Object-oriented Software Design

Nutrition management software

Lecturer: Huynh Xuan Phung

Decorator

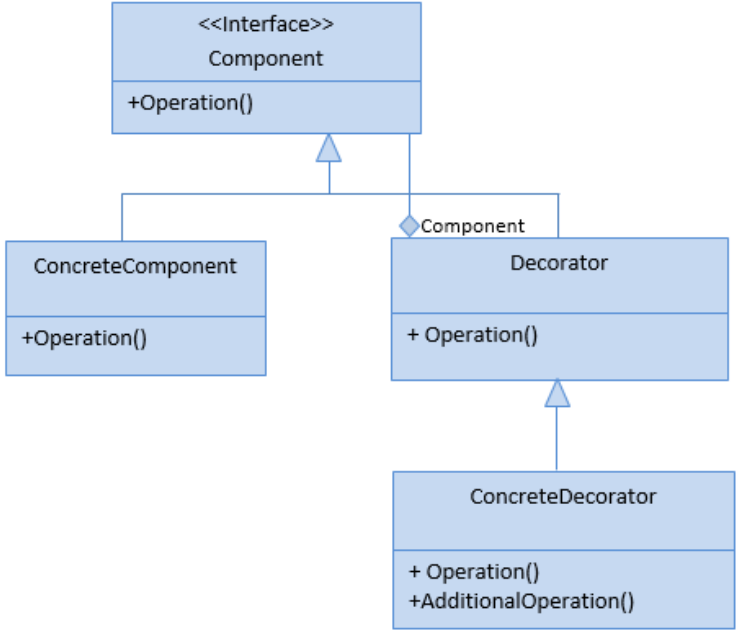
Decorator is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects.

Apply Decorator in nutrition management software to give appropriate advice to users about the current weight of the user.

The **Decorator pattern** is a structural **pattern** that lets you attach additional functionalities to an object dynamically. In other words, the client has the freedom to create an object and then extend it by adding a variety of “**features**” to it.

Decorator

The following class diagram shows us the Decorator Pattern's design:



Decorator Pattern

- **Component:** It defines the interface of the actual object that needs functionality to be added dynamically to the ConcreteComponents.
- **ConcreteComponent:** The actual object in which the functionalities could be added dynamically.
- **Decorator:** This defines the interface for all the dynamic functionalities that can be added to the ConcreteComponent.
- **ConcreteDecorator:** All the functionalities that can be added to the ConcreteComponent. Each needed functionality will be one ConcreteDecorator class.

```
class Component : IComponent
{
    8 references
    public string Operation()
    {
        return "Hi, I have some advice for you !";
    }
}

5 references
class DecoratorA : IComponent
{
    private readonly IComponent _component;

    2 references
    public DecoratorA(IComponent component)
    {
        _component = component;
    }

    // coi như "kế thừa" phương thức này từ object gốc
    // nếu muốn bạn có thể "giả lập ghi đè" bằng cách thay đổi nội dung phương thức này
    8 references
    public string Operation()
    {
        return _component.Operation();
    }

    // bổ sung phương thức này cho object gốc
    1 reference
    public string AddedBehavior()
    {
        return "You need to exercise more often";
    }
}

3 references
class DecoratorB : IComponent
{
    private readonly IComponent _component;

    1 reference
    public DecoratorB(IComponent component)
    {
        _component = component;
    }

    // giả lập ghi đè Operation
    8 references
    public string Operation()
    {
        var s = _component.Operation();
        return $"{s}. you can even buy weight loss pills to use";
    }
}
```

