

Design Patterns - Exercise

Exercise problems for the ["C# Advanced" course @ Software University](#).

Problem solutions are provided if you struggle doing it on your own as it can be too abstract. The solutions are just examples, you don't need to follow them completely. There is no automated testing for this exercise, you have to validate what you've done by yourself.

I. Prototype

Your task is to create a console application for building sandwiches implementing the Prototype Design Pattern.

1. Abstract Class

First, you have to create an abstract class to represent a sandwich, and define a method by which the abstract Sandwich class can clone itself.

Solution

```
abstract class SandwichPrototype
{
    public abstract SandwichPrototype Clone();
}
```

2. ConcretePrototype participants

Now you need the **ConcretePrototype** participant class that can clone itself to create more Sandwich instances. Let's say that a Sandwich consists of four parts: the meat, cheese, bread, and veggies.

Solution

```
public class Sandwich : SandwichPrototype
{
    private string bread;
    private string meat;
    private string cheese;
    private string veggies;

    public Sandwich(string bread, string meat, string cheese, string veggies)
    {
        this.bread = bread;
        this.meat = meat;
        this.cheese = cheese;
        this.veggies = veggies;
    }

    public override SandwichPrototype Clone()
    {
        string ingredientList = GetIngredientList();
        Console.WriteLine("Cloning sandwich with ingredients: {0}", ingredientList);

        return MemberwiseClone() as SandwichPrototype;
    }

    private string GetIngredientList()
    {
        return $"{this.bread}, {this.meat}, {this.cheese}, {this.veggies}";
    }
}
```

3. Sandwich Menu

Let's create a class that will have the purpose to store the sandwiches we've made. It will be like a repository.

Solution

```
public class SandwichMenu
{
    private Dictionary<string, SandwichPrototype> _sandwiches =
        new Dictionary<string, SandwichPrototype>();

    public SandwichPrototype this[string name]
    {
        get { return _sandwiches[name]; }
        set { _sandwiches.Add(name, value); }
    }
}
```

4. Use what you've done

Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating the prototype and then cloning it, thereby populating your **SandwichMenu**.

Solution

```
public static void Main()
{
    SandwichMenu sandwichMenu = new SandwichMenu();

    // Initialize with default sandwiches
    sandwichMenu["BLT"] = new Sandwich("Wheat", "Bacon", "", "Lettuce, Tomato");
    sandwichMenu["PB&J"] = new Sandwich("White", "", "", "Peanut Butter, Jelly");
    sandwichMenu["Turkey"] = new Sandwich("Rye", "Turkey", "Swiss", "Lettuce, Onion, Tomato");

    // Deli manager adds custom sandwiches
    sandwichMenu["LoadedBLT"] = new Sandwich("Wheat", "Turkey, Bacon", "American", "Lettuce, Tomato, Onion, Olives");
    sandwichMenu["ThreeMeatCombo"] = new Sandwich("Rye", "Turkey, Ham, Salami", "Provolone", "Lettuce, Onion");
    sandwichMenu["Vegetarian"] = new Sandwich("Wheat", "", "", "Lettuce, Onion, Tomato, Olives, Spinach");

    // Now we can clone these sandwiches
    Sandwich sandwich1 = sandwichMenu["BLT"].Clone() as Sandwich;
    Sandwich sandwich2 = sandwichMenu["ThreeMeatCombo"].Clone() as Sandwich;
    Sandwich sandwich3 = sandwichMenu["Vegetarian"].Clone() as Sandwich;
}
```

```
C:\WINDOWS\system32\cmd.exe
Cloning sandwich with ingredients: Wheat, Bacon, , Lettuce, Tomato
Cloning sandwich with ingredients: Rye, Turkey, Ham, Salami, Provolone, Lettuce, Onion
Cloning sandwich with ingredients: Wheat, , , Lettuce, Onion, Tomato, Olives, Spinach
Press any key to continue . . .
```

II. Composite

Your task is to create a console application that calculates the total price of gifts that are being sold in a shop. The gift could be a single element (toy) or it can be a complex gift which consists of a box with two toys and another box with maybe one toy and the box with a single toy inside. We have a tree structure representing our complex gift so, implementing the Composite design pattern will be the right solution for us.

1. Component

First, you have to create an abstract class to represent the base gift. It should have two fields (name and price) and a method that calculates the total price. These fields and method are going to be used as an interface between the Leaf and the Composite part of our pattern.

Solution

```
public abstract class GiftBase
{
    protected string name;
    protected int price;

    public GiftBase(string name, int price)
    {
        this.name = name;
        this.price = price;
    }

    public abstract int CalculateTotalPrice();
}
```

2. Basic operations

Create an interface **IGiftOperations** that will contain two operations - Add and Remove (a gift). You should create the interface because the Leaf class doesn't need the operation methods.

Solution

```
public interface IGiftOperations
{
    void Add(GiftBase gift);
    void Remove(GiftBase gift);
}
```

3. Composite Class

Now you have to create the composite class (**CompositeGift**). It should inherit the **GiftBase** class and implement the **IGiftOperations** interface. Therefore, the implementation is pretty forward. It will consist of many objects from the **GiftBase** class. The **Add** method will add a gift and the **Remove** - will remove one. The **CalculateTotalPrice** method will return the price of the **CompositeGift**.

