Design Pattern

1. Singleton Design Pattern

The Singleton is a creational design pattern which allows us to create a single instance of an object and to share that instance with all the users that require it. There is a common opinion that the Singleton pattern is not recommended because it presents a code smell, but there are some cases where it fits perfectly.

For example, some components have no reason to be instanced more than once in a project. Take a logger for example. It is quite common to register logger class as a singleton component because all we have to do is to provide a string to be logged and the logger is going to write it to the file. Then multiple classes may require to write in the same file at the same time from different threads, so having one centralized place for that purpose is always a good solution.

Or maybe sometimes we have a task to read some data from a file and use them through our project. If we know for sure that file won’t change while we read it, we can create a single instance of the object which will read that file and share it through the project to the consumer classes.

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.

So, let’s implement it.

To implement the Singleton pattern, let’s change the SingletonDataContainer class:

public class SingletonDataContainer: ISingletonContainer

{

    private Dictionary<string, int> \_capitals = new Dictionary<string, int>();

    private 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];

    }

    private static SingletonDataContainer instance = new SingletonDataContainer();

    public static SingletonDataContainer Instance => instance;

}

So, what we’ve done here is that we hid our 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.

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.

class Program

{

    static void Main(string[] args)

    {

        var db = SingletonDatabase.Instance;

        var db2 = SingletonDatabase.Instance;

        var db3 = SingletonDatabase.Instance;

        var db4 = SingletonDatabase.Instance;

    }

}

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

But our implementation is not ideal. Let’s construct our object the lazy way.

**Implementing a Thread-Safe Singleton**

Let’s modify our class to implement a thread-safe Singleton by using the Lazy type:

  private static Lazy<SingletonDataContainer> instance = new Lazy<SingletonDataContainer>(() => new SingletonDataContainer());

    public static SingletonDataContainer Instance => instance.Value;

Right now, our class is completely thread-safe. It is loaded in a lazy way which means that our instance is going to be created only when it is actually needed. We can even check if our object is created with the IsValueCreated property if we need to.

Now we can fully consume it in our consumer class:

class Program

{

    static void Main(string[] args)

    {

        var db = SingletonDatabase.Instance;

        Console.WriteLine(db.GetPopulation("Washington, D.C."));

        var db2 = SingletonDatabase.Instance;

        Console.WriteLine(db2.GetPopulation("London"));

    }

}

Factory Pattern

The Factory method is a creational design pattern which provides an interface for creating objects without specifying their concrete classes. It defines a method which we can use to create an object instead of using its constructor. The important thing is that the subclasses can override this method and create objects of different types.

To implement a Factory method pattern, we are going to create a simple Air conditioner application. Our app will receive an input from a user and based on that input will trigger a required action (cooling or warming the room). So let’s start with an interface

public interface Iweather

{

string getData(WeatherData \_weatherData);

}

Now, we need concrete classes to implement this interface:

public class clsForcast:Iweather

{

public string getData(WeatherData \_weatherData)

{

string key = WebConfigurationManager.AppSettings["ApiUserName"];

string strUrl = "https://api.darksky.net/forecast/" + key + "/"+\_weatherData.Lat + "," + \_weatherData.Log ;

string response = null;

try

{

MyWebRequest \_myWebRequest = new MyWebRequest(strUrl);

response = \_myWebRequest.GetResponse();

}

catch (Exception err)

{

Console.Write(err.Message.ToString());

}

return response;

}

}

Another class

public class clsTimeMachine : Iweather

{

//

public string getData(WeatherData \_weatherData)

{

long dt = MyWebRequest.ConvertToUnixTime(\_weatherData.DT);

string key = WebConfigurationManager.AppSettings["ApiUserName"];

string strUrl = "https://api.darksky.net/forecast/" + key + "/" + \_weatherData.Lat + "," + \_weatherData.Log + "," + dt;

string response = null;

try

{

MyWebRequest \_myWebRequest = new MyWebRequest(strUrl);

response = \_myWebRequest.GetResponse();

}

catch (Exception err)

{ }

return response;

}

}

Now let’s create a factory creator for these objects.

static public Iweather getData(WeatherData weatherDataType)

{

Iweather \_iweather;

if (weatherDataType.WeatherType == "TM")

{

\_iweather = \_clsTimeMachine;

}

else if (weatherDataType.WeatherType == "FC")

{

\_iweather = \_clsForcast;

}

else

{

return null;

}

return \_iweather;

}

Now we are ready to start using our Factory methods. In many examples, we can see the switch statement which switches through the user’s input and selects the required factory class.

