

Higher Nationals in Computing

UNIT 20: ADVANCED PROGRAMMING ASSIGNMENT 2

Learner's name: Trinh Thi Dieu Huyen

ID: GDD18606

Class: GCS0801B.1

Subject code: 1651

Assessor name: **PHAN MINH TAM**

Assignment due:

Assignment submitted:

ASSIGNMENT 2 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 16: Cloud computing		
Submission date		Date Received 1st submission	
Re-submission Date		Date Received 2nd submission	
Student Name	Trinh Thi Dieu Huyen	Student ID	GDD18606
Class	1651 GCS0801B.1	Assessor name	Phan Minh Tam
Student declaration <p>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.</p>			
		Student's signature	<i>huyen</i>

Grading grid

P3	P4	P3	P4	D3	D4

☐ **Summative Feedback:**

☐ **Resubmission Feedback:**

Grade:

Assessor Signature:

Date:

Signature & Date:

ASSIGNMENT 2 BRIEF

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 2: Advanced Programming		
Assignment title	Application development with class diagram and design patterns		
Academic Year			
Unit Tutor			
Issue date		Submission date	

Submission Format:

Format: The submission is in the form of an individual written report. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system.

Submission: Students are compulsory to submit the assignment in due date and in a way requested by the Tutors. The form of submission will be a **soft copy in PDF** posted on corresponding course of <http://cms.greenwich.edu.vn/> together with zipped project files.

Note: The Assignment *must* be your own work, and not copied by or from another student or from books etc. If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. Make sure that you know how to reference properly, and that understand the guidelines on plagiarism. *If you do not, you definitely get fail*

Assignment Brief and Guidance:

Scenario: (continued from Assignment 1) Your team has shown the efficient of UML diagrams in OOAD and introduction of some Design Patterns in usages. The next tasks are giving a demonstration of using OOAD and DP in a small problem, as well as advanced discussion of range of design patterns.

Tasks: Your team is now separated and perform similar tasks in parallel. You will choose one of the real scenarios that your team introduced about DP in previous phase, then implement that scenario based on the corresponding class diagram your team created. You may need to amend the diagram if it is needed for your implementation. In additional, you should discuss a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice.

In the end, you need to write a report with the following content:

- A final version of the class diagram based on chosen scenario which has potential of using DP.
- Result of a small program implemented based on the class diagram, explain how you translate from design diagram to code.
- Discussion of a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice (why your DP is the most appropriate in that case).

The working application must also be demonstrated.

Learning Outcomes and Assessment Criteria		
Pass	Merit	Distinction
LO3 Implement code applying design patterns		
P3 Build an application derived from UML class diagrams.	M3 Develop code that implements a design pattern for a given purpose.	D3 Evaluate the use of design patterns for the given purpose specified in M3.
LO4 Investigate scenarios with respect to design patterns		
P4 Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types.	M4 Reconcile the most appropriate design pattern from a range with a series of given scenarios.	D4 Critically evaluate a range of design patterns against the range of given scenarios with justification of your choices.

Table of Contents

ASSIGNMENT 2 FRONT SHEET	1
ASSIGNMENT 2 BRIEF	3
Table of Figures.....	5
Table of Tables.....	6
ASSIGNMENT 2 ANSWERS	1
LO3 Implement code applying design patterns	1
P3 Build an application derived from UML class diagrams.....	1
1. Source Code.....	1
2. Run/Test Program	12
M3 Develop code that implements a design pattern for a given purpose	18
LO4 Investigate scenarios with respect to design patterns.....	20
P4 Discuss a range of design patterns with relevant examples of creational, structural, and behavioral pattern types.....	20
1. Creational pattern	20
1.1. Builder pattern	20
1.2. Singleton pattern.....	22
2. Structural pattern.....	23
2.1. Adapter pattern.....	24
2.2. Decorator pattern	25
3. Behavioral pattern	27
3.1. Observer pattern.....	27
3.2. Iterator pattern	29
M4 Reconcile the most appropriate design pattern from a range with a series of given scenarios	30
REFERENCES	31

Table of Figures

Figure 1: Original options table	12
Figure 2: Result of option 1 - Find Food follow ID.....	12
Figure 3: Result of option 2 - Find Drink follow ID	12
Figure 4: Result of option 3 - Find Staff follow ID	13

Figure 5: Result of option 4 - Create and save food.....	13
Figure 6: Result of option 5 - Create and save drink.....	14
Figure 7: Result of option 6 - Create and save staff.....	14
Figure 8: Result of option 7 - Create and save bill	15
Figure 9: Result of option 8 - Find bill	15
Figure 10: Result of option 9 - Delete bill	16
Figure 11: Result of option 9 - Check deleted bill – Find bill.....	16
Figure 12: Result of option 10 - Update information for food by food ID	17
Figure 13: Result of option 11 - Update information for drink by drink ID	17
Figure 14: Result of option 12 - Update information for staff by staff ID	18
Figure 15: Structure of builder pattern.....	20
Figure 16: Structure of Singleton pattern	23
Figure 17: Structure of adapter pattern.....	24
Figure 18: Structure of Decorator pattern	26
Figure 19: Structure of observer pattern	27
Figure 20: Structure of Iterator pattern.....	29

Table of Tables

No table of figures entries found.

