Exercises: Design Patterns

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.

1. Singleton

We are going to create a simple console application in which we are going to read all the data from a file (which consist of cities with their population) and then use that data. So, to start off, let's create a single interface:

```
namespace SingletonDemo
{
    public interface ISingletonContainer
    {
        int GetPopulation(string name);
    }
}
```

After that, we have to create a class to implement the ISingletonContainer interface. We are going to call it SingletonDataContainer:

```
public class SingletonDataContainer : ISingletonContainer
{
    private Dictionary<string, int> _capitals = new Dictionary<string, int>();

    public SingletonDataContainer()
    {
        Console.WriteLine("Initializing singleton object");

        var elements = File.ReadAllLines("capitals.txt");
        for (int i = 0; i < elements.Length; i += 2)
        {
            _capitals.Add(elements[i], int.Parse(elements[i + 1]));
        }
    }

    public int GetPopulation(string name)
    {
        return _capitals[name];
    }
}
```

So, we have a dictionary in which we store the capital names and their population from our file. As we can see, we are reading from a file in our constructor. And that is all good. Now we are ready to use this class in any consumer by simply instantiating it. But is this really what we need to do, to instantiate the class which reads from a file which never changes (in this particular project. Population of the cities is changing daily). Of course not, so obviously using a Singleton pattern would be very useful here. Let's implement it:

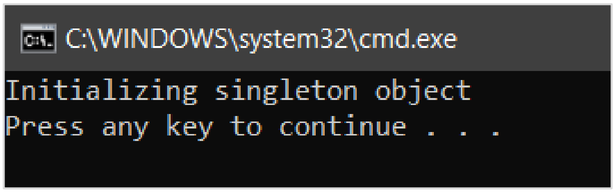
First, we will hide the constructor from the consumer classes by making it private. Then, we've created a single instance of our class and exposed it through the Instance property.

```
private static SingletonDataContainer instance = new SingletonDataContainer();

public static SingletonDataContainer Instance => instance;
```

At this point, we can call the Instance property as many times as we want, but our object is going to be instantiated only once and shared for every other call. Check it for yourself:

```
public static void Main()
{
    var db = SingletonDataContainer.Instance;
    var db2 = SingletonDataContainer.Instance;
    var db3 = SingletonDataContainer.Instance;
    var db4 = SingletonDataContainer.Instance;
}
```

The result in out console will be the following:



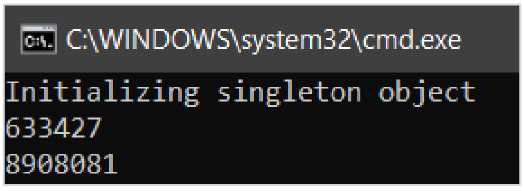
```
Initializing singleton object
Press any key to continue . . .
```

We can see that we are calling our instance four times but it is initialized only once, which is exactly what we want.

Let's check if our console program works:

```
public static void Main()
{
    var db = SingletonDataContainer.Instance;
    Console.WriteLine(db.GetPopulation("Washington, D.C."));
    var db2 = SingletonDataContainer.Instance;
    Console.WriteLine(db2.GetPopulation("London"));
}
```

The expected output should be something like this:

