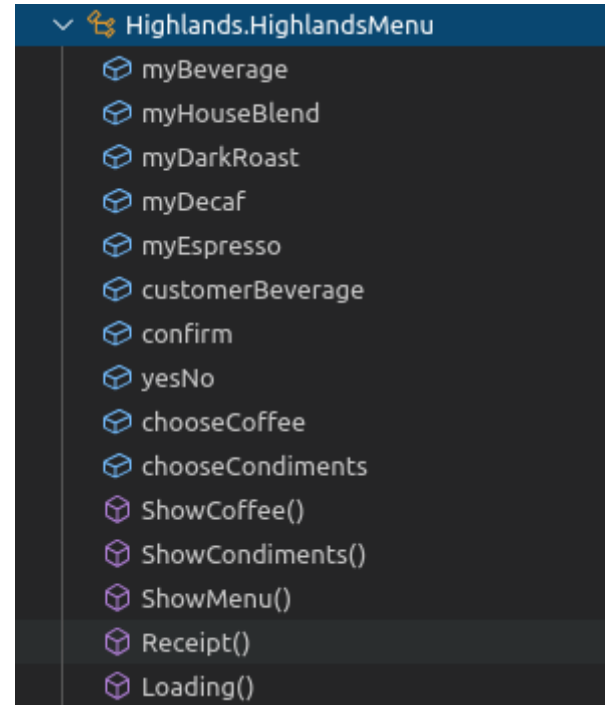
### 1.3. Structure of application



Picture 1. Structure of application



Picture 2. Structure of class Highlands



Picture 3. Structure of class HighlandsMenu

## 1.4. Code Snippet

### 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 **Espresso**. 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 **Espresso**. 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;
    }
}
```

### Class HighLandsMenu:

This class will call in Main of program, so this is one of the most important part to complete the console application using Decorator Design Pattern.

Firstly, I create new Beverage to get base Description and Cost for each beverage and customerBeverage for Customer's order beverage and some field such as **yesNo: Char** used to ask user "Yes or No", **confirm: bool** to make sure that order is finished, **chooseCoffe: int** and **chooseCondiments: int** to get a choice from user want what kind of coffee and condiments.

```
//Create new Beverage to get base Description and Cost
Beverage myBeverage = new Default();
Beverage myHouseBlend = new HouseBlend();
Beverage myDarkRoast = new DarkRoast();
Beverage myDecaf = new Decaf();
Beverage myEspresso = new Espresso();

//Create a Customer's order Beverage
Beverage customerBeverage;
//Confirm the customer's order
bool confirm = true;
//Char will use to ask user Yes or No
char yesNo = 'N';
int chooseCoffee;
int chooseCondiments;
```

### Method Loading():

```
public void Loading()
{
    Console.WriteLine("\tWELCOME TO HIGHLANDS COFFEE CONSOLE APPLICATION\t");
    Console.WriteLine("-----");
    ShowMenu();
    Console.WriteLine("please dont turn off power,Order system is opening!");
    Console.Write("[");
    for (int i = 0; i <= 65; i++)
    {
        Console.Write("#");
        System.Threading.Thread.Sleep(30);
    }
    Console.Write("]");
    System.Threading.Thread.Sleep(1200);
    Console.Clear();
}
```

To make the application more realistic, I have created method **Loading()** to print in screen the Highlands menu and the text like starting the program by using **loop** and **System Threading Sleep** to print the char '#'.

### Method **ShowMenu()**:

From the Default Beverage that I create in the first line of class HighLandsMenu, I used it and create Condiments to get a Description and Cost for each condiment.

After I got Coffee, Condiments Description and Cost, I have printed it into screen.

```
public void ShowMenu()
{
    Beverage mySteamedMilk = new SteamedMilk(myBeverage);
    Beverage myMocha = new Mocha(myBeverage);
    Beverage mySoy = new Soy(myBeverage);
    Beverage myWhip = new Whip(myBeverage);
    Console.WriteLine("----\tHighlands Coffee Menu\t----");

    Console.BackgroundColor = ConsoleColor.Black;
    Console.WriteLine("== Coffees");
    Console.ResetColor();
    Console.WriteLine($" \t{myHouseBlend.getDescription()}: \t\t${myHouseBlend.cost()}");
    Console.WriteLine($" \t{myDarkRoast.getDescription()}: \t\t${myDarkRoast.cost()}");
    Console.WriteLine($" \t{myDecaf.getDescription()}: \t\t\t${myDecaf.cost()}");
    Console.WriteLine($" \t{myEspresso.getDescription()}: \t\t${myEspresso.cost()}");
    Console.WriteLine();
    Console.BackgroundColor = ConsoleColor.Black;
    Console.WriteLine("== Condiments");
    Console.ResetColor();
    Console.WriteLine($" {mySteamedMilk.getDescription()}: \t\t${mySteamedMilk.cost()}");
    Console.WriteLine($" {myMocha.getDescription()}: \t\t\t${myMocha.cost()}");
    Console.WriteLine($" {mySoy.getDescription()}: \t\t\t${mySoy.cost()}");
    Console.WriteLine($" {myWhip.getDescription()}: \t\t\t${myWhip.cost()}");
}
```

### Method **Receipt()**:

The following method that I want to create is **Receipt()**, from the customer ordering I will print into screen what Coffee, Condiments, and Price for customer's beverage.

```
public void Receipt()
{
    Console.WriteLine("\n--- \tCustomer order \t---");

    Console.ForegroundColor = ConsoleColor.Green;
    Console.Write("--Beverage--\t");
    Console.ResetColor();

    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("--Condiments--");
    Console.ResetColor();

    Console.WriteLine();

    Console.WriteLine($"{customerBeverage.getDescription()}");
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.Write("--Price: ");
    Console.ResetColor();
    Console.Write($"{customerBeverage.cost()}");

    Console.WriteLine("\n\nThank you for using our service!");
}
```



## Method **ShowCoffee()**:

In this method, I will show again Coffee Menu but without Condiment, Then User can make a coffee offer through type into screen the number that assigned for each coffee. I also used Try-Catch to handle error from the user's input (When the user's input is empty, the system knows that the user wants to say "YES" or confirmed). After Confirm with system by type 'y','Y' the condiment menu will print out.

```
public void ShowCoffee()
{
    Console.WriteLine("----\tHighlands Coffee\t----");
    Console.WriteLine($" [1] \t{myHouseBlend.getDescription()}:
\t\t${myHouseBlend.cost()}");
    Console.WriteLine($" [2] \t{myDarkRoast.getDescription()}: \t\t${myDarkRoast.cost()}");
    Console.WriteLine($" [3] \t{myDecaf.getDescription()}: \t\t\t${myDecaf.cost()}");
    Console.WriteLine($" [4] \t{myEspresso.getDescription()}: \t\t${myEspresso.cost()}");
    Console.WriteLine("\nChoose Coffee: ");
    chooseCoffee = Int32.Parse(Console.ReadLine());
    try
    {
        while (confirm)
        {
            switch (chooseCoffee)
            {
                case 0:
                    break;
                case 1:
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine($"{myHouseBlend.getDescription()}:
${myHouseBlend.cost()}");
                    Console.ResetColor();
                    customerBeverage = new HouseBlend();
                    break;
                case 2:
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine($"{myDarkRoast.getDescription()}:
${myDarkRoast.cost()}");
                    Console.ResetColor();
                    customerBeverage = new DarkRoast();
                    break;
                case 3:
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine($"{myDecaf.getDescription()}: ${myDecaf.cost()}");
                    Console.ResetColor();
                    customerBeverage = new Decaf();
                    break;
                case 4:
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine($"{myEspresso.getDescription()}:
${myEspresso.cost()}");
                    Console.ResetColor();
                    customerBeverage = new Espresso();
                    break;
                default:
                    break;
            }
        }
    }
```

```

if (chooseCoffee > 4)
{
    Console.WriteLine("Please Coffee in menu");
    Console.Write("Choose Coffee: ");
    chooseCoffee = Int32.Parse(Console.ReadLine());
}
else
{
    Console.Write("-Do you want this Coffee? (Y: Yes, N: No): ");
    yesNo = char.Parse(Console.ReadLine());
    if (yesNo == 'Y' || yesNo == 'y')
        break;
    Console.Write("Choose Coffee: ");
    chooseCoffee = Int32.Parse(Console.ReadLine());
}
}
}
catch (System.Exception)
{
    try
    {
        Console.Write("-Do you want to add more Condiments? (Y: Yes, N: No): ");
        yesNo = char.Parse(Console.ReadLine());
        if (yesNo == 'Y' || yesNo == 'y')
        {
            ShowCondiments();
        }
        else
        {
            confirm = false;
            Receipt();
        }
    }
    catch (System.Exception)
    {
        ShowCondiments();
    }
}
if (confirm)
{
    try
    {
        Console.Write("-Do you want to add more Condiments? (Y: Yes, N: No): ");
        yesNo = char.Parse(Console.ReadLine());
        if (yesNo == 'Y' || yesNo == 'y')
        {
            ShowCondiments();
        }
        else Receipt();
    }
    catch (System.Exception)
    {
        ShowCondiments();
    }
}
}
}

```

### Method **ShowCondiments()**:

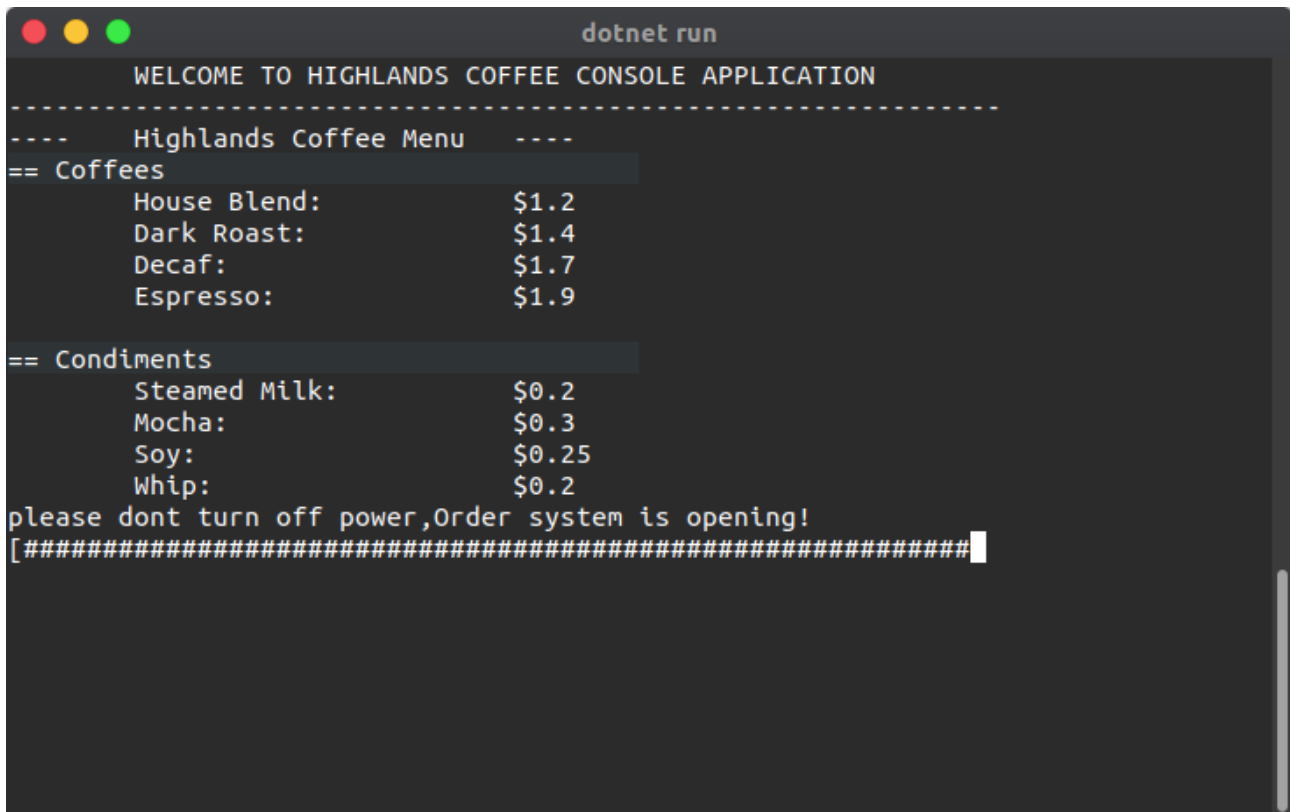
In this method, the Condiment Menu allows to user add condiment by type a number that assigned for each condiment and after add 1 condiment the system will ask the user to add more (loop). If a user wants to cancel additional or finished additional, User can press 'Enter' to done (I used try-catch to do this).

