**Design Patterns in C#**

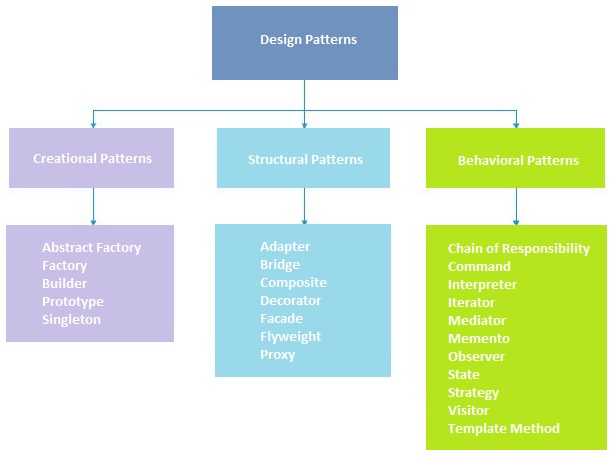
Design patterns provide general solutions or a flexible way to solve common design problems.

Design Patterns are nothing but, you can say, documented and tested solutions for recurring problems in a given context. So, in simple words, we can say that Design Patterns are reusable solutions to the problems that, as a developer, we encounter in our day-to-day programming. Design Patterns are used to solve the problems of Object Generation and Integration.

**Types of Design Patterns**

Gang of Four (GOF) categorized the Design Pattern into three main categories based on the three problem areas (**Object Creation and Initialization, Structural Changes of Classes and Interfaces, and the Relationship Between Classes and communication Between Objects**) of software architecture.

1. [Creational Design Pattern](https://dotnettutorials.net/lesson/creational-design-pattern/) (Object Creation and Initialization)
2. [Structural Design Pattern](https://dotnettutorials.net/lesson/structural-design-pattern/) (Structural Changes of Classes, and Interfaces, and the Relationship Between Classes)
3. [Behavioral Design Pattern](https://dotnettutorials.net/lesson/behavioral-design-pattern/) (Communication Between Objects)



**Creational Design Patterns:**

The **Creational Design Pattern** deals with **Object Creation and Initialization**. The Creational Design Pattern gives the programmer more flexibility in deciding which objects need to be created for a given case/ situation.

For example, if we have a huge project, a huge project means we have a lot of classes, and a lot of classes means we are dealing with many objects. So we need to create different objects (like **new Customer(), new Product(), new Invoice()**, etc.) based on some conditions.

**Structural Design Patterns:**

The **Structural Design Pattern** is used to Manage the Structure of Classes and Interfaces and**the Relationship Between the Classes and Interfaces**.

For example, if we have a Customer and Product class and the Product class is used inside the Customer class, making One-to-Many relationships. As the project proceeds tomorrow, we want to keep the product class from the Customer class as we want to use the Product and Customer classes independently. This is a structural change, and we don’t want this structural change to affect our project. This is where the Structural Design Pattern helps us.

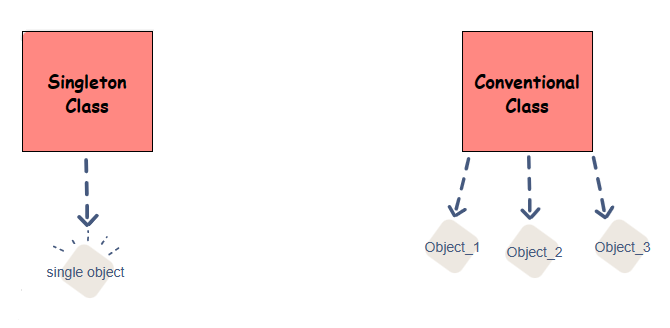
**Behavioral Design Patterns:**

**Behavioral Design Patterns** deal with the **Communication Between Classes and Objects**. That means if you want to change the behavior of a class again, you want it to affect other classes of the project as well.

For example, you have an Invoice class that currently applies taxes as 18%. Tomorrow, if you have to add another extra tax. That means you are changing the behavior of a class. To solve such Behavioral issues, Behavioral Design patterns come into the picture.

**What is Singleton Pattern in C#?**

The Singleton design pattern that restricts the instantiation of a class to one object and provides a way to access its object. This is useful when exactly one object is needed to coordinate actions across the system. That means we need to use the Singleton Design Pattern in C# to ensure that only one instance of a particular class will be created and provide simple global access to that instance for the entire application.