Decorator

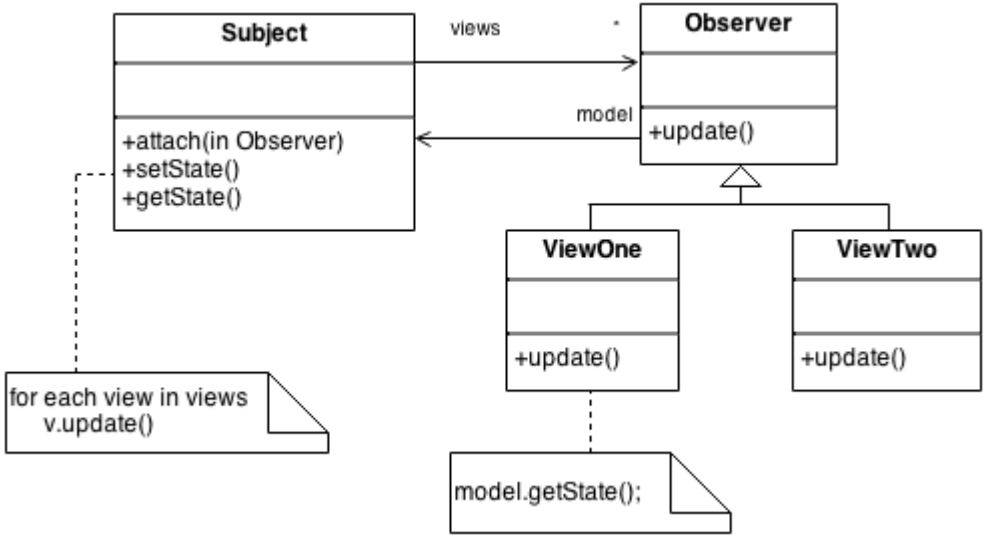
Observer

The **Decorator pattern** is a structural **pattern** that lets you attach additional functionalities to an object dynamically. In other words, the client has the freedom to create an object and then extend it by adding a variety of “**features**” to it.

Observer

Usage examples: The Observer pattern is pretty common in C# code, especially in the GUI components. It provides a way to react to events happening in other objects without coupling to their classes.

Identification: The pattern can be recognized by subscription methods, that store objects in a list and by calls to the update method issued to objects in that list.



Observer

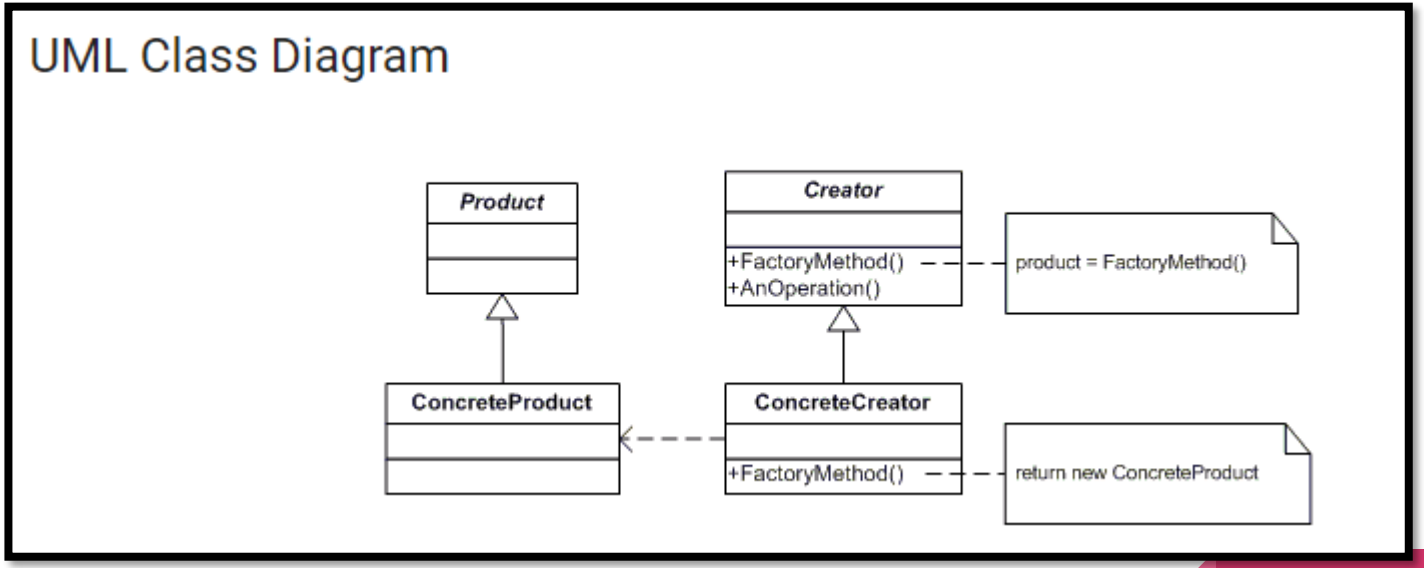
```
public class exercises2
{
    private bool needAttention = false;

    // Some of boys crushing this instance :))
    public IList<menbers> FriendZone = new List<menbers>();

    1 reference
    public void PostFacebook()
    {
        Console.WriteLine("New exercises have been updated");
        NeedAttention = true;
    }

    // State of instance. When state change, observe will know and react
    1 reference
    private bool NeedAttention
    {
        get => needAttention;
        set
        {
            needAttention = value;
            Notify();
        }
    }

    1 reference
    public void Notify()
    {
        foreach (var b in FriendZone)
        {
            b.Care();
        }
    }

    // Register observer.
    2 references
    public void AddToZone(menbers b)
    {
        FriendZone.Add(b);
    }
}
```

```
public class menbers
{
    public string Name;

    2 references
    public menbers(string name)
    {
        Name = name;
    }

    1 reference
    public void Care()
    {
        Console.WriteLine($"{Name}: enjoy that exercise, now");
    }
}
```