ASSIGNMENT 2 ANSWERS

LO3 Implement code applying design patterns

P3 Build an application derived from UML class diagrams

1. Source Code

```
using System;
using System.Collections.Generic;

namespace manageBillRestaurant
{
    class Program
    {
        static void Main(string[] args)
        {

            ObjectFactory objectFactory = new ObjectFactory();
            Dictionary<string, object> data = new Dictionary<string, object>();

            int id;
            IObject obj;

            while(true)
            {
                Console.Clear();
                Console.WriteLine("1. Find Food");
                Console.WriteLine("2. Find Drink");
                Console.WriteLine("3. Find Staff Info");
                Console.WriteLine("4. Create and save Food");
                Console.WriteLine("5. Create and save Drink");
                Console.WriteLine("6. Create and save Staff Info");
                Console.WriteLine("7. Create and save Bill");
                Console.WriteLine("8. Find Bill");
                Console.WriteLine("9. Delete Bill");
                Console.WriteLine("10. Update Food");
                Console.WriteLine("11. Update Drink");
                Console.WriteLine("12. Update Staff");
                Console.WriteLine();
                Console.Write("Select option: ");

                double p;
                string n;
                int option;
```



```
if (Int32.TryParse(Console.ReadLine(), out option))
{
    switch (option)
    {
        case 1://Find Food
            Console.Write("Enter Food ID: ");
            if(!Int32.TryParse(Console.ReadLine(), out id))
                Console.WriteLine("ID is Not Valid!");
            obj = objectFactory.Get(ObjectType.Food, id);

            if(obj != null)
            {
                Console.WriteLine(obj.GetInfo());
            }else
            {
                Console.WriteLine("Data Not Found!");
            }

            break;

        case 2://Find Drink

            Console.Write("Enter Drink ID: ");
            if (!Int32.TryParse(Console.ReadLine(), out id))
                Console.WriteLine("ID is Not Valid!");
            obj = objectFactory.Get(ObjectType.Drink, id);

            if (obj != null)
            {
                Console.WriteLine(obj.GetInfo());
            }
            else
            {
                Console.WriteLine("Data Not Found!");
            }

            break;

        case 3://Find Staff

            Console.Write("Enter Staff ID: ");
            if (!Int32.TryParse(Console.ReadLine(), out id))
                Console.WriteLine("ID is Not Valid!");
            obj = objectFactory.Get(ObjectType.Staff, id);
```

```

        if (obj != null)
        {
            Console.WriteLine(obj.GetInfo());
        }
        else
        {
            Console.WriteLine("Data Not Found!");
        }

        break;

    case 4://Create and save Food

        Console.Write("Enter Food ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");
        Console.Write("Enter Food Name: ");
        n = Console.ReadLine();
        Console.Write("Enter Food Price: ");
        if (!Double.TryParse(Console.ReadLine(), out p))
            Console.WriteLine("Price is Not Valid!");
        objectFactory.Add(ObjectType.Food, new Food()
        { id = id, foodName = n, foodPrice = p });

        break;

    case 5://Create and save Drink

        Console.Write("Enter Drink ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");
        Console.Write("Enter Drink Name: ");
        n = Console.ReadLine();
        Console.Write("Enter Drink Price: ");
        if (!Double.TryParse(Console.ReadLine(), out p))
            Console.WriteLine("Price is Not Valid!");
        objectFactory.Add(ObjectType.Drink, new Drink()
        { id = id, drinkName = n, drinkPrice = p });

        break;

    case 6://Create and save Staff

        Console.Write("Enter Staff ID: ");

```

```

        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");

        Console.Write("Enter Staff Name: ");
        n = Console.ReadLine();
        Console.Write("Enter Staff Position: ");
        string pos = Console.ReadLine();

        objectFactory.Add(ObjectType.Staff, new Staff()
        { id = id, staffName = n, staffPosition = pos });

        break;

case 7://Create Bill

    DateTime aDateTime = DateTime.Now;
    Console.WriteLine("Issue Date: " + aDateTime);

    double totalFood = 0;
    double totalDrink = 0;
    double totalAmount;

    Console.Write("Enter Staff ID: ");
    if (!Int32.TryParse(Console.ReadLine(), out id))
        Console.WriteLine("ID is Not Valid!");
    obj = objectFactory.Get(ObjectType.Staff, id);
    Console.WriteLine(obj.GetInfo());

    while (true)
    {
        Console.Write("Enter Food ID: ");
        string s = Console.ReadLine();
        if (s == "end") break;
        if (!Int32.TryParse(s, out id))
            Console.WriteLine("ID is Not Valid!");
        Food f1 = (Food)
            objectFactory.Get(ObjectType.Food, id);
        Console.WriteLine(f1.GetInfo());
        totalFood += f1.foodPrice;
    }

    while (true)
    {
        Console.Write("Enter Drink ID: ");
        string s = Console.ReadLine();
    }

```

```

        if (s == "end") break;
        if (!Int32.TryParse(s, out id))
            Console.WriteLine("ID is Not Valid!");

        Drink f1 = (Drink)
            objectFactory.Get(ObjectType.Drink, id);
        Console.WriteLine(f1.GetInfo());
        totalDrink += f1.drinkPrice;
    }

    Console.Write("Enter VAT: ");
    double d;
    Double.TryParse(Console.ReadLine(), out d);
    totalAmount = (totalDrink + totalFood) * d;

    Console.Write("Enter Bill ID: ");
    if (!Int32.TryParse(Console.ReadLine(), out id))
        Console.WriteLine("ID is Not Valid!");

    objectFactory.Add(ObjectType.Bill, new Bill()
    {
        id = id,
        totalFood = totalFood,
        totalDrink = totalDrink,
        totalAmount = totalAmount
    });

    Console.WriteLine("VAT: {0:P}", d);
    Console.WriteLine("Total Food {0:C}: ", totalFood);
    Console.WriteLine("Total Drink: {0:C}", totalDrink);
    Console.WriteLine("Total Amount: {0:C}", totalAmount);

    break;

case 8://Find Bill

    Console.Write("Enter Bill ID: ");
    if (!Int32.TryParse(Console.ReadLine(), out id))
        Console.WriteLine("ID is Not Valid!");
    obj = objectFactory.Get(ObjectType.Bill, id);

    if (obj != null)
    {
        Console.WriteLine(obj.GetInfo());
    }