```
public void ShowCondiments()
{
    Beverage mySteamedMilk = new SteamedMilk(myBeverage);
    Beverage myMocha = new Mocha(myBeverage);
    Beverage mySoy = new Soy(myBeverage);
    Beverage myWhip = new Whip(myBeverage);
    Console.WriteLine("----\tHighlands Condimentes\t----");
    Console.WriteLine($" [1] {mySteamedMilk.getDescription()}:
\t\t\t{mySteamedMilk.cost()}");
    Console.WriteLine($" [2] {myMocha.getDescription()}: \t\t\t{myMocha.cost()}");
    Console.WriteLine($" [3] {mySoy.getDescription()}: \t\t\t{mySoy.cost()}");
    Console.WriteLine($" [4] {myWhip.getDescription()}: \t\t\t{myWhip.cost()}");

    try
    {
        Console.Write("\nAdd Condimentes (Press 'Enter' to done): ");
        chooseCondiments = Int32.Parse(Console.ReadLine());
        while (confirm)
        {
            switch (chooseCondiments)
            {
                case 1:
                    customerBeverage = new SteamedMilk(customerBeverage);
                    Console.WriteLine($"Added:{mySteamedMilk.getDescription()}");
                    break;
                case 2:
                    customerBeverage = new Mocha(customerBeverage);
                    Console.WriteLine($"Added:{myMocha.getDescription()}");
                    break;
                case 3:
                    customerBeverage = new Soy(customerBeverage);
                    Console.WriteLine($"Added:{mySoy.getDescription()}");
                    break;
                case 4:
                    customerBeverage = new Whip(customerBeverage);
                    Console.WriteLine($"Added:{myWhip.getDescription()}");
                    break;
                default:
                    break;
            }
            if (chooseCondiments > 4)
            {
                Console.WriteLine("Please choose Condimentes in menu");
            }
            Console.Write("\nAdd Condimentes (Press 'Enter' to done): ");
            chooseCondiments = Int32.Parse(Console.ReadLine());
        }
    }
    catch (System.Exception)
    {
        confirm = false;
        Receipt();
    }
}
```

## 1.5. Result

Console Application Loading:

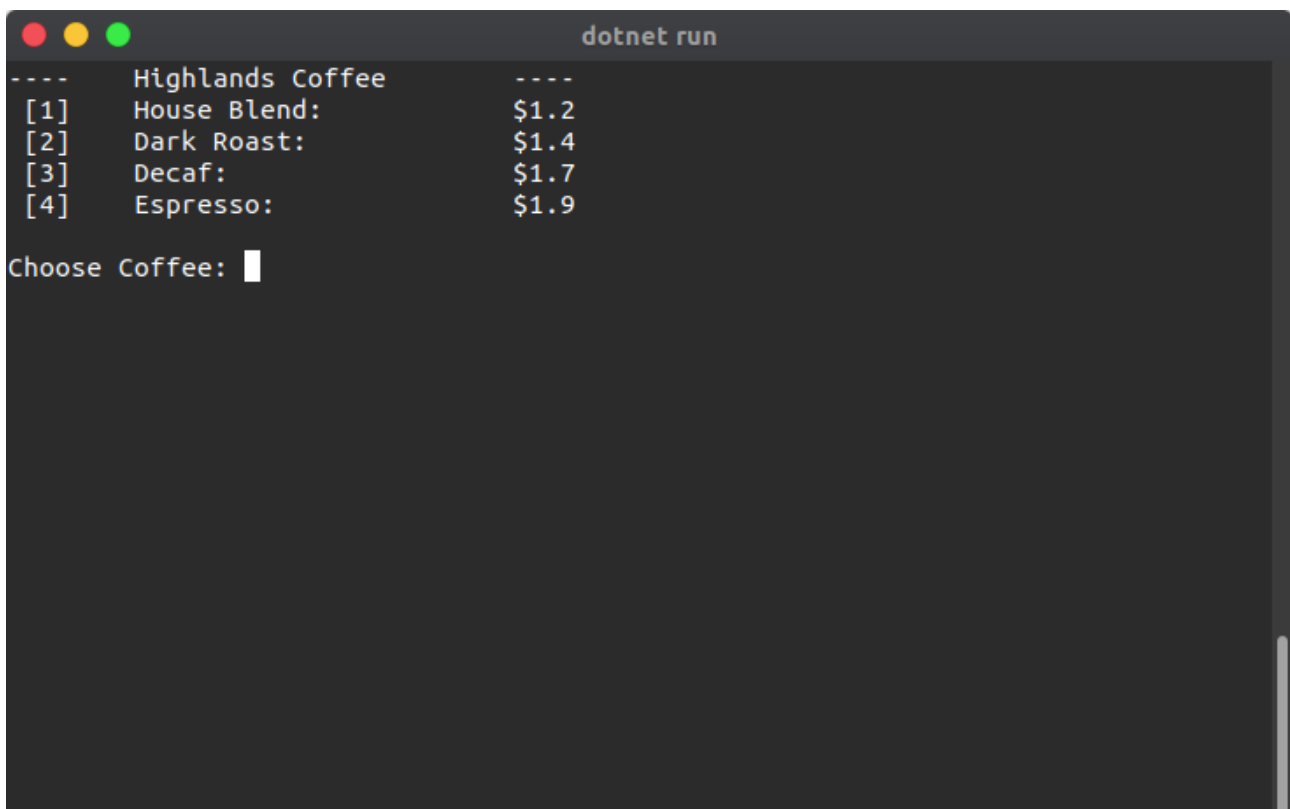


```
dotnet run
WELCOME TO HIGHLANDS COFFEE CONSOLE APPLICATION
-----
---- Highlands Coffee Menu ----
== Coffees
House Blend:      $1.2
Dark Roast:       $1.4
Decaf:            $1.7
Espresso:         $1.9

== Condiments
Steamed Milk:     $0.2
Mocha:            $0.3
Soy:              $0.25
Whip:             $0.2
please dont turn off power,Order system is opening!
[#####]
```

Picture 4. Result of console application loading

Show Coffee and make coffee offer:

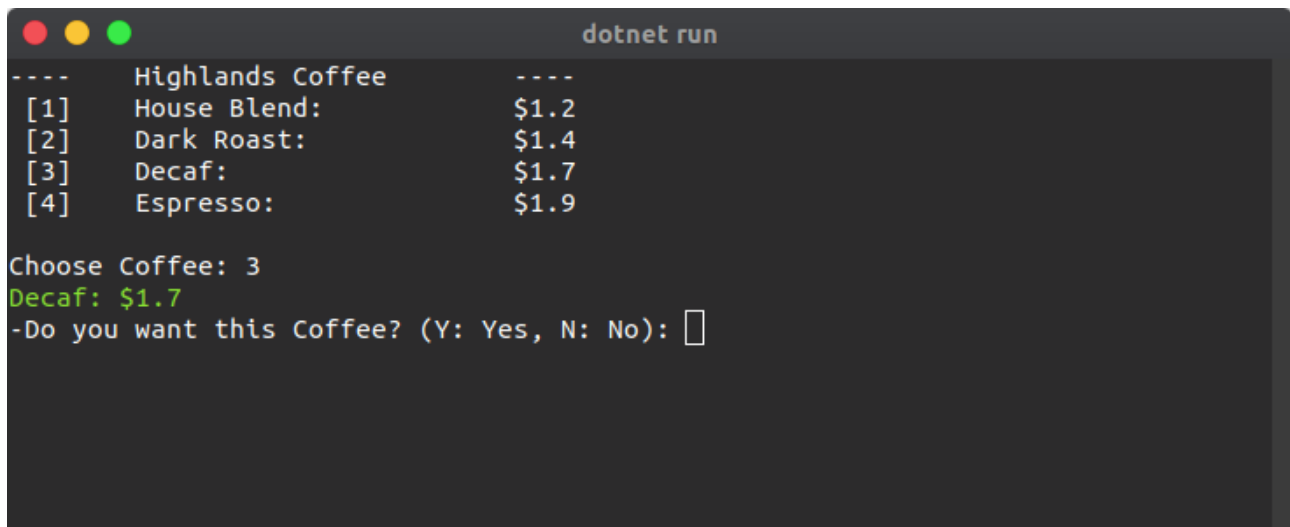


```
dotnet run
---- Highlands Coffee ----
[1] House Blend:      $1.2
[2] Dark Roast:       $1.4
[3] Decaf:            $1.7
[4] Espresso:         $1.9

Choose Coffee: 
```

Picture 5. Result of Show Coffee adn make coffee offer

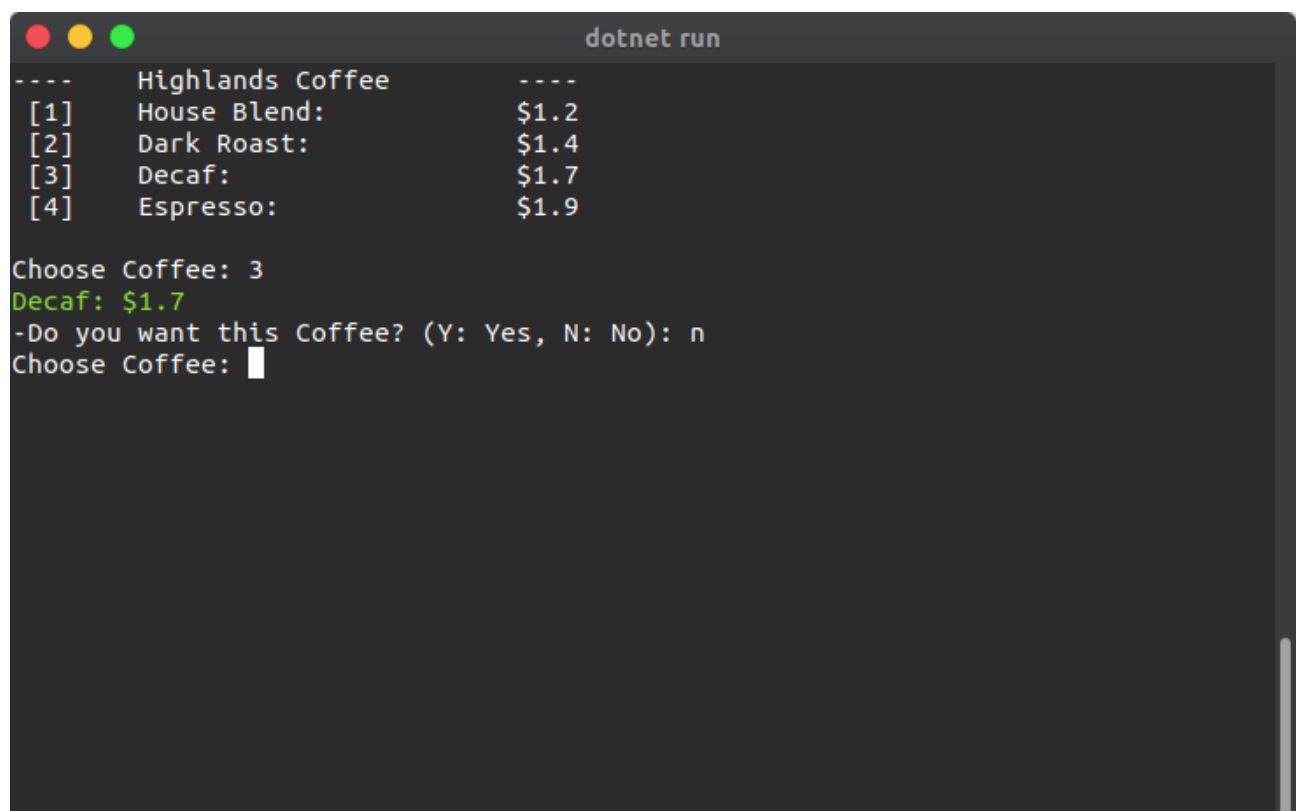
After choosing coffee, the system will make sure the coffee is right or mistake, the user can change the coffee order by type 'n' or 'N' to re-choice. If the Coffee correct, the user can press 'Enter' or type 'y' or 'Y' to continue.

A terminal window titled "dotnet run" with a dark background. It displays a menu for "Highlands Coffee" with four options: [1] House Blend: \$1.2, [2] Dark Roast: \$1.4, [3] Decaf: \$1.7, and [4] Espresso: \$1.9. The user has entered "3" to choose Decaf. The terminal shows "Decaf: \$1.7" in green text. Below that, it asks "-Do you want this Coffee? (Y: Yes, N: No):" followed by a cursor.