Factory

- The function: Check the nutrition of meals
- I choose this design pattern because it’s very useful when you need to provide a high level of flexibility for your code.
- Factory methods can be recognized by creation methods, which create objects from concrete classes, but return them as objects of abstract type or interface.

Factory


UML Class Diagram
Product
Creator
+FactoryMethod()
+AnOperation()
product = FactoryMethod()
ConcreteProduct
ConcreteCreator
+FactoryMethod()
return new ConcreteProduct

Factory

Beef.cs | Meals.cs | PorkFactory.cs | MealsFactory.cs

factoryMethods02

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

namespace factoryMethods02
{
    8 references
    abstract class Meals
    {

        public string Name;
        public int Calories;
        public int Protein;

    }
}
```

Beef.cs | Meals.cs | PorkFactory.cs | MealsFactory.cs

factoryMethods02

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

namespace factoryMethods02
{
    4 references
    class Beef : Meals
    {
        3 references
        public Beef(int Calories, int Protein)
        {
            this.Name = "Beef";
            this.Calories = Calories;
            this.Protein = Protein;
        }
        0 references
        public int getCalories ()
        {
            return this.Calories;
        }
        0 references
        public int getProtein ()
        {
            return this.Protein;
        }

    }
}
```

Factory

Beef.cs Meals.cs PorkFactory.cs MealsFactory.cs Progra

factoryMethods02

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

namespace factoryMethods02.factory
{
    3 references
    abstract class MealsFactory
    {
        3 references
        public abstract Meals GetMeals();
    }
}
```

Beef.cs Meals.cs PorkFactory.cs MealsFactory.cs

factoryMethods02

```
using System;

namespace factoryMethods02
{
    0 references
    public class ClientApplication
    {
        0 references
        static void Main()
        {
            Meals factory = null;
            Console.Write("Enter the meals you would like to check: "
            string car = Console.ReadLine();

            switch (car.ToLower())
            {
                case "chicken":
                    factory = new Chicken(5, 10);
                    break;
                case "beef":
                    factory = new Beef(10, 500);
                    break;
                case "pork":
                    factory = new Pork(500, 1000);
                    break;
                default:
                    break;
            }