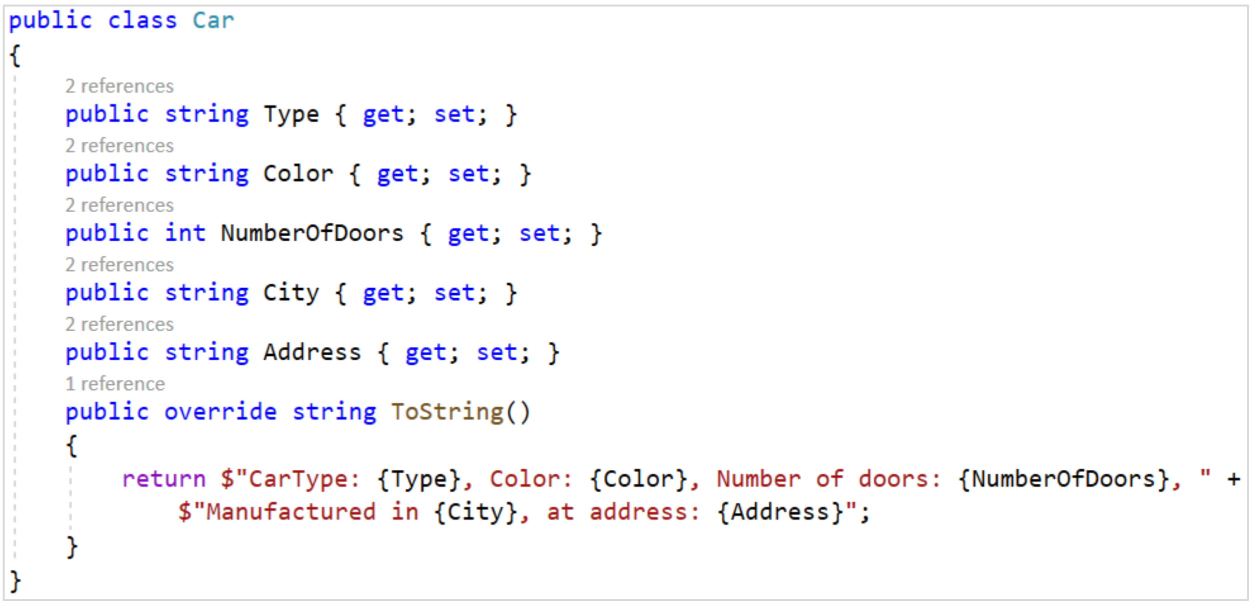


```
Initializing singleton object
633427
8908081
```

2. Facade

Now we will take a look at a Façade example implementation.

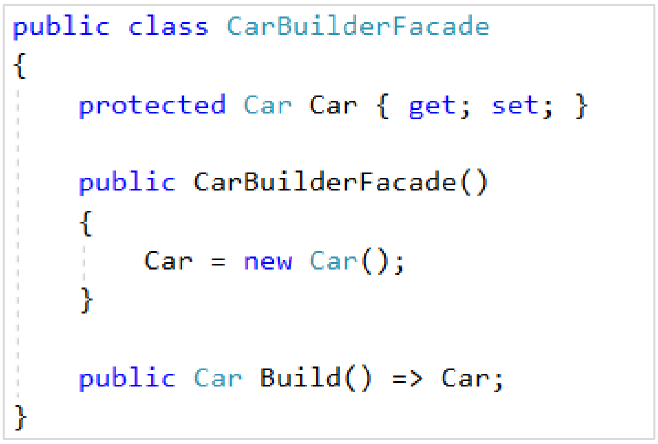
We will start off by creating a class to work with:



```
public class Car
{
    2 references
    public string Type { get; set; }
    2 references
    public string Color { get; set; }
    2 references
    public int NumberOfDoors { get; set; }
    2 references
    public string City { get; set; }
    2 references
    public string Address { get; set; }
    1 reference
    public override string ToString()
    {
        return $"CarType: {Type}, Color: {Color}, Number of doors: {NumberOfDoors}, " +
            $"Manufactured in {City}, at address: {Address}";
    }
}
```

We have the info part and the address part of our object, so we are going to use two builders to create this whole object.

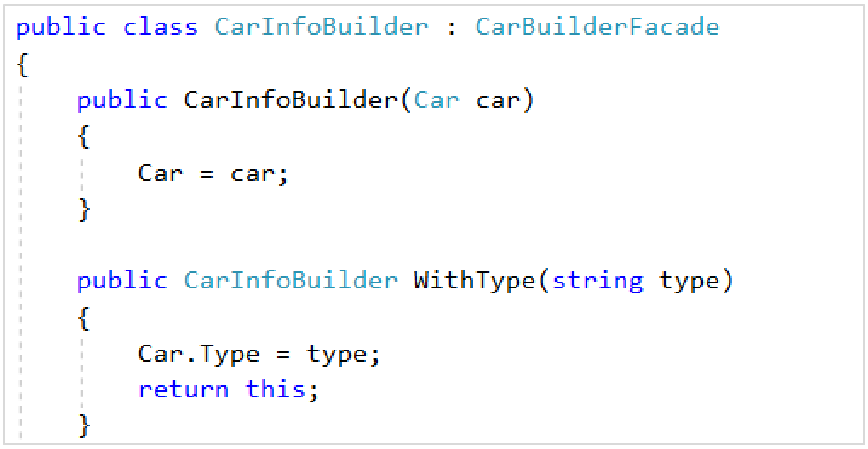
We need a façade, let’s create one:



```
public class CarBuilderFacade
{
    protected Car Car { get; set; }

    public CarBuilderFacade()
    {
        Car = new Car();
    }

    public Car Build() => Car;
}
```