```
dotnet run
---- Highlands Coffee ----
[1] House Blend: $1.2
[2] Dark Roast: $1.4
[3] Decaf: $1.7
[4] Espresso: $1.9

Choose Coffee: 3
Decaf: $1.7
-Do you want this Coffee? (Y: Yes, N: No):
```

Picture 7. Result after choose coffee

A terminal window titled "dotnet run" with a dark background. It displays the same menu as Picture 7. The user has entered "3" to choose Decaf. The terminal shows "Decaf: \$1.7" in green text. Below that, it asks "-Do you want this Coffee? (Y: Yes, N: No):" and the user has entered "n". The terminal then prompts "Choose Coffee:" followed by a cursor.

```
dotnet run
---- Highlands Coffee ----
[1] House Blend: $1.2
[2] Dark Roast: $1.4
[3] Decaf: $1.7
[4] Espresso: $1.9

Choose Coffee: 3
Decaf: $1.7
-Do you want this Coffee? (Y: Yes, N: No): n
Choose Coffee:
```

Picture 6. Result after choose coffee and type 'n'

```
dotnet run
---- Highlands Coffee ----
[1] House Blend:      $1.2
[2] Dark Roast:       $1.4
[3] Decaf:            $1.7
[4] Espresso:        $1.9

Choose Coffee: 3
Decaf: $1.7
-Do you want this Coffee? (Y: Yes, N: No): n
Choose Coffee: 4
Espresso: $1.9
-Do you want this Coffee? (Y: Yes, N: No): y
-Do you want to add more Condiments? (Y: Yes, N: No): ☐
```

Picture 8. Result after choose coffee and type 'y'

When the user makes sure that coffee is right, the system continues to ask the user to add more condiments or not. If the user chooses “No” by type ‘n’ or ‘N’, the system will print the receipt.

```
dotnet run
---- Highlands Coffee ----
[1] House Blend:      $1.2
[2] Dark Roast:       $1.4
[3] Decaf:            $1.7
[4] Espresso:        $1.9

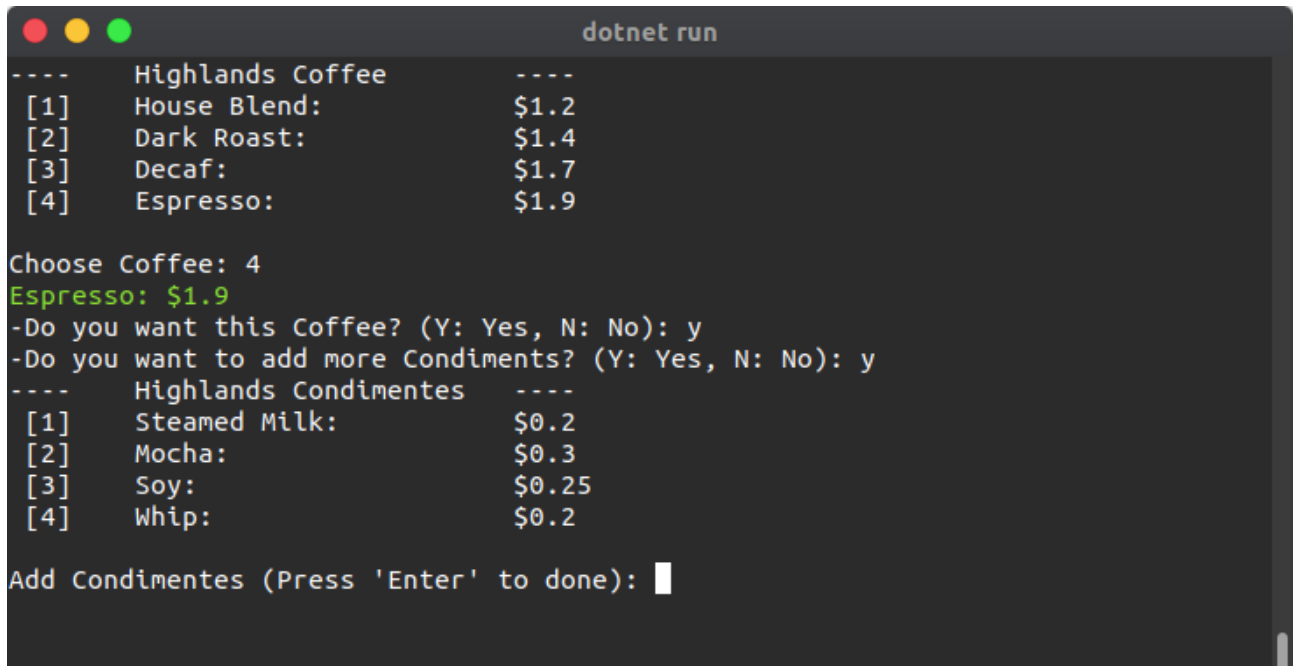
Choose Coffee: 3
Decaf: $1.7
-Do you want this Coffee? (Y: Yes, N: No): n
Choose Coffee: 4
Espresso: $1.9
-Do you want this Coffee? (Y: Yes, N: No): y
-Do you want to add more Condiments? (Y: Yes, N: No): n

--- Customer order ---
--Beverage--      --Condiments--
Espresso
--Price: $1.9

Thank you for using our service!
█
```

Picture 9. Result when user confirm coffee

Besides that, if users choose “YES” by type ‘y’ or ‘Y’, the system will print condiments menu to choose.



```
dotnet run
---- Highlands Coffee ----
[1] House Blend:      $1.2
[2] Dark Roast:       $1.4
[3] Decaf:            $1.7
[4] Espresso:         $1.9

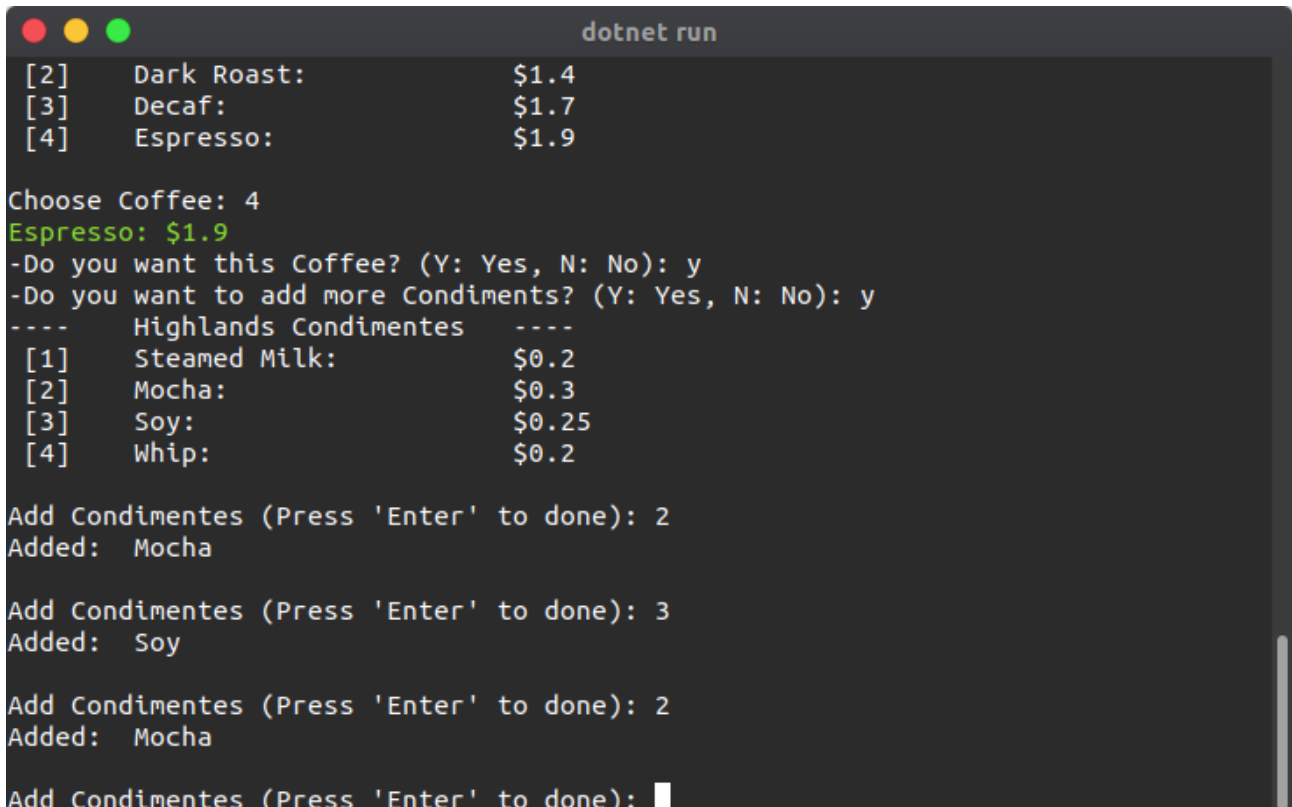
Choose Coffee: 4
Espresso: $1.9
-Do you want this Coffee? (Y: Yes, N: No): y
-Do you want to add more Condiments? (Y: Yes, N: No): y
---- Highlands Condimentes ----
[1] Steamed Milk:     $0.2
[2] Mocha:            $0.3
[3] Soy:              $0.25
[4] Whip:             $0.2

Add Condimentes (Press 'Enter' to done):
```

Picture 10. Result when user want to add condiments

When the condiment menu shows up, the application will be asking the user to choose the Condiments. In this case, there are 2 ways to continue: User can add condiments by type a number that assigned for each condiment.

Another way is if a user doesn't want to add any more condiments then just press 'Enter' on the keyboard and if user want to finish added condiments can also use this way. After that, the receipt will print out.



```
dotnet run
[2] Dark Roast:      $1.4
[3] Decaf:           $1.7
[4] Espresso:        $1.9

Choose Coffee: 4
Espresso: $1.9
-Do you want this Coffee? (Y: Yes, N: No): y
-Do you want to add more Condiments? (Y: Yes, N: No): y
---- Highlands Condimentes ----
[1] Steamed Milk:     $0.2
[2] Mocha:            $0.3
[3] Soy:              $0.25
[4] Whip:             $0.2

Add Condimentes (Press 'Enter' to done): 2
Added: Mocha

Add Condimentes (Press 'Enter' to done): 3
Added: Soy

Add Condimentes (Press 'Enter' to done): 2
Added: Mocha

Add Condimentes (Press 'Enter' to done):
```

Picture 11. Result when user add condiments

```
dotnet run
[1]    Steamed Milk:      $0.2
[2]    Mocha:            $0.3
[3]    Soy:              $0.25
[4]    Whip:            $0.2

Add Condimentes (Press 'Enter' to done): 2
Added:  Mocha

Add Condimentes (Press 'Enter' to done): 3
Added:  Soy

Add Condimentes (Press 'Enter' to done): 2
Added:  Mocha

Add Condimentes (Press 'Enter' to done):

---      Customer order      ---
--Beverage--      --Condiments--
Espresso      Mocha      Soy      Mocha
--Price: $2.75

Thank you for using our service!
```

Picture 12. Result when user finish added condiments

In this receipt there are 3 information such as Beverage, Condiments and Price.