Iweather \_iWeather;

WeatherData \_weatherData = new WeatherData();

\_weatherData.Lat = lat;//19.99;

\_weatherData.Log = log;// 73.78;

\_weatherData.DT = DateTime.Now;

\_weatherData.WeatherType = "TM";

\_iWeather = clsWeatherFactory.getData(\_weatherData);

We can use the Factory method to replace our constructor while creating an object. If our constructor consists of lots of code, we should replace it with the factory method. Furthermore, we can have multiple factory methods with meaningful names and parameter names as well which replace a single constructor.

Let’s imagine that we have functionality in which we convert the list of car manufacturers into JSON format and write it to the screen. But instead of a list, we have been provided with an API that provides us with all the manufacturers in the XML format.

Let’s say we can’t modify the existing API functionality (because of the technical restrictions such as being imported into our project from another solution that we mustn’t modify or as a NuGet package) so we have to find a way around it.

And the proper way to do it is to implement the Adapter pattern to solve this problem.

public class weatherDataResponse

{

private double lat;

public double Lat

{

get

{

return this.lat;

}

set

{

this.lat = value;

}

}

private double log;

public double Log

{

get

{

return this.log;

}

set

{

this.log = value;

}

}

private string timezone;

public string TimeZone

{

get

{

return this.timezone;

}

set

{

this.timezone = value;

}

}

private double apparenttemperature;

public double ApparentTemperature

{

get

{

return this.apparenttemperature;

}

set

{

this.apparenttemperature = value;

}

}

private double pressure;

public double Pressure

{

get

{

return this.pressure;

}

set

{

this.pressure = value;

}

}

private double cloudcover;

public double CloudCover

{

get

{

return this.cloudcover;

}

set

{

this.cloudcover = value;

}

}

private double dewpoint;

public double DewPoint

{

get

{

return this.dewpoint;

}

set

{

this.dewpoint = value;

}

}

private double humidity;

public double Humidity

{

get

{

return this.humidity;

}

set

{

this.humidity = value;

}

}

}

Get data from the API

public class clsForcast:Iweather

{

public string getData(WeatherData \_weatherData)

{

string key = WebConfigurationManager.AppSettings["ApiUserName"];

string strUrl = "https://api.darksky.net/forecast/" + key + "/"+\_weatherData.Lat + "," + \_weatherData.Log ;

string response = null;

try

{

MyWebRequest \_myWebRequest = new MyWebRequest(strUrl);

response = \_myWebRequest.GetResponse();

}

catch (Exception err)

{

Console.Write(err.Message.ToString());

}

return response;

}

}

public class XmlConverter

{

public XDocument GetXML()

{

WeatherData \_weatherData = new WeatherData();

\_weatherData.Lat = 19.99;

\_weatherData.Log = 73.78;

\_weatherData.DT = DateTime.Now;

\_weatherData.WeatherType = "TM";

var xDocument = new XDocument();

var xElement = new XElement("tblWeatherDataResponse");

var xAttributes = clsWeatherFactory.getData(\_weatherData);

Iweather \_iWeather;

\_iWeather = clsWeatherFactory.getData(\_weatherData);

var response = \_iWeather.getData(\_weatherData);

var root = JObject.Parse(response);

tblWeatherDataResponse \_weatherdataResponse=clsProcessData.dataResponse(root);

XmlSerializer xsSubmit = new XmlSerializer(typeof(tblWeatherDataResponse));

// var subReq = new MyObject();

var xml = "";

using (var sww = new StringWriter())

{

using (XmlWriter writer = XmlWriter.Create(sww))

{

xsSubmit.Serialize(writer, \_weatherdataResponse);

xml = sww.ToString(); // Your XML

}

}

xElement.Add(xAttributes);

xDocument.Add(xElement);