We instantiate the **Car** object, which we want to build and expose it through the Build method.

What we need now is to create concrete builders. So, let’s start with the **CarInfoBuilder** which needs to inherit from the facade class:



```
public class CarInfoBuilder : CarBuilderFacade
{
    public CarInfoBuilder(Car car)
    {
        Car = car;
    }

    public CarInfoBuilder WithType(string type)
    {
        Car.Type = type;
        return this;
    }
```

```csharp
    public CarInfoBuilder WithColor(string color)
    {
        Car.Color = color;
        return this;
    }

    public CarInfoBuilder WithNumberOfDoors(int number)
    {
        Car.NumberOfDoors = number;
        return this;
    }
}
```

We receive, through the constructor, an object we want to build and use the fluent interface for building purpose.

Let's do the same for the **CarAddresBuilder** class:

```csharp
public class CarAddressBuilder : CarBuilderFacade
{
    public CarAddressBuilder(Car car)
    {
        Car = car;
    }

    public CarAddressBuilder InCity(string city)
    {
        Car.City = city;
        return this;
    }

    public CarAddressBuilder AtAddress(string address)
    {
        Car.Address = address;
        return this;
    }
}
```

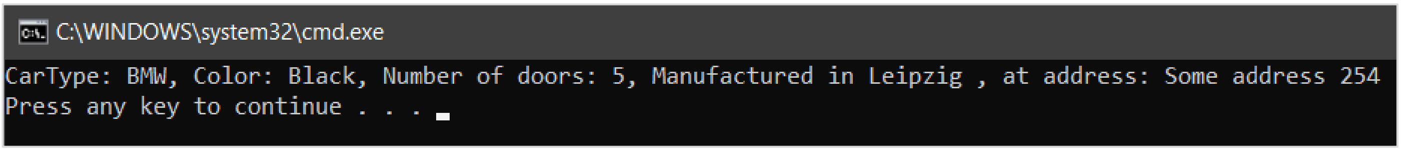
At this moment we have both builder classes, but we can't start building our object yet because we haven't exposed our builders inside the facade class. Well, let's do that:

```csharp
public class CarBuilderFacade
{
    protected Car Car { get; set; }

    public CarBuilderFacade()
    {
        Car = new Car();
    }

    public Car Build() => Car;

    public CarInfoBuilder Info => new CarInfoBuilder(Car);
    public CarAddressBuilder Built => new CarAddressBuilder(Car);
}
```

That's it, we can start building our object:

```
public static void Main()
{
    var car = new CarBuilderFacade()
        .Info
          .WithType("BMW")
          .WithColor("Black")
          .WithNumberOfDoors(5)
        .Built
          .InCity("Leipzig ")
          .AtAddress("Some address 254")
        .Build();

    Console.WriteLine(car);
}
```

And the output should be:



```
C:\WINDOWS\system32\cmd.exe
CarType: BMW, Color: Black, Number of doors: 5, Manufactured in Leipzig , at address: Some address 254
Press any key to continue . . .
```

3. Command Pattern

The Command design pattern consists of the Invoker class, Command class/interface, Concrete command classes and the Receiver class. Having that in mind, in our example, we are going to follow the same design structure.

So, what we are going to do is write a simple app in which we are going to modify the price of the product that will implement the Command design pattern.