```
---      Customer order      ---
--Beverage--      --Condiments--
Espresso      Mocha      Soy      Mocha
--Price: $2.75

Thank you for using our service!
```

Picture 13. Result of print reciept



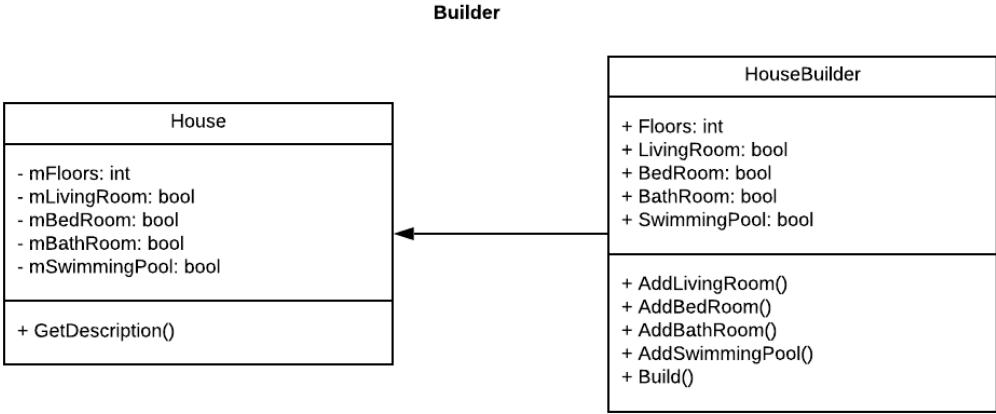
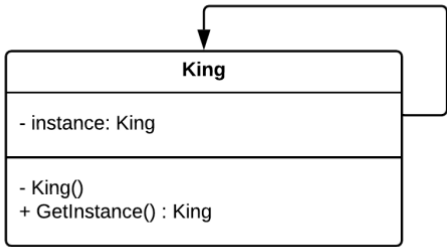
## 1.6. Compare Console application and Requirement

Application's requirements	Application implemented	Result
Serves Coffee to customer	User can take order specific coffee that customer's ordering.	By type a number that assigned for each kind of coffee so user can use the system to server coffee to customer.
The Coffee could add condiments.	After order coffee, user can add more condiment into order's coffee.	When user confirmed that coffee selected, by type a number that assigned for each kind of condiments so user can add more condiments into coffee.
Print receipt with name, condiment and price of coffee	The program prints out the receipt with name, condiment and price after confirm coffee and condiments.	After order coffee and add condiments, the system will auto print out receipt with all information about that coffee include: Name, condiments, price

Table 2. Compare console application and Requirement

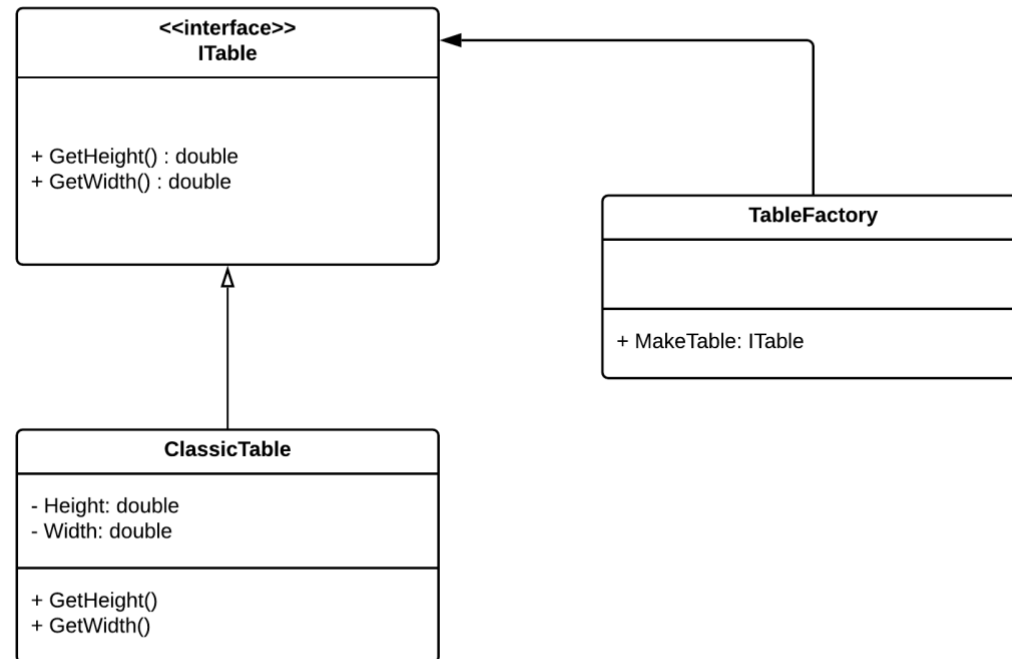
## 2. Scenario Investigation

### 2.1. A range of Design Patterns.

Type of Design Patterns	Description	Class Diagram
C r e a t i o n a l	<b>Builder</b> <ul style="list-style-type: none"> <li>Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.</li> </ul>	 <pre> classDiagram     class House {         - mFloors: int         - mLivingRoom: bool         - mBedRoom: bool         - mBathRoom: bool         - mSwimmingPool: bool         + GetDescription()     }     class HouseBuilder {         + Floors: int         + LivingRoom: bool         + BedRoom: bool         + BathRoom: bool         + SwimmingPool: bool         + AddLivingRoom()         + AddBedRoom()         + AddBathRoom()         + AddSwimmingPool()         + Build()     }     HouseBuilder --&gt; House         </pre> <p>The diagram shows a <b>House</b> class with private attributes <code>mFloors: int</code>, <code>mLivingRoom: bool</code>, <code>mBedRoom: bool</code>, <code>mBathRoom: bool</code>, and <code>mSwimmingPool: bool</code>, and a public method <code>+ GetDescription()</code>. A <b>HouseBuilder</b> class has corresponding public attributes for each of these and public methods <code>+ AddLivingRoom()</code>, <code>+ AddBedRoom()</code>, <code>+ AddBathRoom()</code>, <code>+ AddSwimmingPool()</code>, and <code>+ Build()</code>. An arrow points from <b>HouseBuilder</b> to <b>House</b>, indicating that the builder constructs the house object.</p>
	<b>Singleton</b> <ul style="list-style-type: none"> <li>The Singleton Pattern ensures you have at most one instance of a class in your application.</li> <li>The Singleton Pattern also provides a global access point to that instance.</li> <li>DotNet's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a</li> </ul>	 <pre> classDiagram     class King {         - instance: King         - King()         + GetInstance() : King     }     King --&gt; King         </pre> <p>The diagram shows a <b>King</b> class with a private static attribute <code>- instance: King</code>, a private constructor <code>- King()</code>, and a public static method <code>+ GetInstance() : King</code>. A self-referencing arrow on the <b>King</b> class indicates that the static method <code>GetInstance</code> returns a reference to the class itself, ensuring only one instance exists.</p>

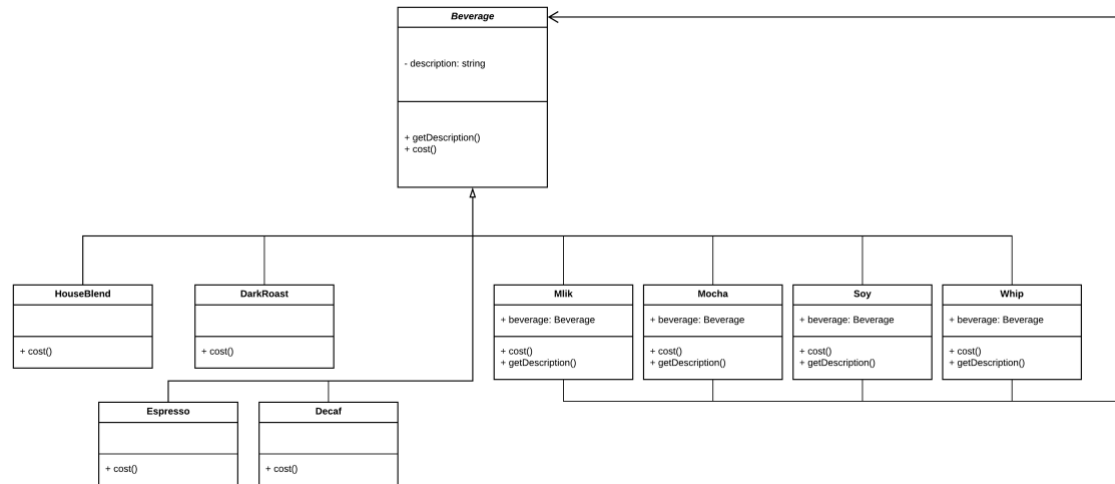
## Simple Factory

- Simple Factory, while not a bone fide design pattern is a simple way to decouple your clients from concrete classes.
- The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom.



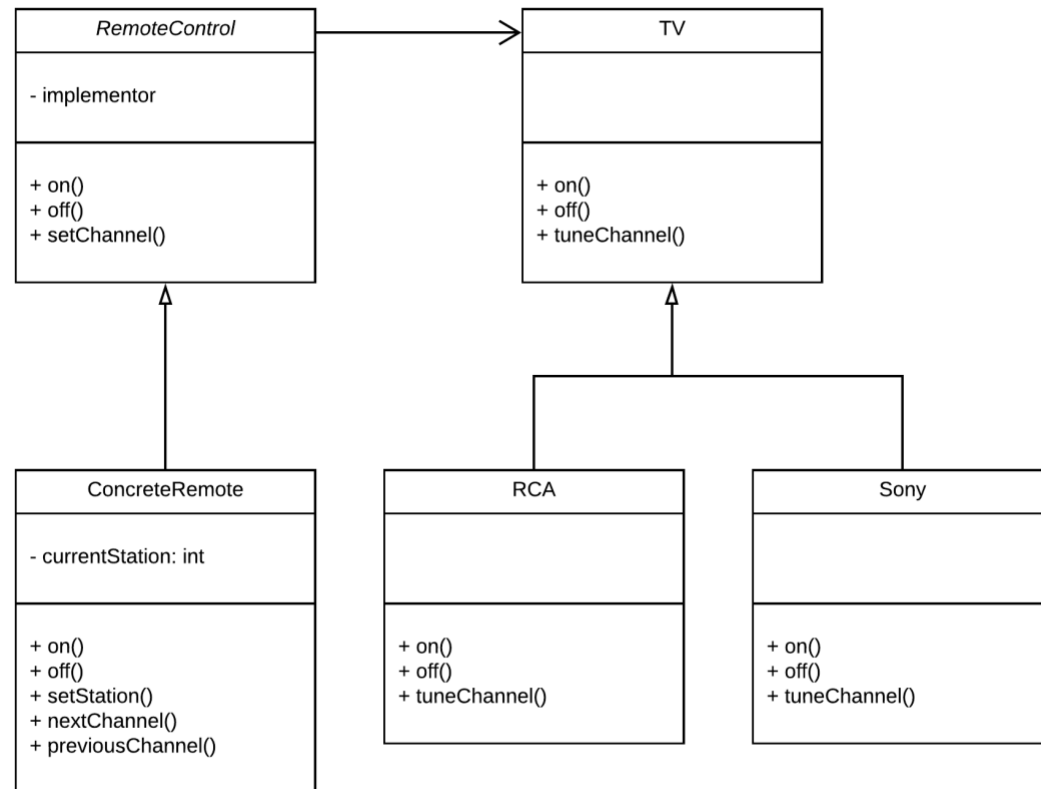
**Decorator**

- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- Decorators can result in many small objects in our design, and overuse can be complex.



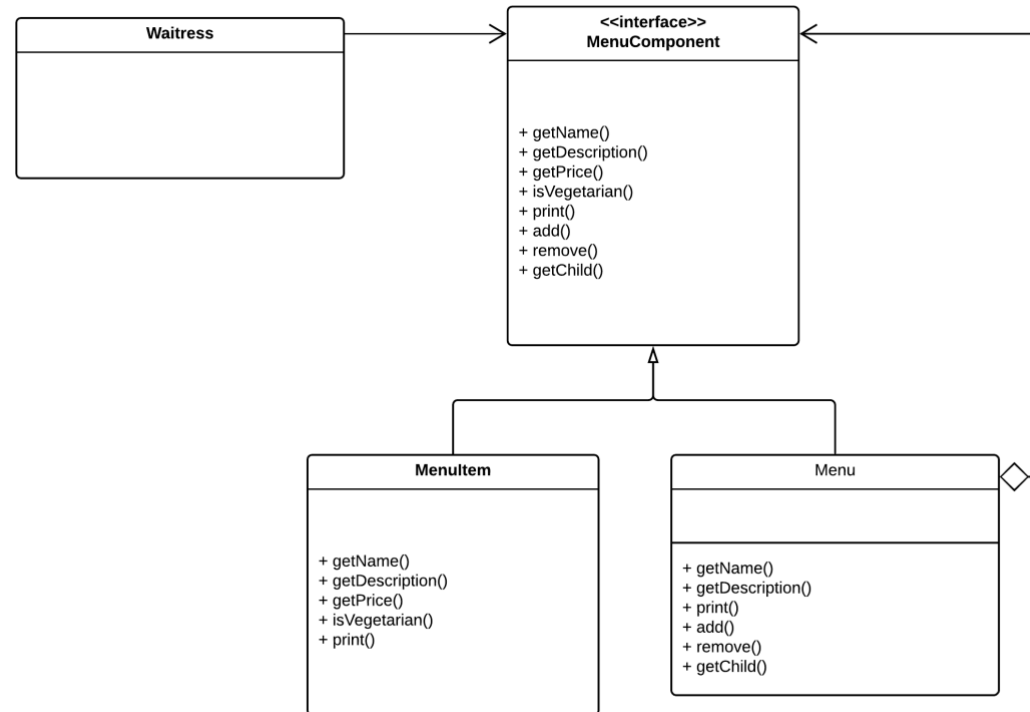
## Bridge

- The Bridge Pattern allows you to vary the implementation and the abstraction by placing the two in separate class hierarchies.



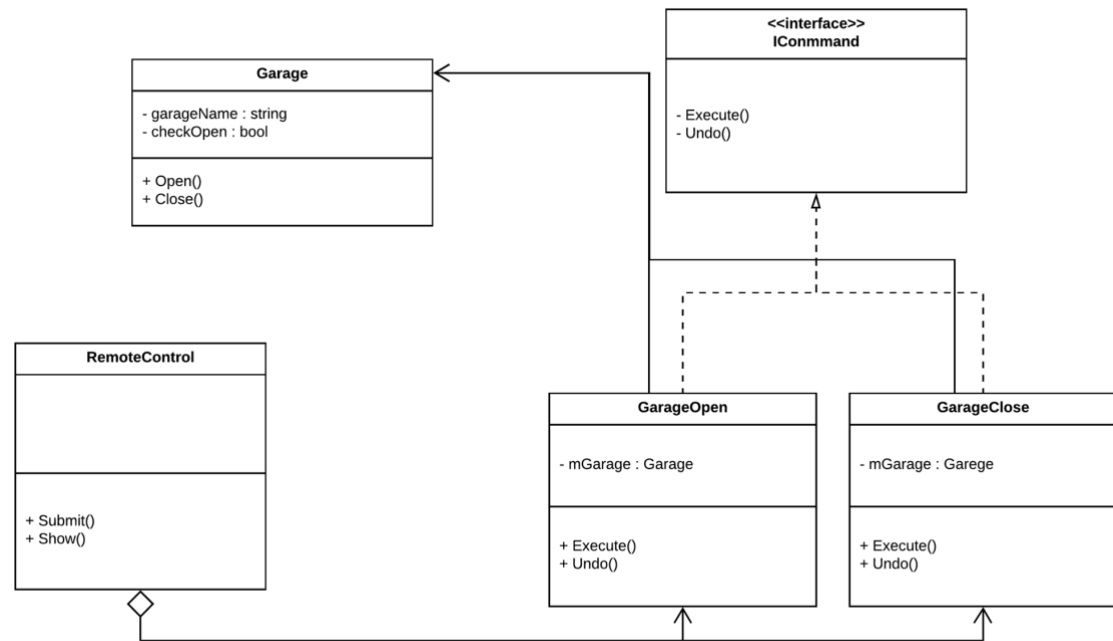
## Composite

- The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.
- Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



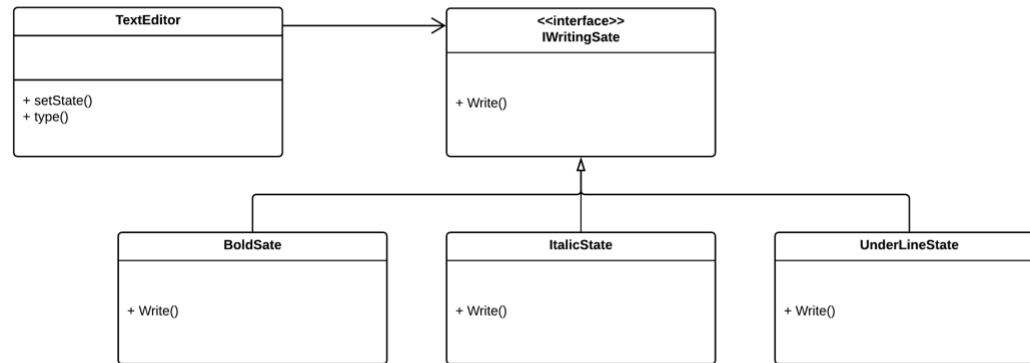
**Command**

- The command pattern decouples an object, making a request from the one that knows how to perform it.
- A command object is at the center of this decoupling and encapsulate a receiver with an action.
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.
- Commands may also be used to implement logging and transactional systems.