Solution

```
public class CompositeGift : GiftBase, IGiftOperations
{
    private List<GiftBase> _gifts;

    public CompositeGift(string name, int price)
        : base(name, price)
    {
        _gifts = new List<GiftBase>();
    }

    public void Add(GiftBase gift)
    {
        _gifts.Add(gift);
    }

    public void Remove(GiftBase gift)
    {
        _gifts.Remove(gift);
    }

    public override int CalculateTotalPrice()
    {
        int total = 0;

        Console.WriteLine($"{name} contains the following products with prices:");

        foreach (var gift in _gifts)
        {
            total += gift.CalculateTotalPrice();
        }

        return total;
    }
}
```

4. Leaf Class

You should also create a Leaf class (**SingleGift**). It will not have sub-levels so it doesn't require add and delete operations. Therefore, it should only inherit the **GiftBase** class. It will be like a single gift, without component gifts.

Solution

```
public class SingleGift : GiftBase
{
    public SingleGift(string name, int price)
        : base(name, price)
    {
    }

    public override int CalculateTotalPrice()
    {
        Console.WriteLine($"{name} with the price {price}");

        return price;
    }
}
```

5. Use what you've done

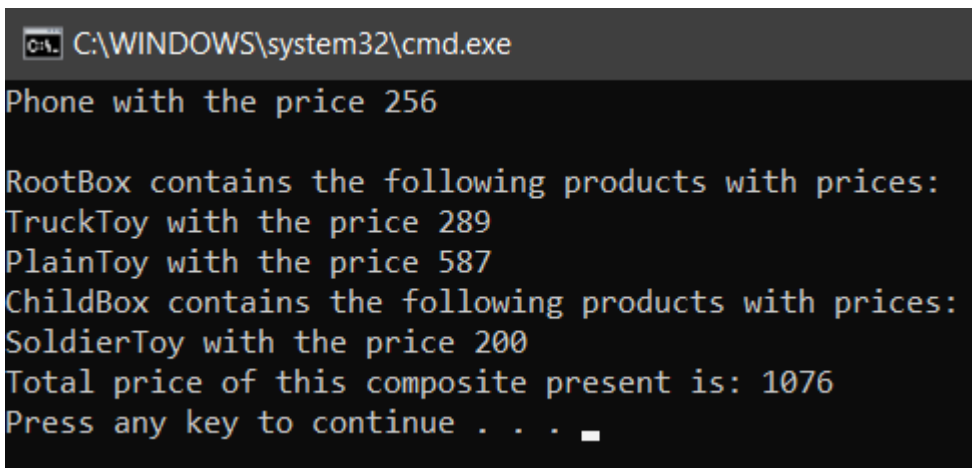
Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating the Leaf class (**SingleGift**) and the Composite class (**CompositeGift**) and using their methods.

Solution

```
public static void Main()
{
    var phone = new SingleGift("Phone", 256);
    phone.CalculateTotalPrice();
    Console.WriteLine();

    var rootBox = new CompositeGift("RootBox", 0);
    var truckToy = new SingleGift("TruckToy", 289);
    var plainToy = new SingleGift("PlainToy", 587);
    rootBox.Add(truckToy);
    rootBox.Add(plainToy);
    var childBox = new CompositeGift("ChildBox", 0);
    var soldierToy = new SingleGift("SoldierToy", 200);
    childBox.Add(soldierToy);
    rootBox.Add(childBox);

    Console.WriteLine($"Total price of this composite present is: {rootBox.CalculateTotalPrice()}");
}
```



```
C:\WINDOWS\system32\cmd.exe
Phone with the price 256

RootBox contains the following products with prices:
TruckToy with the price 289
PlainToy with the price 587
ChildBox contains the following products with prices:
SoldierToy with the price 200
Total price of this composite present is: 1076
Press any key to continue . . .
```

III. Template Pattern

There are easily hundreds of types of bread currently being made in the world, but each kind involves specific steps in order to make them. Your task is to model a few different kinds of bread that all use this same pattern, which is a good fit for the Template Design Pattern.

1. Abstract Class

First, you have to create an abstract class (**Bread**) to represent all breads we can bake. It should have two abstract void methods **MixIngredients()**, **Bake()**, one virtual void method **Slice()** and the template method - **Make()**.

Solution

```
public abstract class Bread
{
    public abstract void MixIngredients();

    public abstract void Bake();

    public virtual void Slice()
    {
        Console.WriteLine("Slicing the " + GetType().Name + " bread!");
    }

    // The template method
    public void Make()
    {
        MixIngredients();
        Bake();
        Slice();
    }
}
```

2. Concrete Classes

Extend the application by adding several Concrete Classes for different types of **Bread**. Examples: **TwelveGrain**, **Sourdough**, **WholeWheat**.

Solution

```
public class TwelveGrain : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for 12-Grain Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the 12-Grain Bread. (25 minutes)");
    }
}
```

```

class Sourdough : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for Sourdough Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the Sourdough Bread. (20 minutes)");
    }
}

class WholeWheat : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for Whole Wheat Bread.");
    }

    public override void Bake()
    {
        Console.WriteLine("Baking the Whole Wheat Bread. (15 minutes)");
    }
}

```

3. Use what you've done

Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating objects of the classes you've just made. It was that simple. In fact, this might be something you've been already using but you didn't know it was a Design Pattern.

Solution

```

public static void Main()
{
    Sourdough sourdough = new Sourdough();
    sourdough.Make();

    TwelveGrain twelveGrain = new TwelveGrain();
    twelveGrain.Make();

    WholeWheat wholeWheat = new WholeWheat();
    wholeWheat.Make();
}

```

C:\WINDOWS\system32\cmd.exe

```
Gathering Ingredients for Sourdough Bread.  
Baking the Sourdough Bread. (20 minutes)  
Slicing the Sourdough bread!  
Gathering Ingredients for 12-Grain Bread.  
Baking the 12-Grain Bread. (25 minutes)  
Slicing the TwelveGrain bread!  
Gathering Ingredients for Whole Wheat Bread.  
Baking the Whole Wheat Bread. (15 minutes)  
Slicing the WholeWheat bread!  
Press any key to continue . . .
```