```

```
        else
        {
            Console.WriteLine("Data Not Found!");
        }

        break;

    case 9://Delete Bill

        Console.Write("Enter Bill ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");
        Console.Write("Deleted this Bill");
        objectFactory.Remove(ObjectType.Bill, id);

        break;

    case 10://Update Food

        Console.Write("Enter Old Food ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");

        Food f = (Food) objectFactory.Get(ObjectType.Food, id);

        Console.Write("Enter New Food ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");
        Console.Write("Enter New Food Name: ");
        n = Console.ReadLine();
        Console.Write("Enter New Food Price: ");
        if (!Double.TryParse(Console.ReadLine(), out p))
            Console.WriteLine("Price is Not Valid!");

        f.Update(id, n, p);

        break;

    case 11://Update Drink

        Console.Write("Enter Old Drink ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");

        Drink dri = (Drink)
```

```

        objectFactory.Get(ObjectType.Drink, id);

        Console.Write("Enter New Drink ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");

        Console.Write("Enter New Drink Name: ");
        n = Console.ReadLine();
        Console.Write("Enter New Drink Price: ");
        if (!Double.TryParse(Console.ReadLine(), out p))
            Console.WriteLine("Price is Not Valid!");

        dri.Update(id, n, p);

        break;

case 12://Update Staff

        Console.Write("Enter Old Staff ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");

        Staff sf = (Staff)
            objectFactory.Get(ObjectType.Staff, id);

        Console.Write("Enter New Staff ID: ");
        if (!Int32.TryParse(Console.ReadLine(), out id))
            Console.WriteLine("ID is Not Valid!");
        Console.Write("Enter New Staff Name: ");
        n = Console.ReadLine();
        Console.Write("Enter New Staff Position: ");
        string posi = Console.ReadLine();

        sf.Update(id, n, posi);

        break;

default:
    break;
}

Console.ReadLine();
}

```

```

        else Console.WriteLine("Invalid Selection!");
    }
}

#region Object

public abstract class IObject
{
    public int id;

    public abstract string GetInfo();

    public int GetID()
    {
        return this.id;
    }
}

class Food : IObject
{
    public string foodName;
    public double foodPrice;

    public override string GetInfo()
    {
        return string.Format("foodID {0}. foodName {1}. foodPrice {2}.",
            id, foodName, foodPrice);
    }

    public void Update(int id, string foodName, double foodPrice)
    {
        this.id = id;
        this.foodName = foodName;
        this.foodPrice = foodPrice;
    }
}

class Drink : IObject
{
    public string drinkName;
    public double drinkPrice;
}

```

```
public override string GetInfo()
{
    return string.Format("drinkID {0}. drinkName {1}. drinkPrice {2}.",
        id, drinkName, drinkPrice);
}

public void Update(int id, string drinkName, double drinkPrice)
{
    this.id = id;

    this.drinkName = drinkName;
    this.drinkPrice = drinkPrice;
}
}

class Staff : IObject
{
    public string staffName;
    public string staffPosition;

    public override string GetInfo()
    {
        return string.Format("staffID {0}. staffName {1}. staffPosition {2}.",
            id, staffName, staffPosition);
    }

    public void Update(int id, string staffName, string staffPosition)
    {
        this.id = id;
        this.staffName = staffName;
        this.staffPosition = staffPosition;
    }
}

class Bill : IObject
{
    public double totalFood;
    public double totalDrink;
    public double totalAmount;

    public override string GetInfo()
    {
        return string.Format("totalFood {0}. totalDrink {1}. totalAmount {2}.",
            totalFood, totalDrink, totalAmount); ;
    }
}
```



```

    }
}
enum ObjectType
{
    Food,
    Drink,
    Staff,
    Bill
}

#endregion Object

#region Object Factory

class ObjectFactory
{
    ObjectType _type;
    int _id;
    Dictionary<ObjectType, List<IObject>>
    data = new Dictionary<ObjectType, List<IObject>>();

    public ObjectFactory() {

        List<IObject> foodList = new List<IObject>();

        foodList.Add(new Food()
        { id = 101, foodName = "Fish", foodPrice = 23.56 });
        foodList.Add(new Food()
        { id = 102, foodName = "Chicken", foodPrice = 55.23 });
        foodList.Add(new Food()
        { id = 103, foodName = "Craf", foodPrice = 56.22 });

        List<IObject> drinkList = new List<IObject>();

        drinkList.Add(new Drink()
        { id = 201, drinkName = "Mango", drinkPrice = 5.3 });
        drinkList.Add(new Drink()
        { id = 202, drinkName = "Melon", drinkPrice = 8.3 });
        drinkList.Add(new Drink()
        { id = 203, drinkName = "Orange", drinkPrice = 5.7 });

        List<IObject> staffList = new List<IObject>();

        staffList.Add(new Staff()

```

```

        { id = 301, staffName = "Lisa", staffPosition = "Cashier" });

        List<IObject> billList = new List<IObject>();
        data[ObjectType.Food] = foodList;
        data[ObjectType.Drink] = drinkList;
        data[ObjectType.Staff] = staffList;
        data[ObjectType.Bill] = billList;
    }
    public IObject Get(ObjectType type, int id)
    {

        _type = type;
        _id = id;

        return GetObject();
    }
    public void Add(ObjectType type, IObject obj)
    {
        data[type].Add(obj);
    }
    public void Remove(ObjectType type, int id)
    {
        IObject obj = Get(type, id);

        if(obj != null) data[type].Remove(obj);
    }
    public IObject GetObject()
    {
        IObject obj = null;

        foreach (IObject item in data[_type])
        {
            if (item.id == _id)
            {
                obj = item;
                break;
            }
        }
        return obj;
    }
}

#endregion Object Factory
}

```

2. Run/Test Program

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff
Select option: _
```

Figure 1: Original options table

- At "Select option", enter "1" to search for food information. Then enter food ID needs info search at "Enter Food ID". Due to this is finding info detail of food follow ID.

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff
Select option: 1
Enter Food ID: 101
foodID 101. foodName Fish. foodPrice 23.56.
```

Diagram annotations: A red box labeled "Input" points to "Select option: 1". A green box labeled "Output" points to the result line "foodID 101. foodName Fish. foodPrice 23.56.".