            Console.WriteLine("\nNutritional content per 100g : \n");
            Console.WriteLine("Meals Type: "+factory.Name);
            Console.WriteLine("Protein: "+factory.Protein);
            Console.WriteLine("Calories: "+factory.Calories);
            Console.ReadKey();
        }
    }
}
```

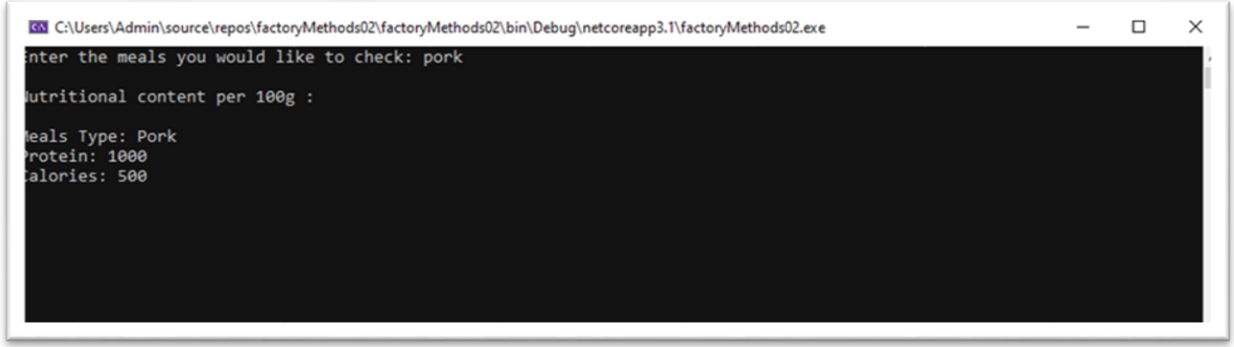
Beef.cs Meals.cs PorkFactory.cs MealsFactory

factoryMethods02

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

namespace factoryMethods02.factory
{
    1 reference
    class PorkFactory : MealsFactory
    {
        private int _Calories;
        private int _Protein;

        0 references
        public PorkFactory(int Calories, int Protein)
        {
            _Calories = Calories;
            _Protein = Protein;
        }

        3 references
        public override Meals GetMeals()
        {
            return new Pork(_Calories, _Protein);
        }
```

Factory

Result



Builder

- The function: Register

- I choose this design pattern because it’s especially useful when you need to create an object with lots of possible configuration options.

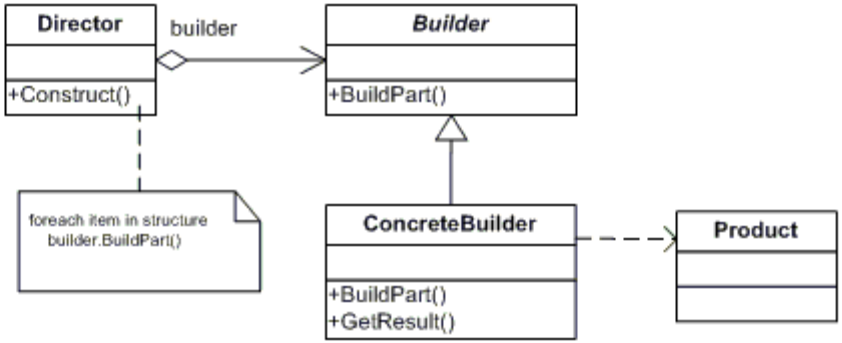
- The Builder pattern can be recognized in a class, which has a single creation method and several methods to configure the resulting object. Builder methods often support chaining

  For example,

  (someBuilder->setValueA(1)->setValueB(2)->create()).