## State

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- The State Pattern represents state as a full-blown class.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in design.





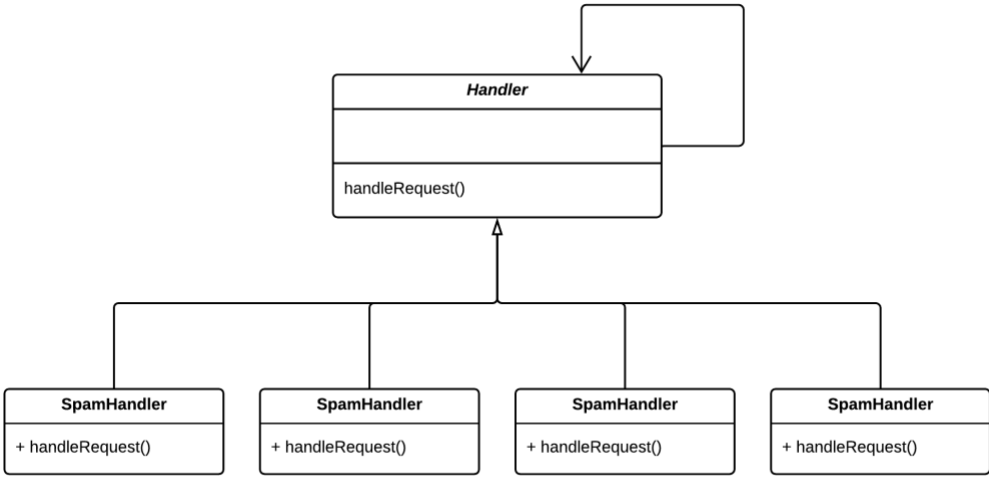
	<b>Chain of Responsibility</b>	<ul style="list-style-type: none"> <li>• Use the Chain of Responsibility Pattern when user want to give more than one object a chance to handle a request.</li> <li>• With the Chain of Responsibility Pattern, you create a chain of objects that examine a request. Each object in turn examines the request and handles it, or passes it on to the next object in the chain.</li> </ul>	 <pre> classDiagram     class Handler {         &lt;&lt;abstract&gt;&gt;         +handleRequest()     }     class SpamHandler {         +handleRequest()     }     Handler &lt; -- SpamHandler     SpamHandler --&gt; Handler     </pre> <p>The diagram illustrates the Chain of Responsibility pattern. It features an abstract <b>Handler</b> class with a <code>handleRequest()</code> method. Below it, there are four concrete <b>SpamHandler</b> classes, each also implementing the <code>handleRequest()</code> method. A solid line with an open arrowhead points from each <b>SpamHandler</b> class to the <b>Handler</b> class, indicating inheritance. Additionally, a curved arrow points from the <code>handleRequest()</code> method of the <b>Handler</b> class back to the <b>Handler</b> class itself, representing a self-reference for passing the request to the next handler in the chain.</p>
--	--------------------------------	--	--

Table 3. A range of Design Patterns. (viblo, n.d.)

## 2.2. The most appropriate design pattern from a range with a series of given scenarios.

In this case, I have used **Decorator Pattern** for Highlands Coffee Shop because these character (sjsu, n.d.):

- Decorators have the same supertype as the objects they decorate.
- I can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job. (The key point)
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.
- Decorators are also popular if you work with streams in frameworks such as .NET.

What **Decorator Pattern** can do answer these questions:

- What happens when the price of milk goes up? What do they do when they add a new caramel topping?
- 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 (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like **hasWhip()**.
- What if a customer wants a double mocha?

Consider a case of a Highland Coffee shop. In the Coffee shop, they will sell a few Coffee varieties and they will also provide condiments on the menu. Now imagine a situation wherein if the Coffee shop has to provide prices for each combination of Coffees and condiments. Even if there are four basic coffee and 6 different condiments, the application would go crazy maintaining all these concrete combinations of coffee and condiments.

In this case, the ordering system just wants to add condiments to specific coffee. This means the system didn't add any field and only change the value of each field that implemented before. Besides that, we still keep the base Coffee then make it turn to another Coffee with multi condiments.

If we use Builder Patterns, so the system is going to create new field and value for each item, so if in some cases, the user can't add more condiments for coffee without deleting the old coffee.

If we use Factory Patterns, there are a lot of kinds of coffees, so when implemented Factory Pattern that makes the program look like heavy and hard to maintain.

And this table below is the result after comparison some design patterns (tutorialspoint, n.d.):

Decorator	Adapter
<p><b>Decorator</b> is also called "Smart Proxy." This is used when you want to add functionality to an object, but not by extending that object's type. This allows you to do so at runtime. For example: Adding mocha in a coffee.</p>	<p>Adapts interface of an existing class to another interface. For example: Electrical adapter.</p>
	Proxy
	<p>Proxy could be used when you want to lazy-instantiate an object, or hide the fact that you're calling a remote service, or control access to the object.</p>
	Bridge
	<p>Bridge is very similar to Adapter, but we call it Bridge when you define both the abstract interface and the underlying implementation. I.e. you're not adapting to some legacy or third-party code, you're the designer of all the code but you need to be able to swap out different implementations.</p>
	Facade
