**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.

## 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:

Graphical user interface, text, application

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

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.

A picture containing text

Description automatically generated

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:

Graphical user interface, text

Description automatically generated

The result in out console will be the following:

Text

Description automatically generated

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:

Text

Description automatically generated

The expected output should be something like this:

Text

Description automatically generated

## Facade

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

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

Graphical user interface, text, application

Description automatically generated

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:

Text

Description automatically generated

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:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

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:

Graphical user interface, text, application

Description automatically generated

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:

Graphical user interface, text, application

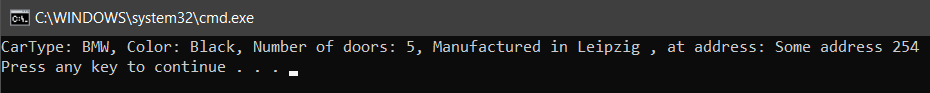
Description automatically generated

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

Text

Description automatically generated with medium confidence

And the output should be:



## 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:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, text

Description automatically generated

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

Text

Description automatically generated

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

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

The output should be like this:

Text

Description automatically generated

## Prototype

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

### 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.

Graphical user interface, text

Description automatically generated

### 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.

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

### 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.

Graphical user interface, text, application

Description automatically generated

#### 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**.

A picture containing text

Description automatically generated

Text

Description automatically generated with low confidence

Text

Description automatically generated

## 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.

### 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.

Graphical user interface, text, application

Description automatically generated

#### 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.

Graphical user interface, text

Description automatically generated

### 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**.

Graphical user interface, text, application, email

Description automatically generated

### 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.

Graphical user interface, text, application, email

Description automatically generated

#### 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.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

## Template Pattern

There are easily [hundreds of types of bread](https://en.wikipedia.org/wiki/List_of_breads) 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.

### 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()**.

Graphical user interface, text, application, email

Description automatically generated

### Concrete Classes

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

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

#### 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.

A screenshot of a computer

Description automatically generated with low confidence

Text

Description automatically generated