That being said, let's start with the **`Product`** receiver class, which should contain the base business logic in our app:

```
public class Product
{
    public string Name { get; set; }
    public int Price { get; set; }

    public Product(string name, int price)
    {
        Name = name;
        Price = price;
    }
```

```
    public void IncreasePrice(int amount)
    {
        Price += amount;
        Console.WriteLine($"The price for the {Name} has been increased by {amount}$.");
    }
    public void DecreasePrice(int amount)
    {
        if (amount < Price)
        {
            Price -= amount;
            Console.WriteLine($"The price for the {Name} has been decreased by {amount}$.");
        }
    }

    public override string ToString() => $"Current price for the {Name} product is {Price}$.";
}
```

Now the Client class can instantiate the **Product** class and execute the required actions. But the Command design pattern states that we shouldn't use receiver classes directly. Instead, we should extract all the request details into a special class - Command. Let's do that.

The first thing we are going to do is to add the **ICommand** interface:

```
public interface ICommand
{
    void ExecuteAction();
}
```

Just to enumerate our price modification actions, we are going to add a simple **PriceAction** enumeration:

```
public enum PriceAction
{
    Increase,
    Decrease
}
```

Finally, let's add the **ProductCommand** class:

```
public class ProductCommand : ICommand
{
    private readonly Product _product;
    private readonly PriceAction _priceAction;
    private readonly int _amount;

    public ProductCommand(Product product, PriceAction priceAction, int amount)
    {
        _product = product;
        _priceAction = priceAction;
        _amount = amount;
    }
    public void ExecuteAction()
    {
        if (_priceAction == PriceAction.Increase)
        {
            _product.IncreasePrice(_amount);
        }
        else
        {
            _product.DecreasePrice(_amount);
        }
    }
}
```

As we can see, the **`ProductCommand`** class has all the information about the request and based on that executes required action.

To continue on, let's add the **`ModifyPrice`** class, which will act as Invoker:

```
public class ModifyPrice
{
    private readonly List<ICommand> _commands;
    private ICommand _command;

    public ModifyPrice()
    {
        _commands = new List<ICommand>();
    }

    public void SetCommand(ICommand command) => _command = command;

    public void Invoke()
    {
        _commands.Add(_command);
        _command.ExecuteAction();
    }
}
```

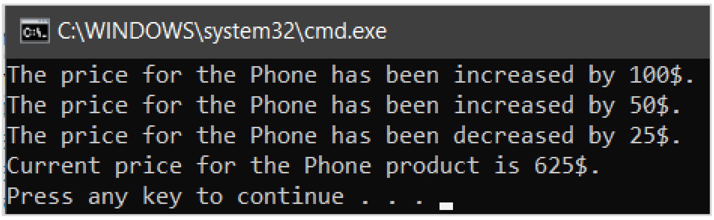
This class can work with any command that implements the **`ICommand`** interface and store all the operations as well.

Now, we can start working with the client part:

```
public static void Main()
{
    var modifyPrice = new ModifyPrice();
    var product = new Product("Phone", 500);

    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Increase, 100));

    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Increase, 50));

    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Decrease, 25));

    Console.WriteLine(product);
}
```

```
private static void Execute(Product product, ModifyPrice modifyPrice, ICommand productCommand)
{
    modifyPrice.SetCommand(productCommand);
    modifyPrice.Invoke();
}
```

The output should be like this:



```
C:\WINDOWS\system32\cmd.exe
The price for the Phone has been increased by 100$.
The price for the Phone has been increased by 50$.
The price for the Phone has been decreased by 25$.
Current price for the Phone product is 625$.
Press any key to continue . . .
```

4. Prototype

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

4.1 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.