	<p>Facade is a higher-level (read: simpler) interface to a subsystem of one or more classes. Suppose you have a complex concept that requires multiple objects to represent. Making changes to that set of objects is confusing, because I don't always know which object has the method I need to call. That's the time to write a Facade that provides high-level methods for all the complex operations you can do to the collection of objects</p>
	Strategy
	<p>The strategy pattern allows you to change the implementation of something used at runtime.</p>

Table 4. Compare Decorator with another Pattern

### 3.1. The use of Design Pattern for the given purpose specified

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.

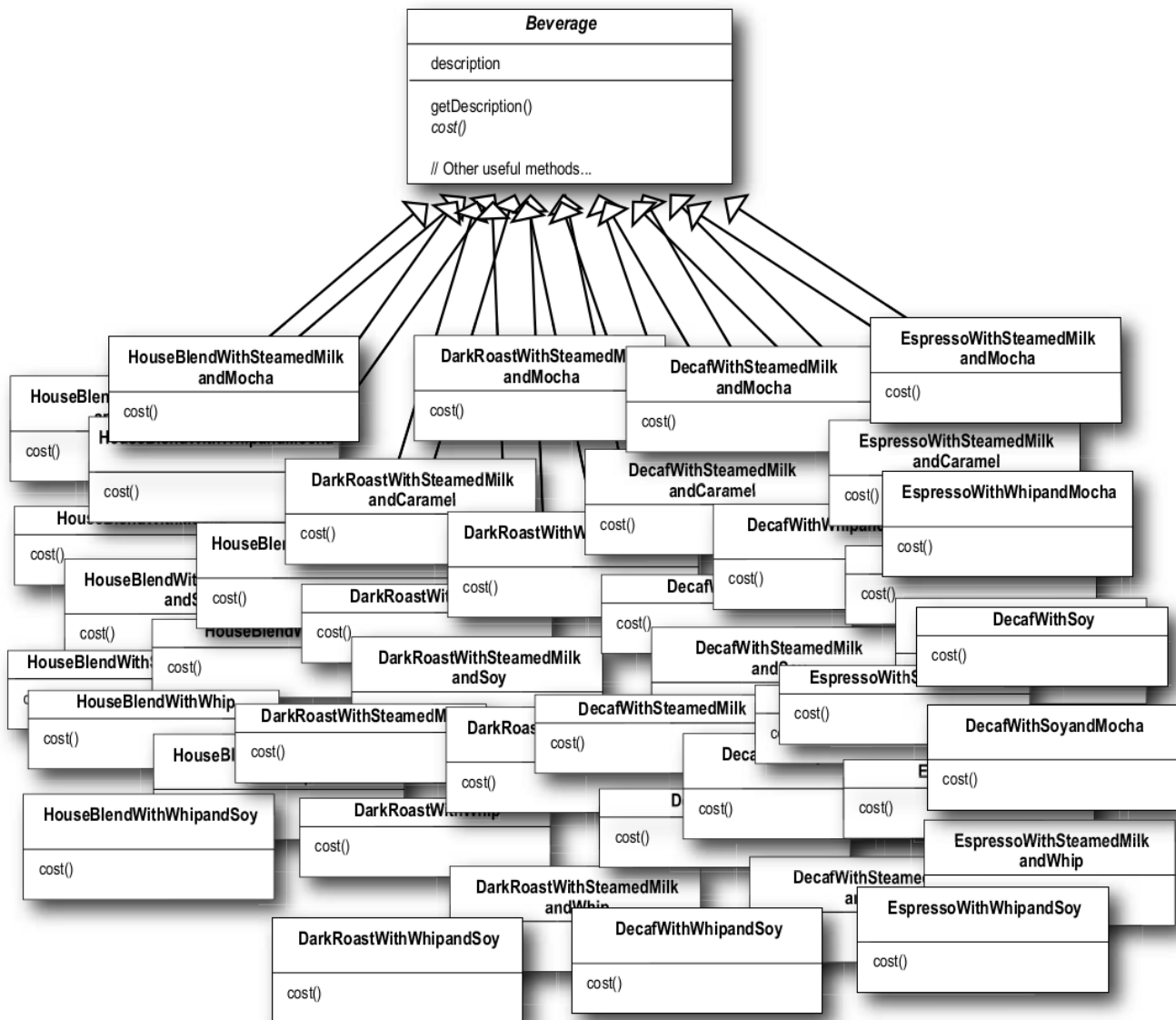


Figure 2. Class diagram before using design pattern

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?

### 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! Now I will start with a **beverage** and **decorate** it with the **condiments** at runtime.

**The Decorator Pattern** attaches additional responsibilities to an object dynamically.

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

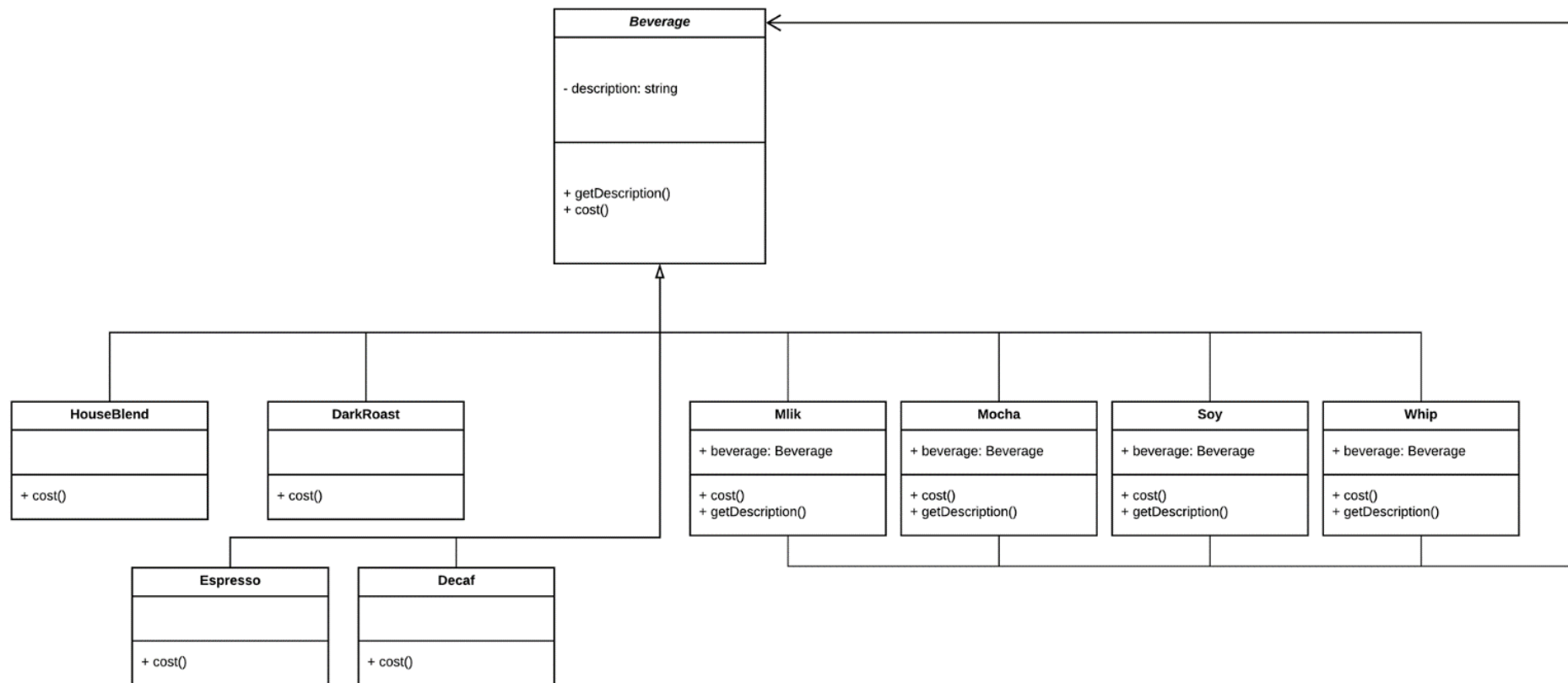
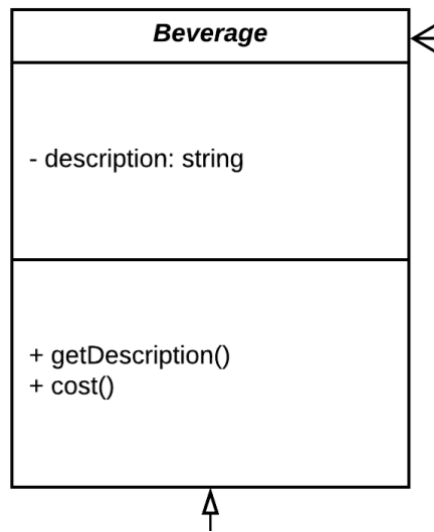


Figure 3. Class Diagram after using Design Pattern

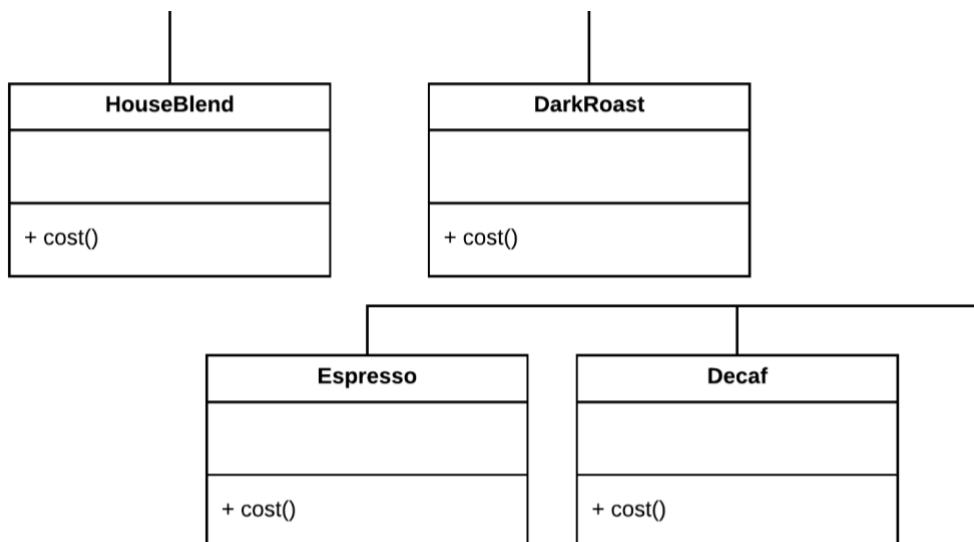
**Explanation:**

Beverage acts as our abstract component:



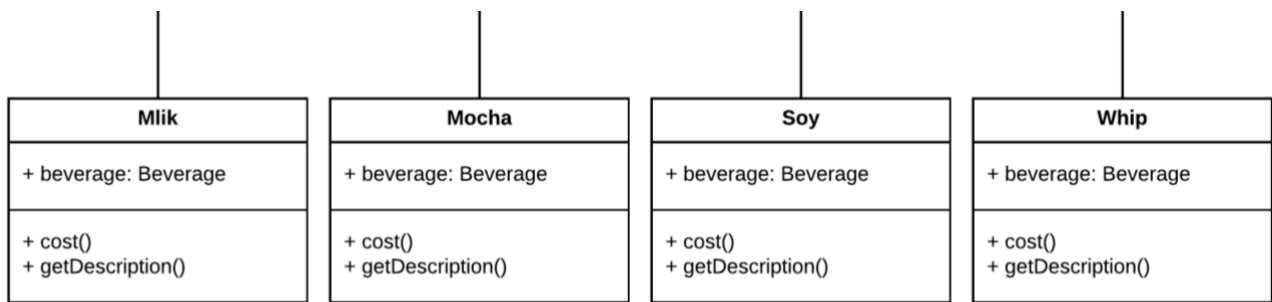
*Figure 4. Beverage class*

The four concrete components, one per coffee type.



*Figure 5. Four concrete components*

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



- 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 sub-classing the abstract class Beverage 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**.

### Summary:

Before using Decorator Pattern	After using Decorator Pattern
There are too much class to make coffee and coffee with condiment.	The system had structure for Coffee and Condiment for each Coffee and Condiment there are specific class.
It hard to maintain, update or upgrade system.	Easy to maintain, update or upgrade system.
It hard to use, requirement user remembers all the type of coffee and condiments.	Easy to choosing coffee and add condiment. The System doesn't requirement users remember all the type of coffee and condiments.

*Table 5. Summary compare before and after using Decorator Pattern*

Base on the book: “Design Patterns: Elements of Reusable Object-Oriented Programming” (uml, n.d.), using Decorator is a good choice to add responsibilities to individual objects dynamically and transparently, that is without affecting other objects and for responsibilities that can be withdrawn. When extension by subclass is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of sub-classes to support every combination. Or a class definition may be hidden or otherwise unavailable for sub-classing.

I used **Decorator** for this problem because these benefits (codeproject, n.d.):

- **More flexibility than static inheritance:** The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritances. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities. Decorators also make it easy to add a property twice.  
**For example:** to give a Coffee a double Mocha, simply attach two MochaDecorators. Inheriting from a Condiments class twice is error-prone at best.
- **Avoids feature-laden classes high up in the hierarchy:** Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, I can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed of simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.
- **A decorator and its component aren't identical:** A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself.
- **Lots of little objects:** A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.



### 3.1. Evaluate a range of design patterns against the range of given scenarios with justification of your choices.

To justify my selected design pattern, I will compare the prototype design pattern between **Decorator Pattern** and **Builder Pattern**, **Factory Pattern** – What can be used to create a program for Coffee Shop.

Builder Pattern	Decorator Pattern	Factory Pattern
Separate the construction of a complex object from its representation so that the same construction process can create different representations.	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.	Is simply a wrapper function around a constructor (possibly one in a different class). The key difference is that a factory method pattern requires the entire object to be built in a single method call, with the entire parameters pass in on a single line. The final object will be returned.

Table 6. Compare some Pattern can implement to ordering beverage

## Implement with Builder Pattern:

- Class diagram:

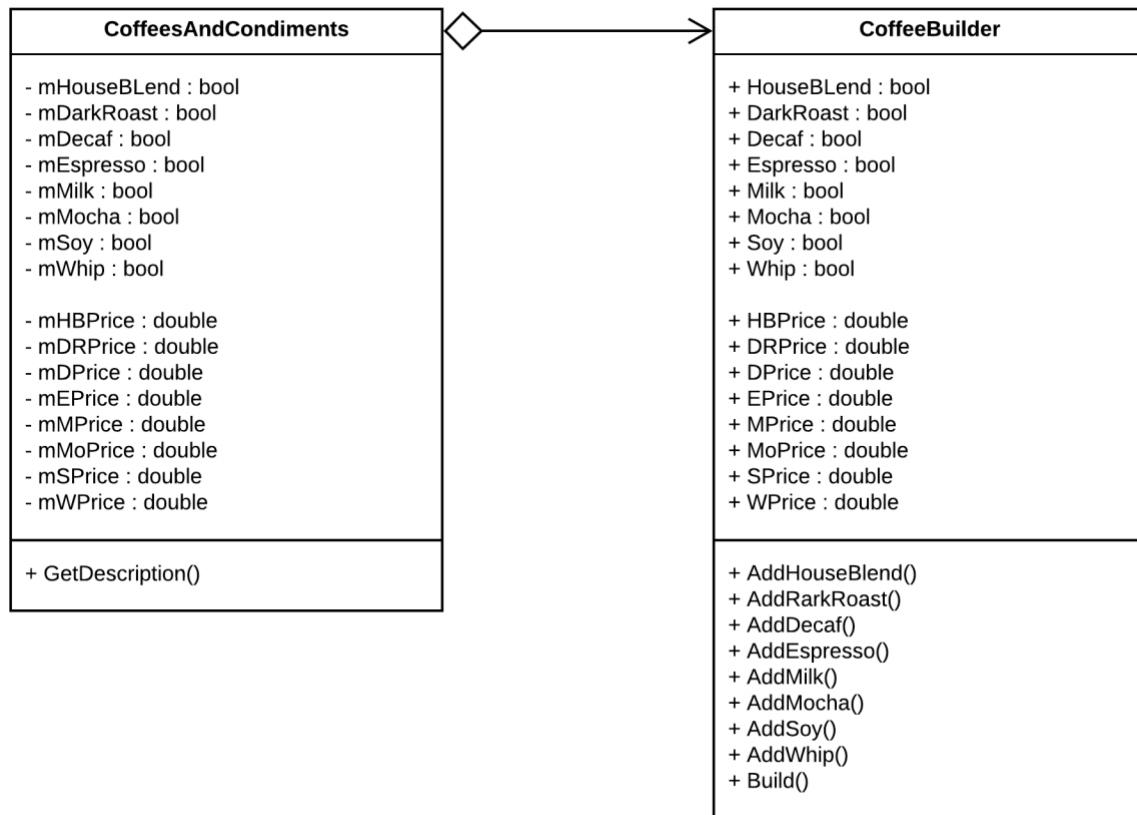


Figure 6. Class diagram of using Builder Pattern for HighLands Coffee

## Code snippet:

```
class CoffeesAndCondiments
{
    private bool mHouseBlend;
    private double mHBPrice;
    private bool mDarkRoast;
    private double mDRPrice;
    private bool mDecaf;
    private double mDPrice;
    private bool mEspresso;
    private double mEPrice;
    private bool mMilk;
    private double mMPPrice;
    private bool mMocha;
    private double mMoPrice;
    private bool mSoy;
    private double mSPPrice;
    private bool mWhip;
    private double mWPrice;

    public CoffeesAndCondiments(CoffeeBuilder builder)
    {
        this.mHouseBlend = builder.HouseBlend;
        this.mHBPrice = builder.HouseBlendPrice;
        this.mDarkRoast = builder.DarkRoast;
        this.mDRPrice = builder.DarkRoastPrice;
        this.mDecaf = builder.Decaf;
        this.mDPrice = builder.DecafPrice;
        this.mEspresso = builder.Espresso;
        this.mEPrice = builder.EspressoPrice;
        this.mMilk = builder.Milk;
        this.mMPPrice = builder.MilkPrice;
        this.mMocha = builder.Mocha;
        this.mMoPrice = builder.MochaPrice;
        this.mSoy = builder.Soy;
        this.mSPPrice = builder.SoyPrice;
        this.mWhip = builder.Whip;
        this.mWPrice = builder.WhipPrice;
    }

    public string GetDescription()
    {
        var sb = new StringBuilder();
        double totalPrice = this.mHBPrice + this.mDRPrice + this.mDPrice +
this.mEPrice + this.mMPPrice + this.mMoPrice + this.mSPPrice + this.mWPrice;
        sb.Append($"The Coffee House Blend: {this.mHouseBlend}\n");
        sb.Append($"The Coffee Dark Roast: {this.mDarkRoast}\n");
        sb.Append($"The Coffee Decaf: {this.mDecaf}\n");
        sb.Append($"The Coffee Espresso: {this.mEspresso}\n");
        sb.Append("-----\n");
        sb.Append($"Milk: \t{this.mMilk}\n");
        sb.Append($"Mocha: \t{this.mMocha}\n");
        sb.Append($"Soy: \t{this.mSoy}\n");
        sb.Append($"Whip: \t{this.mWhip}\n");
        sb.Append($"Total Price: ${totalPrice}");
        return sb.ToString();
    }
}
```

```

class CoffeeBuilder
{
    public bool HouseBlend;
    public double HouseBlendPrice;
    public bool DarkRoast;
    public double DarkRoastPrice;
    public bool Decaf;
    public double DecafPrice;
    public bool Espresso;
    public double EspressoPrice;
    public bool Milk;
    public double MilkPrice;
    public bool Mocha;
    public double MochaPrice;
    public bool Soy;
    public double SoyPrice;
    public bool Whip;
    public double WhipPrice;
    public CoffeeBuilder()
    {
    }
    public CoffeeBuilder AddHouseBlend()
    {
        this.HouseBlend = true;
        this.HouseBlendPrice = 1.2;
        return this;
    }
    public CoffeeBuilder AddDarkRoast()
    {
        this.DarkRoast = true;
        this.DarkRoastPrice = 1.4;
        return this;
    }
    public CoffeeBuilder AddDecaf()
    {
        this.Decaf = true;
        this.DecafPrice = 1.7;
        return this;
    }
    public CoffeeBuilder AddEspresso()
    {
        this.Espresso = true;
        this.EspressoPrice = 1.9;
        return this;
    }
    public CoffeeBuilder AddMocha()
    {
        this.Mocha = true;
        this.MochaPrice = .30;
        return this;
    }
    public CoffeeBuilder AddSoy()
    {
        this.Soy = true;
        this.SoyPrice = .25;
        return this;
    }
    public CoffeeBuilder AddWhip()
    {
        this.Whip = true;
        this.WhipPrice = .20;
        return this;
    }
    public CoffeesAndCondiments Build()
    {
        return new CoffeesAndCondiments(this);
    }
}

```

```

static void Main(string[] args)
{
    var coffee1 = new CoffeeBuilder().AddEspresso()
                                      .AddMilk()
                                      .AddSoy()
                                      .Build();

    var coffee2 = new CoffeeBuilder().AddHouseBlend()
                                      .AddMocha()
                                      .AddMocha()
                                      .Build();

    var coffee3 = new CoffeeBuilder().AddHouseBlend()
                                      .AddMocha()
                                      .AddMocha()
                                      .Build();

    Console.WriteLine(coffee1.GetDescription());
    Console.WriteLine();

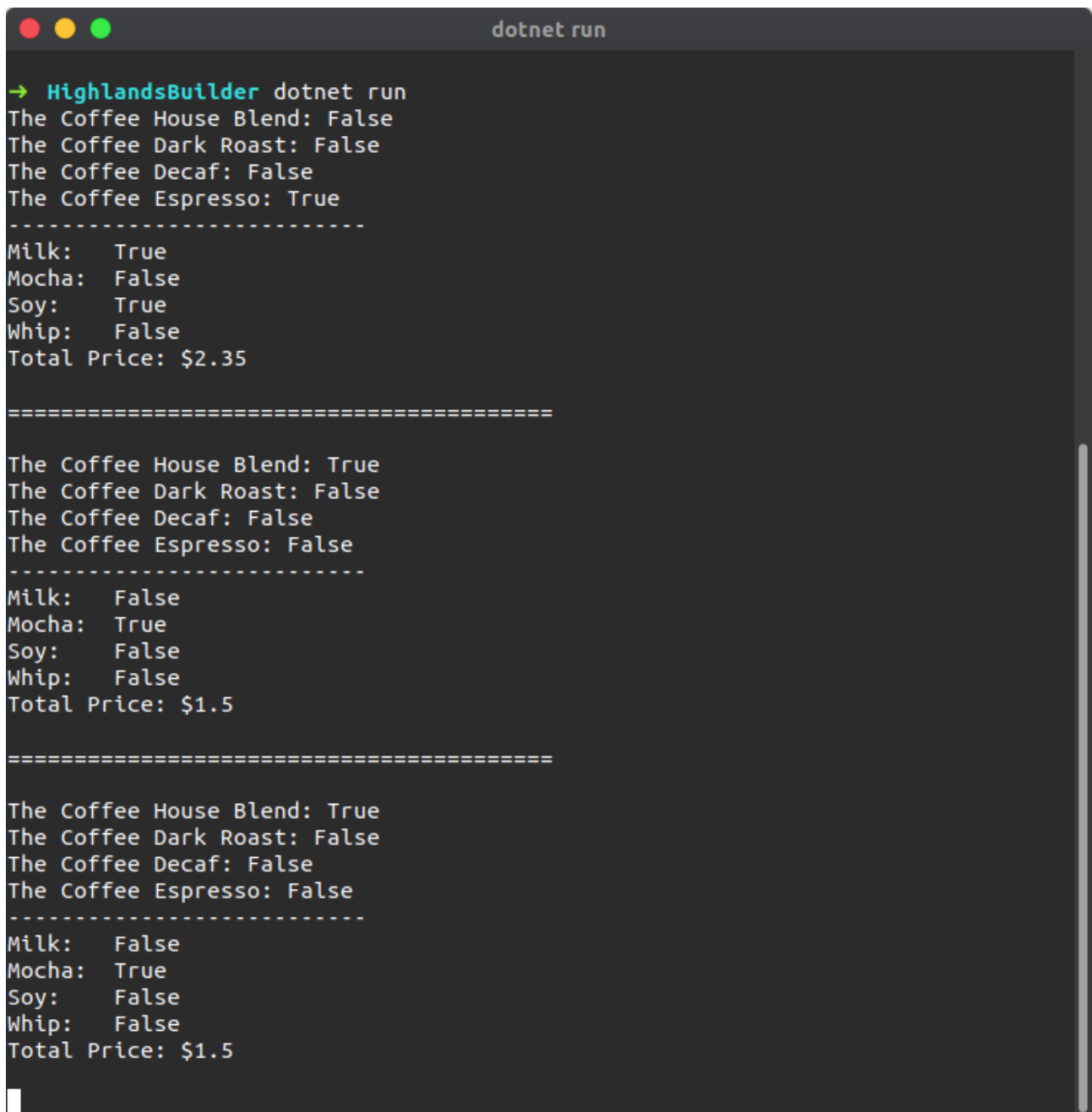
    Console.WriteLine("=====");
    Console.WriteLine();
    Console.WriteLine(coffee2.GetDescription());
    Console.WriteLine();

    Console.WriteLine("=====");
    Console.WriteLine();
    Console.WriteLine(coffee3.GetDescription());

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

```

- **Result:**



```
dotnet run
→ HighlandsBuilder dotnet run
The Coffee House Blend: False
The Coffee Dark Roast: False
The Coffee Decaf: False
The Coffee Espresso: True
-----
Milk:    True
Mocha:   False
Soy:     True
Whip:    False
Total Price: $2.35

=====

The Coffee House Blend: True
The Coffee Dark Roast: False
The Coffee Decaf: False
The Coffee Espresso: False
-----
Milk:    False
Mocha:   True
Soy:     False
Whip:    False
Total Price: $1.5