Builder

UML Diagram



Builder

Builder.csproj | ConcreateBuilder.cs | BUILDER.cs | USERS.cs | Program.cs

Builder | Builder.Progr

```
using System;

namespace Builder
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var userBuilder = new ConcreateBuilder()
                .SetLName("Tan Thanh")
                .SetGender("Male")
                .SetOld("21t")
                .SetHeight("174cm")
                .SetWeight("70kg");

            Console.WriteLine(userBuilder.build());
        }
    }
}
```

Builder.csproj | ConcreateBuilder.cs | BUILDER.cs | USERS.cs | Program.cs

Builder | Builder.USERS

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

namespace Builder
{
    4 references
    public class USERS
    {
        private string firstName;
        private string lastName;
        private string gender;
        private string old;
        private string height;
        private string weight;
        private string email;


        1 reference
        public USERS( string firstName, string lastName, string gender, string old, string height, string weight, string email)
        {
            this.firstName = firstName;
            this.lastName = lastName;
            this.gender = gender;
            this.old = old;
            this.height = height;
            this.weight = weight;
            this.email = email;
        }
    }
}
```

Builder

Builder.csproj | ConcreteBuilder.cs | BUILDER.cs | USERS.cs

Builder

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

namespace Builder
{
    1 reference
    public class ConcreteBuilder : UserBuilder
    {
        private string firstName;
        private string lastName;
        private string gender;
        private string old;
        private string height;
        private string weight;
        private string email;


        1 reference
        public UserBuilder SetFName(string firstName)
        {
            this.firstName = firstName;
            return this;
        }

        2 references
        public UserBuilder SetLName(string lastName)
        {
            this.lastName = lastName;
            return this;
        }

        2 references
        public UserBuilder SetGender(string gender)
        {
            this.gender = gender;
            return this;
        }

```

Builder.csproj | ConcreteBuilder.cs | BUILDER.cs | USERS.cs | Program.cs

Builder | Builder.Co

```
        this.email = email;
        return this;
    }

    2 references
    public USERS build()
    {
        Console.WriteLine("Register User With Information Is: \n");

        if (firstName != null)
        {
            Console.WriteLine("First name: " + firstName.ToString());
        }
        if (lastName != null)
        {
            Console.WriteLine("Last name: " + lastName.ToString());
        }
        if (gender != null)
        {
            Console.WriteLine("Gender: " + gender.ToString());
        }
        if (old != null)
        {
            Console.WriteLine("Old: " + old.ToString());
        }
        if (height != null)
        {
            Console.WriteLine("Height: " + height.ToString());
        }
        if (weight != null)
        {
            Console.WriteLine("Weight: " + weight.ToString());
        }
        if (email != null)
        {
            Console.WriteLine("Email: " + email.ToString());
        }


        return new USERS(firstName, lastName, gender, old, height, weight, email);
    }
}
```

Builder.csproj | ConcreteBuilder.cs | BUILDER.cs

Builder

```
using System;

namespace Builder
{
    15 references
    public interface UserBuilder
    {
        2 references
        USERS build();

        1 reference
        UserBuilder SetFName(string firstName);
        2 references
        UserBuilder SetLName(string lastName);
        2 references
        UserBuilder SetGender(string gender);
        2 references
        UserBuilder SetOld(string old);
        2 references
        UserBuilder SetHeight(string height);
        2 references
        UserBuilder SetWeight(string weight);
        1 reference
        UserBuilder SetEmail(string email);
    }
}
```