Figure 2: Result of option 1 - Find Food follow ID

- Similar such as to option 1, at "Select option", enter "2".

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff
Select option: 2
Enter Drink ID: 201
drinkID 201. drinkName Mango. drinkPrice 5.3.
```

Diagram annotations: A red box labeled "Input" points to "Select option: 2". A green box labeled "Output" points to the result line "drinkID 201. drinkName Mango. drinkPrice 5.3.".

Figure 3: Result of option 2 - Find Drink follow ID

- Similar such as to option 1, at "Select option", enter "3".

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 3
Enter Staff ID: 301
staffID 301. staffName Lisa. staffPosition Cashier.
  
```

Figure 4: Result of option 3 - Find Staff follow ID

- At "Select option", enter "4" to create and save for new food information. Then enter the new food ID at "Enter Food ID", the new food name at "Enter Food Name", the new food price at "Enter Food Price". After that, continue back option 1 to check the new food just created has been saved or not (similar such as to option 1, at "Select option", enter "1").

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Before:
Select option: 4
Enter Food ID: 123
Enter Food Name: Cat
Enter Food Price: 123.23

After:
Select option: 1
Enter Food ID: 123
foodID 123. foodName Cat. foodPrice 123.23.
  
```

Figure 5: Result of option 4 - Create and save food

- Similar such as to option 4, at "Select option", enter "5". After that, continue back option 2 to check the new food just created has been saved or not (similar such as to option 2, at "Select option", enter "2").

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 5
Enter Drink ID: 456
Enter Drink Name: tea
Enter Drink Price: 25.3

Select option: 2
Enter Drink ID: 456
drinkID 456. drinkName tea. drinkPrice 25.3.

```

Figure 6: Result of option 5 - Create and save drink

- Similar such as to option 4, at "Select option", enter "6". After that, continue back option 3 to check the new food just created has been saved or not (similar such as to option 3, at "Select option", enter "3").

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 6
Enter Staff ID: 789
Enter Staff Name: Mino
Enter Staff Position: Boss

Select option: 3
Enter Staff ID: 789
staffID 789. staffName Mino. staffPosition Boss.

```

Figure 7: Result of option 6 - Create and save staff

- At "Select option", enter "7" it will show the issue date is present date include the date, month, year, hour, minute, seconds. After that, enter staff ID at "Enter Staff ID" to show info of staff creates the bill. Similarly, continue to enter food ID at "Enter Food ID" and drink ID at "Enter Drink ID". Note, to finish entering food or drink information, let enter "end" at "Enter Food ID" or "Enter Drink ID". Then, enter vat at "VAT" to calculate and enter bill ID at "Enter Bill ID" to save bill new created.

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe

6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 7
Issue Date: 21/12/2020 1:25:45 AM
Enter Staff ID: 301
staffID 301. staffName Lisa. staffPosition Cashier.
Enter Food ID: 101
foodID 101. foodName Fish. foodPrice 23.56.
Enter Food ID: 102
foodID 102. foodName Chicken. foodPrice 55.23.
Enter Food ID: end
Enter Drink ID: 201
drinkID 201. drinkName Mango. drinkPrice 5.3.
Enter Drink ID: 201
drinkID 201. drinkName Mango. drinkPrice 5.3.
Enter Drink ID: end
Enter VAT: 0.2
Enter Bill ID: 123456
VAT: 20.00%
Total Food $78.79:
Total Drink: $10.60
Total Amount: $17.88
```

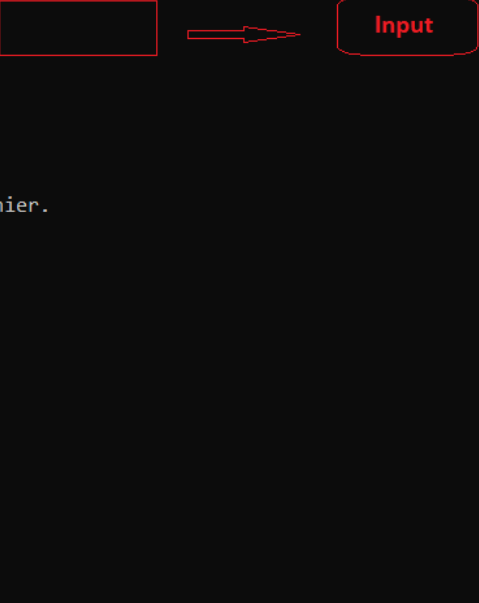


Figure 8: Result of option 7 - Create and save bill

- Next, at "Select option" enter "8" to implement check the new bill just created has been saved or not by bill ID (enter bill ID at "Enter Bill ID").

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe

1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 8
Enter Bill ID: 123456
totalFood 78.78999999999999. totalDrink 10.6. totalAmount 17.877999999999997.
```

Figure 9: Result of option 8 - Find bill

- Next, at "Select option" enter "9" to implement delete the new bill by bill ID (enter bill ID at "Enter Bill ID").

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 9
Enter Bill ID: 123456
Deleted this Bill
```

Figure 10: Result of option 9 - Delete bill

Then, to check this bill has been deleted or not, we will enter "8" at the "Select option" (similar such as option 8).

```
C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe
1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 8
Enter Bill ID: 123456
Data Not Found!
```

Figure 11: Result of option 9 - Check deleted bill – Find bill

- Next, at "Select option" enter "10" to update info food by food ID. First, enter food ID need update at "Enter Old Food ID". Second, enter info that needs an update. Finally, check again the info after the update by option 1.

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe

1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Select option: 10
Enter Old Food ID: 101
Enter New Food ID: 147
Enter New Food Name: Dog
Enter New Food Price: 23.23

Update

Select option: 1
Enter Food ID: 101
Data Not Found!

After update

Select option: 1
Enter Food ID: 147
foodID 147. foodName Dog. foodPrice 23.23.