##### **Implementation Guidelines of Singleton Design Pattern in C#**

1. We need to declare a constructor that should be **private** and **parameterless**. This is required because it will restrict the class from being instantiated from outside the class. It only instantiates from within the class.
2. The class should be declared sealed, ensuring it cannot be inherited. This is going to be useful when you are dealing with the nested class.
3. We must create a **private static variable** referencing the class’s singleton instance.
4. We also need to create a **public static property/method** that will return the singleton instance of the class. This method or property first checks whether an instance of the singleton class is created. If the singleton instance is created, it returns that instance; otherwise, it will create an instance and then return it.

##### **Real-Time Example to Understand Singleton Design Pattern in C#**

**ILog.cs**

namespace SingletoninMVC.Logger

{

public interface ILog

{

void LogException(string message);

}

}

**Log.cs**

using System;

using System.IO;

using System.Text;

namespace SingletoninMVC.Logger

{

public sealed class Log : ILog

{

//Private Constructor to Restrict Class Instantiation from outside the Log class

private Log()

{

}

//Creating Log Instance using Eager Loading

private static readonly Log LogInstance = new Log();

//Returning the Singleton LogInstance

//This Method is Thread Safe as it uses Eager Loading

public static Log GetInstance()

{

return LogInstance;

}

//This Method Log the Exception Details in a Log File

public void LogException(string message)

{

//Create the Dynamic File Name

string fileName = $"Exception\_{DateTime.Now.ToShortDateString()}.log";

//Create the Path where you want to Create the Log file

string logFilePath = $"{AppDomain.CurrentDomain.BaseDirectory}\\{fileName}";

//Build the String Object using StringBuilder for a Better Performance

StringBuilder sb = new StringBuilder();

sb.AppendLine("----------------------------------------");

sb.AppendLine(DateTime.Now.ToString());

sb.AppendLine(message);

//Write the StringBuilder Message into the Log File Path using StreamWriter Object

using (StreamWriter writer = new StreamWriter(logFilePath, true))

{

writer.Write(sb.ToString());

writer.Flush();

}

}

}

}

**EmployeeController**

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.Entity;

using System.Linq;

using System.Net;

using System.Web;

using System.Web.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

using Microsoft.AspNetCore.Mvc;

using SingletoninMVC.Logger;

using SingletoninMVC.Models;

namespace SingletoninMVC.Controllers

{

public class EmployeeController : Controller

{

private ILog \_ILog;

private EmployeeDBContext db = new EmployeeDBContext();

public EmployeeController()

{

//Get the Singleton Log Instance

\_ILog = Log.GetInstance();

}

//Whenever Any Exception Occurred, the following OnException Method will Execute

protected override void OnException(ExceptionContext filterContext)

{

//First, Log the Exception Details

\_ILog.LogException(filterContext.Exception.ToString());

//Then set that the Exception is Handled

filterContext.ExceptionHandled = true;

//Then Redirect to the Error view

this.View("Error").ExecuteResult(this.ControllerContext);

}

// GET: Employee

public ActionResult Index()

{

return View(db.Employees.ToList());

}

}

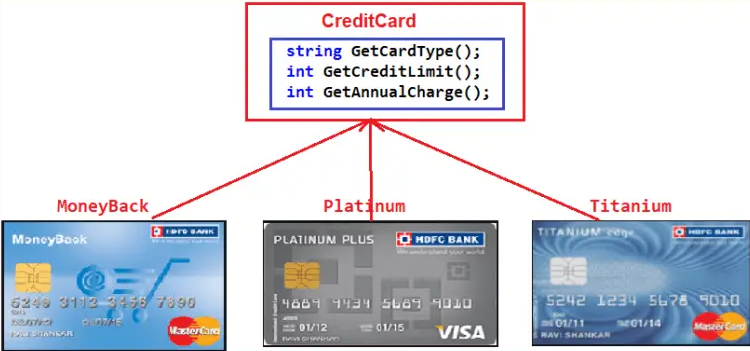
}

##### **What is Factory Design Pattern in C#?**

According to Gang of Four (GoF), “**A factory is an object used for creating other objects. In technical terms, we can say that a factory is a class with a method. That method will create and return different objects based on the received input parameter**“.

##### **Real-Time Example to Understand Factory Design Pattern in C#**

We have three credit card classes, i.e., MoneyBack, Titanium, and Platinum. These three classes are the subclasses of the CreditCard superclass or, you can say, super interface. The CreditCard superclass or super interface has three methods, i.e., GetCardType, GetCreditLimit, and GetAnnualCharge. The subclasses, i.e., MoneyBack, Titanium, and Platinum, have implemented the above three methods of the CreditCard.