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

4.2 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.

```
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}";
    }
}
```

4.3 Sandwich Menu

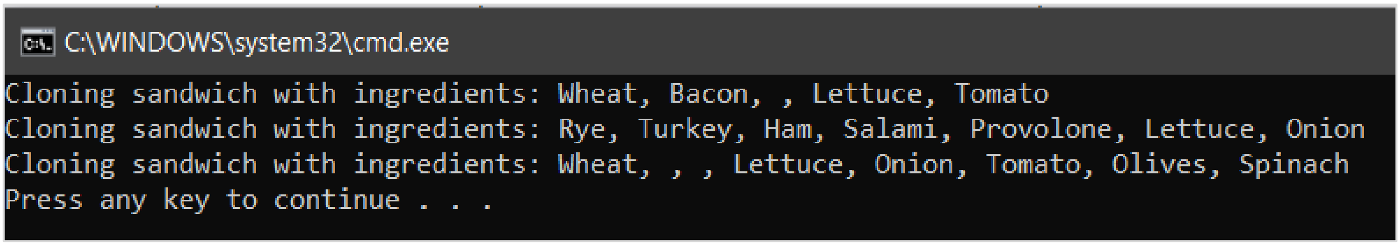
Let’s create a class that will have the purpose to store the sandwiches we’ve made. It will be like a repository.

```
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); }
    }
}
```

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`**.

```
static void Main(string[] args)
{
    SandwichMenu sandwichMenu = new SandwichMenu();

    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");

    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");
    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 . . .
```

5. 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.

5.1 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.

```
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();
}
```

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.

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

5.2 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**.

```
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;
    }
}
```

5.3 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.

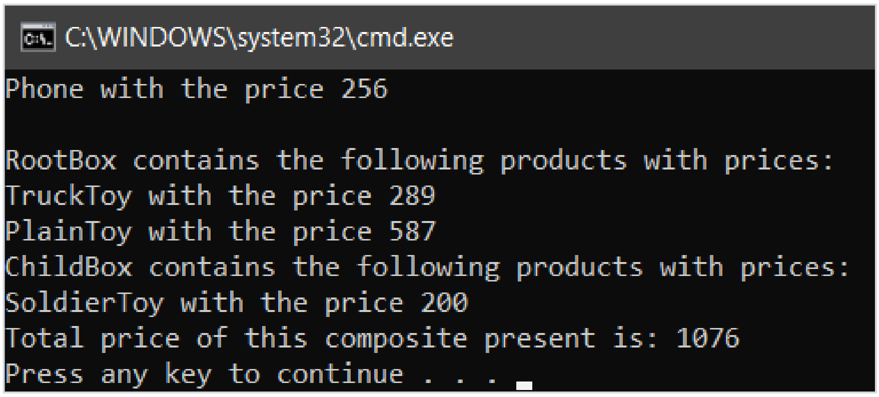
```csharp
public class SingleGift : GiftBase
{
    public SingleGift(string name, int price)
        : base(name, price)
    {
    }

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

        return price;
    }
}
```

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.

```csharp
static void Main(string[] args)
{
    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);
```

```csharp
    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 . . .
```

6. 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.

6.1 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()**.

```
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();
    }
}
```

6.2 Concrete Classes

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

```
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)");
    }
}
```

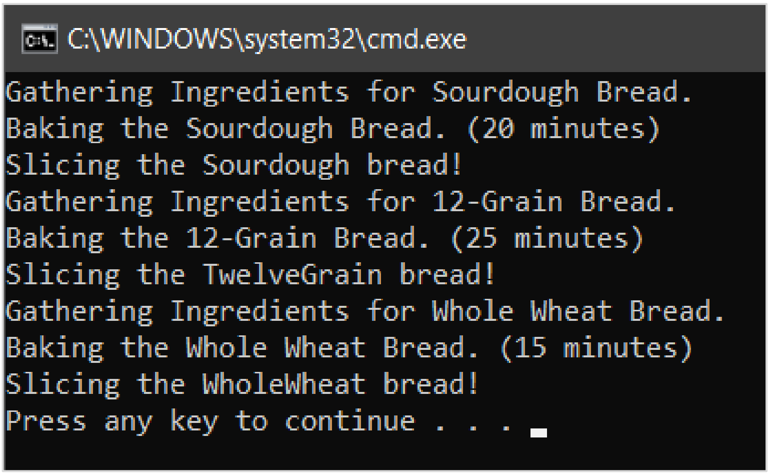
```
class Sourdough : Bread
{
    public override void MixIngredients()
    {
        Console.WriteLine("Gathering Ingredients for Sourdough Bread.");
    }

    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)");
    }
}
```

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.

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

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

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



```
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 . . .
```