=====

The Coffee House Blend: True
The Coffee Dark Roast: False
The Coffee Decaf: False
The Coffee Espresso: False
-----
Milk:    False
Mocha:   True
Soy:     False
Whip:    False
Total Price: $1.5
```

*Picture 14. Result after using Builder Pattern for ordering beverage*

**The result 1:** I used CoffeeBuilder() to make offer for Espresso and Milk, Soy Condiments.

**The result 2:** I used CoffeeBuilder() to make offer for House Blend with Mocha condiment.

**The result 3:** I used CoffeeBuilder() to make offer for House Blend with double Mocha condiment.

- **Conclusion of using Builder pattern:**

When using the **Builder Pattern** to create coffee, I also can order a specific coffee with specific condiments. But there is one thing that Builder Pattern can't do is if a customer wants to have double Mocha then the Builder Pattern just create a new coffee with just 1 Mocha.

In the code that I implemented before, I have tried to build double mocha, but the result of price and beverage description just show only 1 Mocha and Price for coffee with 1 Mocha.

This Pattern will be a good pattern for the Coffee shop if the requirements from the Coffee Shop didn't ask for order more condiments for a coffee.

When implementing in C#, there is too much field to do so that will cause some problem for developers if the new developer joins in and work with this system.

After using **Builder Pattern** for ordering coffee at Highlands, I feel that not working well and which makes the system look heavy and hard to maintain, upgrade and update any beverage or condiment.

## Implement with Simple Factory Pattern:

- Class Diagram:

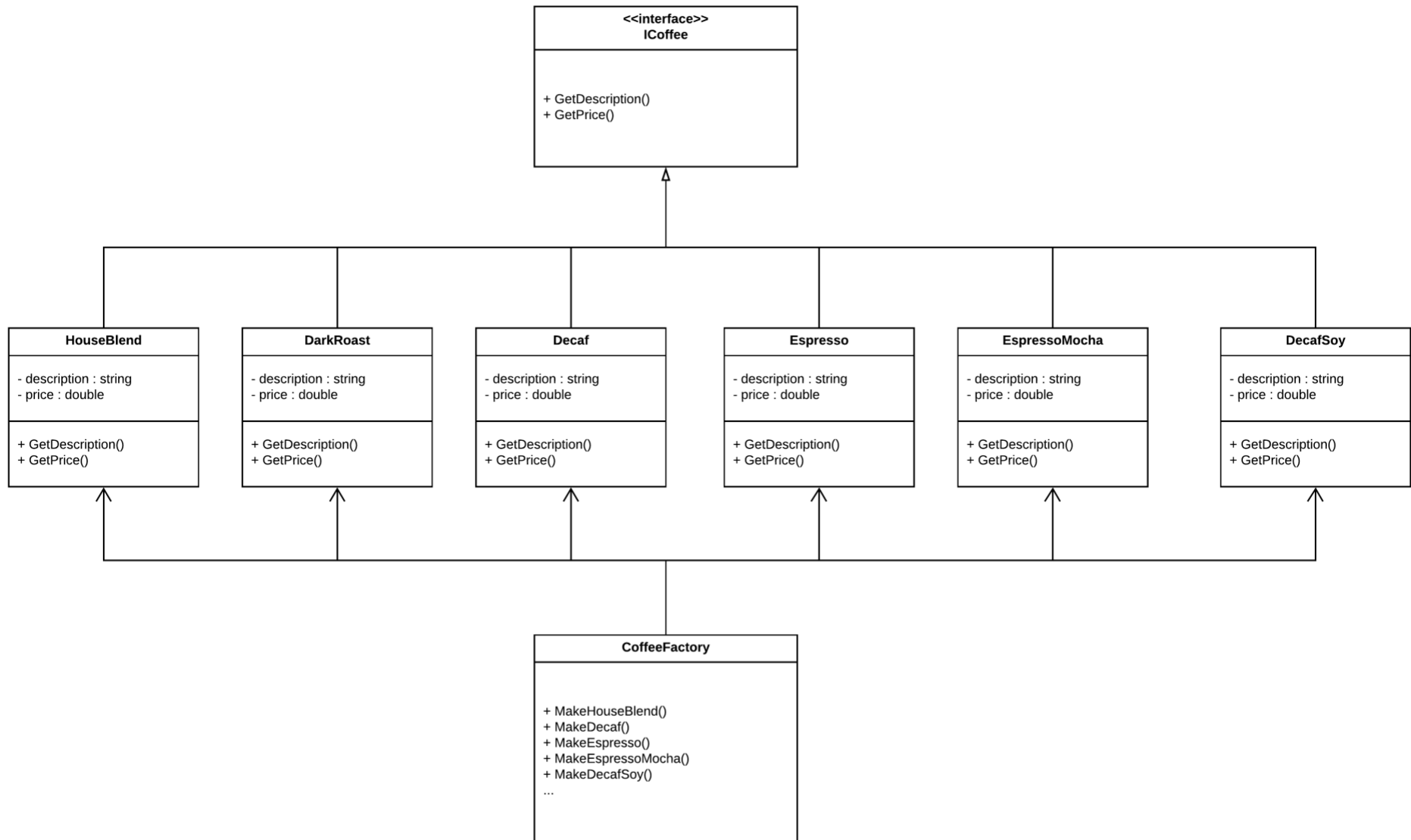


Figure 7. Class diagram of using Simple Factory Pattern for HighLands Coffee



- **Code Snippet:**

```
public interface ICoffee
{
    string GetDescription();
    double GetPrice();
}
```

```
public class HouseBlend : ICoffee
{
    private string description;
    private double price;
    public HouseBlend()
    {
        this.description = "House Blend";
        this.price = 1.2;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}
```

```
public class DarkRoast : ICoffee
{
    private string description;
    private double price;
    public DarkRoast()
    {
        this.description = "Dark Roast";
        this.price = 1.4;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}
```

```

public class Decaf : ICoffee
{
    private string description;
    private double price;
    public Decaf()
    {
        this.description = "Decaf";
        this.price = 1.7;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}

```

```

public class Espresso : ICoffee
{
    private string description;
    private double price;
    public Espresso()
    {
        this.description = "Espresso";
        this.price = 1.9;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}

public class EspressoMocha : ICoffee
{
    private string description;
    private double price;
    public EspressoMocha()
    {
        this.description = "Espresso + Mocha";
        this.price = 2.2;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}

```

```

public class DecafSoy : ICoffee
{
    private string description;
    private double price;
    public DecafSoy()
    {
        this.description = "Decaf + Soy";
        this.price = 1.95;
    }
    public string GetDescription()
    {
        return this.description;
    }

    public double GetPrice()
    {
        return this.price;
    }
}

public static class CoffeeFactory
{
    public static ICoffee MakeHouseBlend()
    {
        return new HouseBlend();
    }

    public static ICoffee MakeEspressoMocha()
    {
        return new EspressoMocha();
    }

    public static ICoffee MakeDecafSoy()
    {
        return new DecafSoy();
    }
}

```

```

static void Main(string[] args)
{
    var coffee1 = CoffeeFactory.MakeHouseBlend();
    Console.WriteLine($"Coffee: {coffee1.GetDescription()}, Price: {coffee1.GetPrice()}");

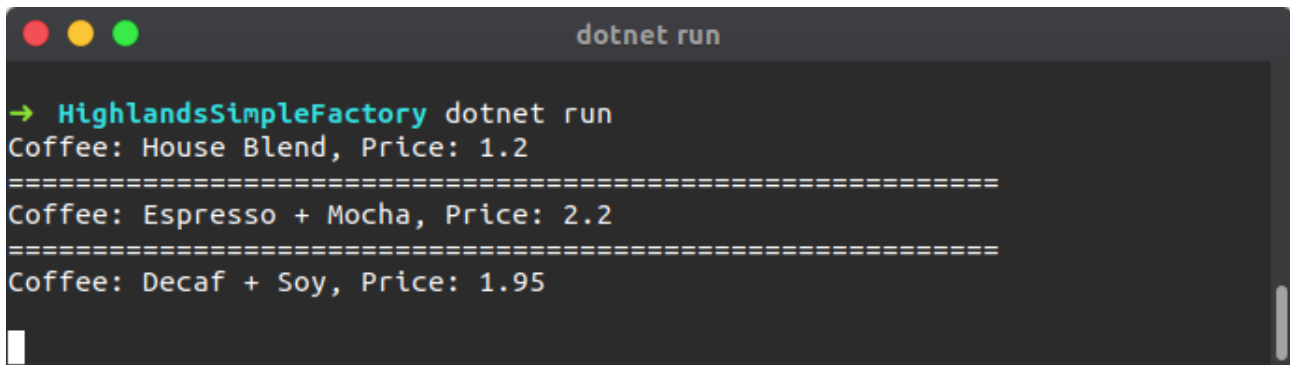
    Console.WriteLine("=====");
    var coffee2 = CoffeeFactory.MakeEspressoMocha();
    Console.WriteLine($"Coffee: {coffee2.GetDescription()}, Price: {coffee2.GetPrice()}");

    Console.WriteLine("=====");
    var coffee3 = CoffeeFactory.MakeDecafSoy();
    Console.WriteLine($"Coffee: {coffee3.GetDescription()}, Price: {coffee3.GetPrice()}");

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

```

- **Result:**



```
dotnet run
→ HighlandsSimpleFactory dotnet run
Coffee: House Blend, Price: 1.2
=====
Coffee: Espresso + Mocha, Price: 2.2
=====
Coffee: Decaf + Soy, Price: 1.95
```

*Picture 15. Result after using Simple Factory Pattern for ordering beverage*

**The result 1:** I used MakeHouseBlend() in CoffeeFactory to make an offer for House Blend and didn't add any condiments.

**The result 2:** I used MakeEspressoMocha() in CoffeeFactory to make an offer for Espresso with Mocha condiment.

**The result 3:** I used MakeDecafSoy() in CoffeeFactory to make an offer for Decaf with Soy condiment.

- **Conclusion of using Simple Factory Pattern:**

When using the **Simple Factory Pattern** to create an order for the coffee shop, I also can create a specific coffee with specific condiments. But with this Pattern, I also meet some problems with the builder pattern that when customer want to have double condiments and the system can't do that without creating new Factory such as `MakeEspressoMochaMocha()`.

In the code that I implemented, the result of the system shows the receipt match with the requirement that shows information about coffee, condiment, and price.

But when customers have any asking to add more condiments or change coffee after ordered, it will make some problems with using simple Factory.

When implementing in C#, there are too much class have to implement so that will cause some problem to add new coffee or condiments into the system.

After using the **Simple Factory Pattern** for ordering coffee at Highlands, I feel that not working well and which makes the system look like heavy and hard to maintain, upgrade and update any beverage or condiment.

To make sure that using **Decorator** is correct I will compare with some related Patterns: (wikipedia, n.d.)

Adapter Pattern	Composite Pattern	Strategy Pattern
A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.	A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities. It isn't intended for object aggregation.	A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

*Table 7. Compare Decorator Pattern with related Patterns*

So in this scenario, the Decorator Pattern is good to implement into the system, it will make the system easy to use and code more cleanly and easy to upgrade, update or maintain. All of the things that the Decorator Pattern provides match all of the scenario's requirements.

## CONCLUSION

When I first encountered the buzz surrounding Design Patterns, I was rather put off by it. The more I heard about it, the more it seemed to be just taking stuff that was obvious to me as an experienced programmer and codifying it with a complex series of definitions, sometimes with debate-inspiring subtleties.

The other advance of the design patterns text is to give you a framework to think about the programming idioms you've discovered that don't yet have a universal name. When you feel that you're writing or reading a piece of code for the nth time, ask yourself:

- What's the general pattern of the code?
- What's the intent of the pattern?
- What's the abstract structure of the classes and methods involved in it (the pattern common to every time you've written it)?
- What are the consequences of doing it that way?
- How does it relate to other design patterns you've discovered or used?

After this report, I also have a lot of knowledge about OOP, specially is about using Design Pattern in a real problem to build a program. Each Design Pattern have special job and solve a difference problem. When a developer has experience in Design Pattern, a developer will know what, when and how to use Design Pattern to complete the program. After using Design Pattern, the program will be easy to use, update, upgrade and maintain.

The most important points need to note and improve is when initiating program, the selection Design Pattern and design UML.

## References

- codeproject. (n.d.). Retrieved from codeproject:  
<https://www.codeproject.com/Tips/468951/Decorator-Design-Pattern-in-Java>
- sjsu. (n.d.). Retrieved from sjsu: <http://www.cs.sjsu.edu/~pearce/modules/cases/uw/decorator.htm>
- tutorialspoint. (n.d.). Retrieved from tutorialspoint:  
[https://www.tutorialspoint.com/design\\_pattern/decorator\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm)
- uml. (n.d.). Retrieved from uml: <http://www.uml.org.cn/c++/pdf/designpatterns.pdf>
- viblo. (n.d.). Retrieved from viblo: <https://viblo.asia/p/tong-hop-cac-bai-huong-dan-ve-design-pattern-23-mau-co-ban-cua-gof-3P0IPQPG5ox>
- wikipedia. (n.d.). Retrieved from wikipedia: [https://en.wikipedia.org/wiki/Composite\\_pattern](https://en.wikipedia.org/wiki/Composite_pattern)