#### **Example Without using Factory Design Pattern in C#**

##### **Create Interface ICreditCard.cs**

namespace FactoryDesignPattern

{

public interface ICreditCard

{

string GetCardType();

int GetCreditLimit();

int GetAnnualCharge();

}

}

##### **Creating Product Classes (MoneyBack, Titanium, and Platinum)**

**MoneyBack.cs**

namespace FactoryDesignPattern

{

class MoneyBack : ICreditCard

{

public string GetCardType()

{

return "MoneyBack";

}

public int GetCreditLimit()

{

return 15000;

}

public int GetAnnualCharge()

{

return 500;

}

}

}

**Titanium.cs:**

namespace FactoryDesignPattern

{

public class Titanium : ICreditCard

{

public string GetCardType()

{

return "Titanium Edge";

}

public int GetCreditLimit()

{

return 25000;

}

public int GetAnnualCharge()

{

return 1500;

}

}

}

**Platinum.cs**

namespace FactoryDesignPattern

{

public class Platinum : ICreditCard

{

public string GetCardType()

{

return "Platinum Plus";

}

public int GetCreditLimit()

{

return 35000;

}

public int GetAnnualCharge()

{

return 2000;

}

}

}

##### **Client Code (Main Method)**

using System;

namespace FactoryDesignPattern

{

class Program

{

static void Main(string[] args)

{

//Generally we will get the Card Type from UI.

//Here we are hardcoded the card type

string cardType = "MoneyBack";

ICreditCard cardDetails = null;

//Based of the CreditCard Type we are creating the appropriate type instance using if else condition

if (cardType == "MoneyBack")

{

cardDetails = new MoneyBack();

}

else if (cardType == "Titanium")

{

cardDetails = new Titanium();

}

else if (cardType == "Platinum")

{

cardDetails = new Platinum();

}

if (cardDetails != null)

{

Console.WriteLine("CardType : " + cardDetails.GetCardType());

Console.WriteLine("CreditLimit : " + cardDetails.GetCreditLimit());

Console.WriteLine("AnnualCharge :" + cardDetails.GetAnnualCharge());

}

else

{

Console.Write("Invalid Card Type");

}

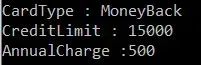
Console.ReadLine();

}

}

}

The above code implementation is very straightforward. Once we get the CardType value, we create the appropriate Credit Card instance using the IF-ELSE Condition. Then, we call the three methods to display the credit card information on the console window. So, when you run the application, you will get the output as expected, as shown below.



##### **What is the Problem with the above Code Implementation?**

The above code implementation introduces the following problems

1. First, the **Tight Coupling** between the client class (Program) and Product Classes (MoneyBack, Titanium, and Platinum). So, when we make changes in one class, we must also make changes in the other classes.
2. Secondly, suppose we add a new Credit Card. In that case, we also need to modify the client code, i.e., the main method of the Program class, by adding an extra **IF-ELSE Condition,** which not only overheads the development but also the testing process.

##### **Factory Design Pattern Implementation in C#**

As per the definition of Factory Design Pattern, the Factory Design Pattern creates an object without exposing the object creation logic to the client, and the client refers to the newly created object using a common interface.

Please have a look at the following image. Our factory class is responsible for creating and returning the appropriate Product (i.e., MoneyBack, Titanium, and Platinum) object. As you can see, this class has one static method, i.e., GetCreditcard, and this method takes one input parameter and, based on the parameter value it will create one of the credit card (i.e., MoneyBack, Platinum, and Titanium) objects and store that object in the superclass (CrditCard) reference variable and finally return that superclass reference variable to the caller of this method i.e. to the client or you can say in our example it is the Main method of the Program class.

##### **Creating Factory Class CreditCardfactory.cs**

using System;

namespace FactoryDesignPattern

{

public class CreditCardFactory

{

public static ICreditCard GetCreditCard(string cardType)

{

ICreditCard cardDetails = null;

if (cardType == "MoneyBack")

{

cardDetails = new MoneyBack();

}

else if (cardType == "Titanium")

{

cardDetails = new Titanium();

}

else if (cardType == "Platinum")

{

cardDetails = new Platinum();

}

return cardDetails;

}

}

}

##### **Client Code (Main Method)**

using System;

namespace FactoryDesignPattern

{

class Program

{

static void Main(string[] args)

{

ICreditCard cardDetails = CreditCardFactory.GetCreditCard("Platinum");

if (cardDetails != null)

{

Console.WriteLine("CardType : " + cardDetails.GetCardType());

Console.WriteLine("CreditLimit : " + cardDetails.GetCreditLimit());

Console.WriteLine("AnnualCharge :" + cardDetails.GetAnnualCharge());

}

else

{

Console.Write("Invalid Card Type");

}

Console.ReadLine();

}

}

}