Before update

Select option: 1
Enter Food ID: 101
foodID 101. foodName Fish. foodPrice 23.56.
    
```

Figure 12: Result of option 10 - Update information for food by food ID

- Similarly such as option 10. However, at "Select option" enter "11" and check again the info after the update by option 2.

```

C:\Users\Admin\source\repos\manageBillRestaurant\manageBillRestaurant\bin\Debug\netcoreapp3.1\manageBillRestaurant.exe

1. Find Food
2. Find Drink
3. Find Staff Info
4. Create and save Food
5. Create and save Drink
6. Create and save Staff Info
7. Create and save Bill
8. Find Bill
9. Delete Bill
10. Update Food
11. Update Drink
12. Update Staff

Update

Select option: 11
Enter Old Drink ID: 201
Enter New Drink ID: 258
Enter New Drink Name: Latte
Enter New Drink Price: 25.36

Select option: 2
Enter Drink ID: 201
Data Not Found!

After update

Select option: 2
Enter Drink ID: 258
drinkID 258. drinkName Latte. drinkPrice 25.36.

Before update

Select option: 2
Enter Drink ID: 201
drinkID 201. drinkName Mango. drinkPrice 5.3.
    
```

Figure 13: Result of option 11 - Update information for drink by drink ID

- Similarly such as option 10 and 11. However, at "Select option" enter "12" and check again the info after the update by option 3.

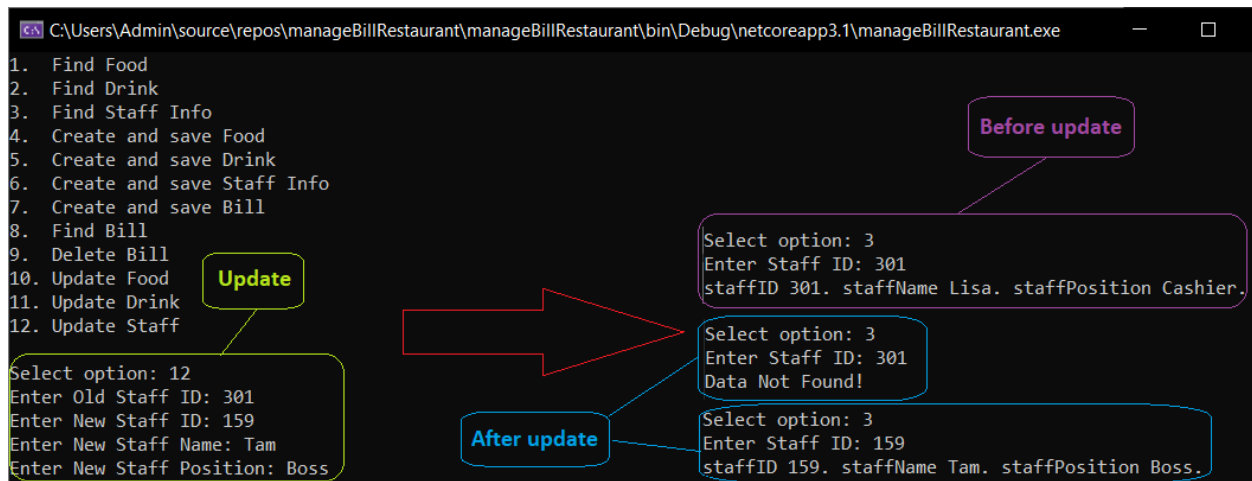


Figure 14: Result of option 12 - Update information for staff by staff ID

M3 Develop code that implements a design pattern for a given purpose

We use the factory design pattern for our program. Factory design pattern comes under a creational pattern as this pattern provides one of the best ways to create an object. In the Factory pattern, we create objects without exposing the creation logic to the client and refer to the newly created objects using a common interface.

Source Code for factory design pattern:

```
#region Object Factory

class ObjectFactory
{
    ObjectType _type;
    int _id;
    Dictionary<ObjectType, List<IObject>>
    data = new Dictionary<ObjectType, List<IObject>>();

    public ObjectFactory() {

        List<IObject> foodList = new List<IObject>();

        foodList.Add(new Food()
        { id = 101, foodName = "Fish", foodPrice = 23.56 });
        foodList.Add(new Food()
        { id = 102, foodName = "Chicken", foodPrice = 55.23 });
        foodList.Add(new Food()
```

```

        { id = 103, foodName = "Craf", foodPrice = 56.22 });

        List<IObject> drinkList = new List<IObject>();

        drinkList.Add(new Drink()
        { id = 201, drinkName = "Mango", drinkPrice = 5.3 });

        drinkList.Add(new Drink()
        { id = 202, drinkName = "Melon", drinkPrice = 8.3 });
        drinkList.Add(new Drink()
        { id = 203, drinkName = "Orange", drinkPrice = 5.7 });
        List<IObject> staffList = new List<IObject>();
        staffList.Add(new Staff()
        { id = 301, staffName = "Lisa", staffPosition = "Cashier" });
        List<IObject> billList = new List<IObject>();
        data[ObjectType.Food] = foodList;
        data[ObjectType.Drink] = drinkList;
        data[ObjectType.Staff] = staffList;
        data[ObjectType.Bill] = billList;
    }
    public IObject Get(ObjectType type, int id)
    {
        _type = type;
        _id = id;
        return GetObject();
    }
    public void Add(ObjectType type, IObject obj)
    {
        data[type].Add(obj);
    }
    public void Remove(ObjectType type, int id)
    {
        IObject obj = Get(type, id);

        if(obj != null) data[type].Remove(obj);
    }
    public IObject GetObject()
    {
        IObject obj = null;
        foreach (IObject item in data[_type])
        {
            if (item.id == _id)
            {
                obj = item;
                break;
            }
        }
    }

```