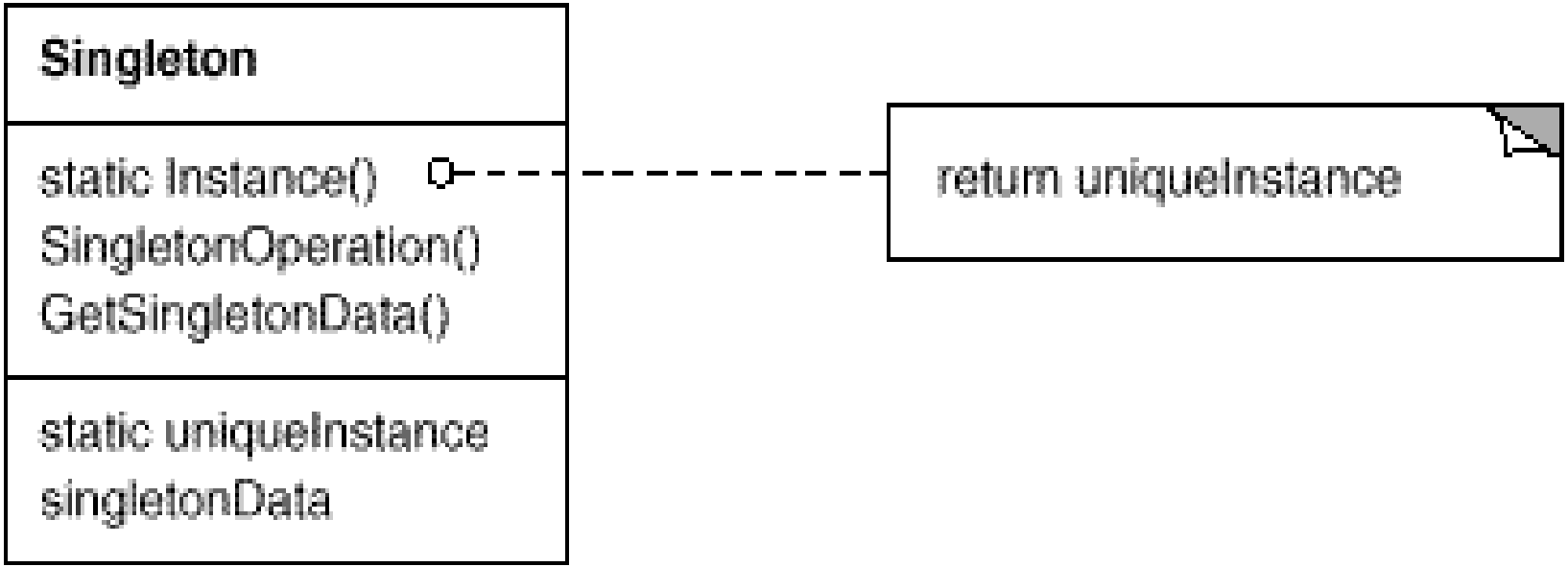
Builder

- Result

Microsoft Visual Studio Debug Console

```
Register User With Information Is:

Last name: Tan Thanh
Gender: Male
Old: 21t
Height: 174cm
Weight: 70kg
Builder.USERS

C:\Users\Admin\source\repos\Builder\Builder\bin\Debug\netcoreapp3.1\Builder.
To automatically close the console when debugging stops, enable Tools->Optio
le when debugging stops.
Press any key to close this window . . .
```

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Singleton

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
namespace Singleton
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            MySingleton mySingleton = MySingleton.Instance;
            mySingleton.DoSomething();
        }
    }
    5 references
    public class MySingleton
    {
        private static MySingleton instance = new MySingleton();
        1 reference
        private MySingleton() { }
        1 reference
        public static MySingleton Instance
        { get { return instance; } }

        1 reference
        public void DoSomething()
        {
            Console.WriteLine("Calorie and nutrition counters are probably the most popular type of nutrition app.
            Console.ReadLine();
        }

    }
}
```

Singleton

- Result:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;

C:\Users\maiho\source\repos\Singleton\Singleton\bin\Debug\netcoreapp3.1\Singleton.exe
Calorie and nutrition counters are probably the most popular type of nutrition app. Our main objective is to measure cal
ories consumed and burned by the user in a day.

        { get { return instance; } }

        1 reference
        public void DoSomething()
        {
            Console.WriteLine("Calorie and nutrition counters are probably the most popular type
            Console.ReadLine();
        }

    }
```