Console.WriteLine(xDocument);

return xDocument;

}

}

This is how the xDocument should look like:

<tblWeatherDataResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/ModelData">

<ApparentTemperature>74.91</ApparentTemperature>

<CloudCover i:nil="true"/>

<DewPoint i:nil="true"/>

<EntryDate i:nil="true"/>

<Humidity>0.92</Humidity>

<Icon>rain</Icon>

<IsActive i:nil="true"/>

<Lat>19.99</Lat>

<Long>73.78</Long>

<MoonPhase i:nil="true"/>

<Ozone i:nil="true"/>

<PrecipIntensity i:nil="true"/>

<PrecipProbability i:nil="true"/>

<Pressure>1002.3</Pressure>

<RequestTime>2019-07-26T17:26:44+05:30</RequestTime>

<Summary>Possible Light Rain</Summary>

<SunriseTime i:nil="true"/>

<SunseTime i:nil="true"/>

<Temperature>23.0611111111111</Temperature>

<TimeZone>Asia/Kolkata</TimeZone>

<UvIndex>1</UvIndex>

<tblDailies/>

<tblHourlies>...</tblHourlies>

<tblMinutelies/>

<weatherDataResponseId>0</weatherDataResponseId>

</tblWeatherDataResponse>

Now let’s implement a JsonConverter class:

public class JsonConverter

{

private XDocument \_weatherDataResponse;

public JsonConverter(XDocument weatherDataResponse)

{

\_weatherDataResponse = weatherDataResponse;

}

public void ConvertToJson()

{

var jsonweatherData = JsonConvert.SerializeObject(\_weatherDataResponse, Formatting.Indented);

Console.WriteLine("\nPrinting JSON list\n");

Console.WriteLine(jsonweatherData);

}

}

Of course, for the serialization to work we need to install the Newtonsoft.Json library, so don’t forget to do that.

**Adapter Implementation**

As we can see, there is no way to pass an xDocument to the JsonConverter class and there shouldn’t be one, so we need to create the adapter class which will make these two interfaces work together.

To do this, we are going to start with the IXmlToJson interface to define the behavior of our adapter class:

public interface IXmlToJson

{

void ConvertXmlToJson();

}

Then, let’s continue with the XmlToJsonAdapter class which is going to implement the IXmlToJsoninterface:

public class XmlToJsonAdapter : IXmlToJson

{

private readonly XmlConverter \_xmlConverter;

public XmlToJsonAdapter(XmlConverter xmlConverter)

{

\_xmlConverter = xmlConverter;

}

public void ConvertXmlToJson()

{

XDocument weatherData = \_xmlConverter.GetXML();

new JsonConverter(weatherData).ConvertToJson();

}

}

We have created our adapter class which converts the Xml document object into the list of manufacturers and provides that list to the JsonConverter class.

So, as you can see, we have enabled collaboration between two completely different interfaces by just introducing an adapter class to our project.

Now, we can make a call to this adapter class from our client class:

var xmlConverter = new XmlConverter();

var adapter = new XmlToJsonAdapter(xmlConverter);

adapter.ConvertXmlToJson();

Result will be as follows:

{

"ApparentTemperature": "74.91",

"CloudCover": {

"@nil": "true"

},

"DewPoint": {

"@nil": "true"

},

"EntryDate": {

"@nil": "true"

},

"Humidity": "0.92",

"Icon": "rain",

"IsActive": {

"@nil": "true"

},

"Lat": "19.99",

"Long": "73.78",

"MoonPhase": {

"@nil": "true"

},

"Ozone": {

"@nil": "true"

},

"PrecipIntensity": {

"@nil": "true"

},

"PrecipProbability": {

"@nil": "true"

},

"Pressure": "1002.3",

"RequestTime": "2019-07-26T17:26:44+05:30",

"Summary": "Possible Light Rain",

"SunriseTime": {

"@nil": "true"

},

"SunseTime": {

"@nil": "true"

},

"Temperature": "23.0611111111111",

"TimeZone": "Asia/Kolkata",

"UvIndex": "1",

"tblDailies": null,

"tblHourlies": "...",

"tblMinutelies": null,

"weatherDataResponseId": "0"

}