```

    }
    return obj;
  }
}

#endregion Object Factory

```

LO4 Investigate scenarios with respect to design patterns

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

1. Creational pattern

Creational Patterns provide ways to create objects while hiding the creation logic, instead of instantiating objects directly using the new operator. This gives the program more flexibility in deciding which objects need to be created for a given use case.

There are 6 types of creational design patterns: Factory Method Pattern, Abstract Factory Pattern, Singleton Pattern, Prototype Pattern, Builder Pattern, Object Pool Pattern.

Below, I will introduce detail two types of design patterns are Singleton Pattern and Builder Pattern.

1.1. Builder pattern

The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

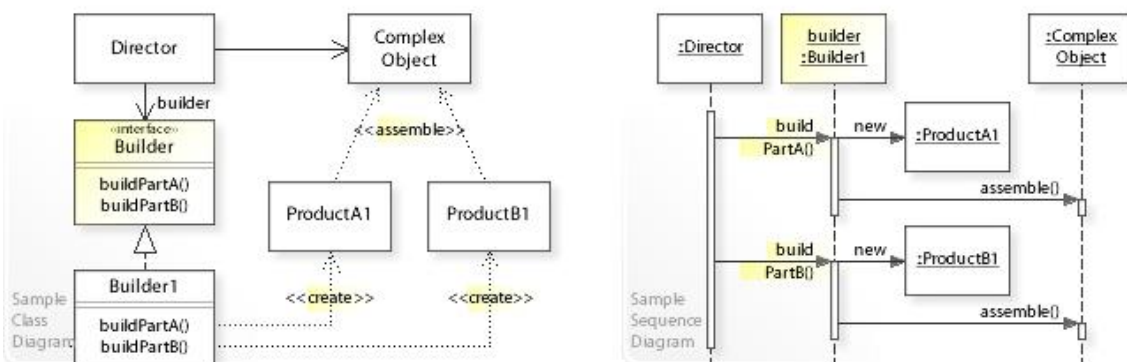


Figure 15: Structure of builder pattern

Example code:

```

/// <summary>
/// Represents a product created by the builder
/// </summary>
public class Car

{
    public string Make { get; set; }
    public string Model { get; set; }
    public int NumDoors { get; set; }
    public string Colour { get; set; }
    public Car(string make, string model, string colour, int numDoors)
    {
        Make = make;
        Model = model;
        Colour = colour;
        NumDoors = numDoors;
    }
}

/// <summary>
/// The builder abstraction
/// </summary>
public interface ICarBuilder
{
    string Colour { get; set; }
    int NumDoors { get; set; }
    Car GetResult();
}

/// <summary>
/// Concrete builder implementation
/// </summary>
public class FerrariBuilder : ICarBuilder
{
    public string Colour { get; set; }
    public int NumDoors { get; set; }

    public Car GetResult()
    {
        return NumDoors == 2 ?
            new Car("Ferrari", "488 Spider", Colour, NumDoors) : null;
    }
}

```

```
/// <summary>
/// The director
/// </summary>
public class SportsCarBuildDirector
{
    private ICarBuilder _builder;
    public SportsCarBuildDirector(ICarBuilder builder)
    {
        _builder = builder;
    }

    public void Construct()
    {
        _builder.Colour = "Red";
        _builder.NumDoors = 2;
    }
}

public class Client
{
    public void DoSomethingWithCars()
    {
        var builder = new FerrariBuilder();
        var director = new SportsCarBuildDirector(builder);

        director.Construct();
        Car myRaceCar = builder.GetResult();
    }
}
```

1.2. Singleton pattern

In singleton design pattern ensures a class has only one instance in the program and provides a global point of access to it. Mean a class that only allows a single instance of itself to be created and usually gives simple access to that instance. Most commonly, singletons don't allow any parameters to be specified when creating the instance since the second request of an instance with a different parameter could be problematic. If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.

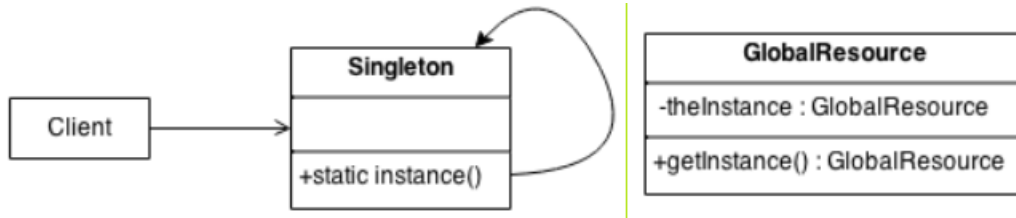


Figure 16: Structure of Singleton pattern

Example code:

```

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

    }

    private static Singleton instance = null;

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

```

2. Structural pattern

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplify the structure by identifying relationships. These patterns focus on, how the classes inherit from each other and how they are composed of other classes.

There are 7 types of structural design patterns: Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, Proxy Pattern.

Below, I will introduce the detail of two types of design patterns are Adapter Pattern and

Decorator Pattern.

2.1. Adapter pattern

The adapter pattern is a software design pattern (also known as Wrapper, an alternative naming shared with the decorator pattern) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

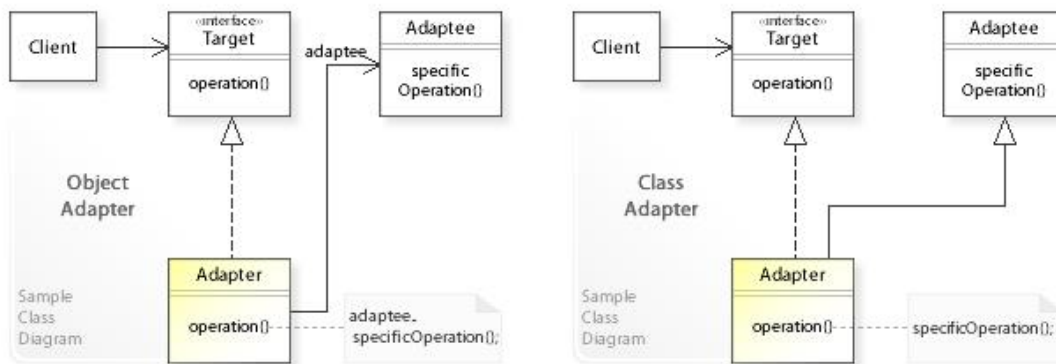


Figure 17: Structure of adapter pattern

Example code:

```

using System;

namespace RefactoringGuru.DesignPatterns.Adapter.Conceptual
{
    // The Target defines the domain-specific interface used by the client code.
    public interface ITarget
    {
        string GetRequest();
    }

    // The Adaptee contains some useful behavior, but its interface is
    // incompatible with the existing client code. The Adaptee needs some
    // adaptation before the client code can use it.
    class Adaptee
    {
        public string GetSpecificRequest()
        {
            return "Specific request.";
        }
    }
}
    
```

```
// The Adapter makes the Adaptee's interface compatible with the Target's
// interface.
class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        this._adaptee = adaptee;
    }

    public string GetRequest()
    {
        return $"This is '{this._adaptee.GetSpecificRequest()}';"
    }
}

class Program
{
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        ITarget target = new Adapter(adaptee);

        Console.WriteLine("Adaptee interface is incompatible with the client.");
        Console.WriteLine("But with adapter client can call it's method.");

        Console.WriteLine(target.GetRequest());
    }
}
```

2.2. Decorator pattern

A decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

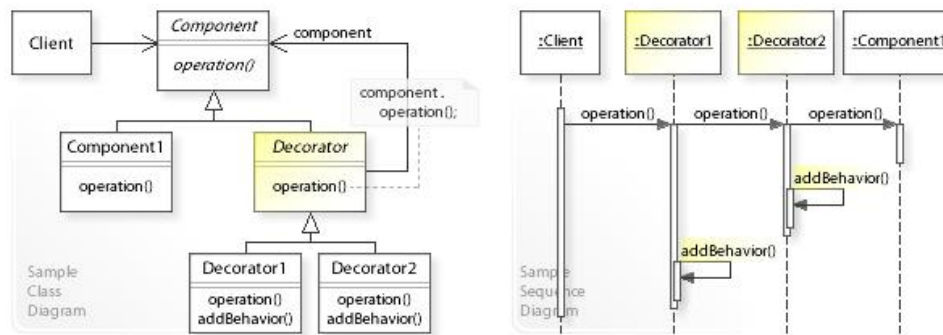


Figure 18: Structure of Decorator pattern

Example code:

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

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

3. Behavioral pattern

Behavioral design patterns are concerned with the interaction and responsibility of objects. In these design patterns, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled. That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

There are 12 types of structural design patterns: Chain of Responsibility Pattern, Command Pattern, Interpreter Pattern, Iterator Pattern, Mediator Pattern, Memento Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Pattern, Visitor Pattern, Null Object.

Below, I will introduce detail two types of design pattern is Observer Pattern and Interpreter Pattern

3.1. Observer pattern

The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

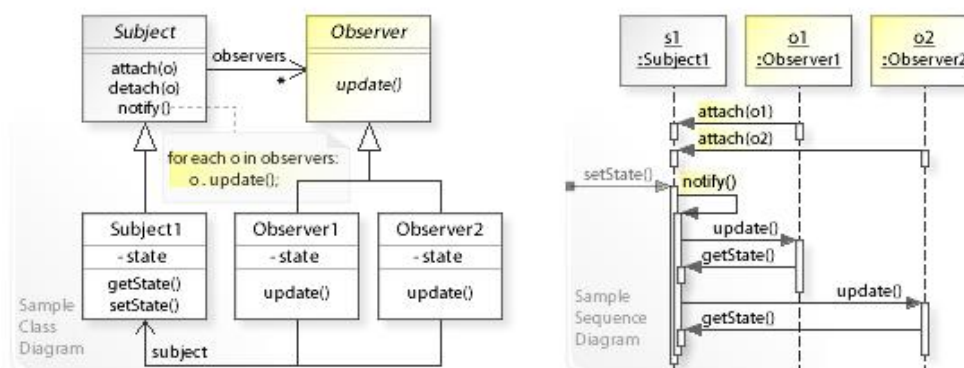


Figure 19: Structure of observer pattern

Example code:

```
public class Payload
{
    public string Message { get; set; }
}

public class Subject : IObservable<Payload>
{
    public IList<IObserver<Payload>> Observers { get; set; }

    public Subject()
    { Observers = new List<IObserver<Payload>>(); }

    public IDisposable Subscribe(IObserver<Payload> observer)
    {
        if (!Observers.Contains(observer))
        { Observers.Add(observer); }
        return new Unsubscriber(Observers, observer);
    }

    public void SendMessage(string message)
    {
        foreach (var observer in Observers)
        { observer.OnNext(new Payload { Message = message }); }
    }
}

public class Unsubscriber : IDisposable
{
    private IObserver<Payload> observer;
    private IList<IObserver<Payload>> observers;
    public Unsubscriber(IList<IObserver<Payload>>
        observers, IObserver<Payload> observer)
    {
        this.observers = observers;
        this.observer = observer;
    }

    public void Dispose()
    {
        if (observer != null && observers.Contains(observer))
        { observers.Remove(observer); }
    }
}
```

```
public class Observer : IObservable<Payload>
{
    public string Message { get; set; }

    public void OnCompleted(){}

    public void OnError(Exception error) {}

    public void OnNext(Payload value)
    {Message = value.Message; }

    public IDisposable Register(Subject subject)
    {
        return subject.Subscribe(this); }
}
```

3.2. Iterator pattern

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

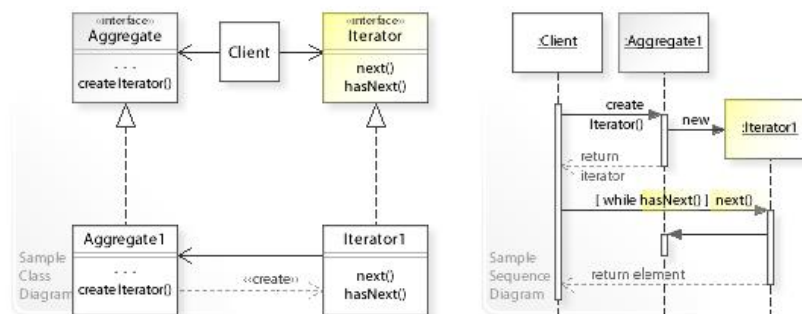


Figure 20: Structure of Iterator pattern

Example code:

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

M4 Reconcile the most appropriate design pattern from a range with a series of given scenarios

We use the Factory Design Pattern because Factory Method Pattern allows the sub-classes to choose the type of objects to create. It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class so that it will work with any classes that implement that interface or that extend that abstract class.

Because we need to create complex products. Means "encapsulate" a complex product creation process. It has a single point of control for multiple products, or need to manage the lifetime of resources that these products consume from a single point of reference. In some cases, it is too complicated to rewrite product instantiation and handling code with "new". So, the factory method ensures reuse and uniformity while minimizing potential coding errors.

On the other hand, the factory is useful for "harmonize" the creation of different objects through a shared Factory. This can be used to abstract away from specific Constructor-details for each subclass of a common superclass. It can be used to unify the creation of objects sharing a commonality. An example might be said in a visual programming IDE where we are using a toolkit in 2 or more projects open at the same time; the toolkit and its features would most likely be instantiated by a factory pattern because it is so complex graphically and logically, and it requires careful management of one or more of its instances.

In short, using the factory design pattern allows us to hide the implementation of an application seam (the core interfaces that make up our application); allows us to easily test the seam of an application (that is to mock/stub) certain parts of your application so we can build and test the other parts; allows us to change the design of our application more readily, this is known as loose coupling.

REFERENCES

- [1] Dan Clark, 2020. *Beginning C# Object-Oriented Programming*. 2nd ed. [ebook] Available at: <https://drive.google.com/drive/folders/11Ad6JnmaMWjmAgen0d4fsqWnUQSIqrus> [Accessed 28 November 2020].
- [2] Tam Phan, 2020. *Unit 20: Advanced Programming (1651)*. [lecture] Available at: <https://drive.google.com/drive/folders/1LWDEnXlwM0Ru6NS3nOyQ1rs90ekD3G38> [Accessed 28 November 2020]
- [3] Manish Agrahari, 2020. *Introduction To Object Oriented Programming Concepts In C#*. [online] C-sharpcorner.com. Available at: <https://www.c-sharpcorner.com/UploadFile/mkagrahari/introduction-to-object-oriented-programming-concepts-in-C-Sharp/#:~:text=A%20class%20is%20the%20core,perform%20operations%20on%20the%20data.>> [Accessed 28 November 2020].
- [4] Mahesh Alle, 2020. *Singleton Design Pattern In C#*. [online] C-sharpcorner.com. Available at: <https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/> [Accessed 4 December 2020].
- [5] En.wikipedia.org. 2020. *Decorator Pattern*. [online] Available at: https://en.wikipedia.org/wiki/Decorator_pattern#Structure [Accessed 4 December 2020].
- [6] En.wikipedia.org. 2020. *Iterator Pattern*. [online] Available at: https://en.wikipedia.org/wiki/Iterator_pattern#Structure [Accessed 4 December 2020].
- [7] Wrapper, 2020. *Decorator*. [online] Refactoring.guru. Available at: <https://refactoring.guru/design-patterns/decorator#:~:text=Decorator%20is%20a%20structural%20design,objects%20that%20contain%20the%20behaviors.>> [Accessed 4 December 2020].
- [8] Vaskaran Sarcar and Priya Shimanthoor, 2018. *Design Patterns In C#*. [ebook] Available at: <https://drive.google.com/drive/folders/11Ad6JnmaMWjmAgen0d4fsqWnUQSIqrus> [Accessed 4 December 2020].
- [9] Admin, 2020. *Behavioral Design Patterns - Javatpoint*. [online] www.javatpoint.com. Available at: <https://www.javatpoint.com/behavioral-design-patterns> [Accessed 20 December 2020].
- [10] En.wikipedia.org. 2020. *Adapter Pattern*. [online] Available at: https://en.wikipedia.org/wiki/Adapter_pattern#Structure [Accessed 20 December 2020].

- [11] Refactoring.guru. 2020. *Design Patterns: Adapter In C#*. [online] Available at: <<https://refactoring.guru/design-patterns/adapter/csharp/example#:~:text=Adapter%20is%20a%20structural%20design,recognizable%20by%20the%20second%20object.>> [Accessed 20 December 2020].
- [12] En.wikipedia.org. 2020. *Builder Pattern*. [online] Available at: <https://en.wikipedia.org/wiki/Builder_pattern#Structure> [Accessed 20 December 2020].
- [13] Tutorialspoint.com. 2020. *Design Pattern - Factory Pattern - Tutorialspoint*. [online] Available at: <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm> [Accessed 20 December 2020].
- [14] www.javatpoint.com. 2020. *Factory Method Design Pattern - Javatpoint*. [online] Available at: <<https://www.javatpoint.com/factory-method-design-pattern>> [Accessed 20 December 2020].
- [15] Pete Smith-Kline, 2013. *What Are The Pros And Cons Of The Factory Design Pattern?* - Quora. [online] Quora.com. Available at: <<https://www.quora.com/What-are-the-pros-and-cons-of-the-factory-design-pattern>> [Accessed 20 December 2020].
- [16] [duplicate], w., Lee, A., Analysis, F. and Tylychko, A., 2020. *What's The Advantage Of Factory Pattern?*. [online] Stack Overflow. Available at: <<https://stackoverflow.com/questions/9141178/whats-the-advantage-of-factory-pattern>> [Accessed 20 December 2020].