**SOLID Design Principles in C#**

The SOLID Design Principles in C# are the design principles that help us solve most software design problems. These design principles provide multiple ways to remove the tightly coupled code between the software components (between classes), making the software designs more understandable, flexible, and maintainable.

##### **Why Do We Need to Learn SOLID Design Principles?**

##### As a developer, we start developing applications using our experience and knowledge. But over time, the applications might cause bugs. We must alter the application design for every change request or new feature request. After some time, we might need to put in a lot of effort, even for simple tasks, which might require the full working knowledge of the entire system. But we can’t blame the change requests or new feature requests as they are part of the software development. We can’t stop them, and we can’t refuse them either. So who is the culprit here? Obviously, it is the Design of the Application.

##### **What are the Main Reasons Behind Most Unsuccessful Applications?**

1. Putting More Functionalities on Classes. (In simple words, we put many functionalities into the class even though they are unrelated to that class.)
2. Implementing Tight Coupling Between the Software Components (i.e., Between the Classes). If the classes depend on each other, changing one class will also affect the other.

##### **How to Overcome the Unsuccessful Application Development Problems?**

1. We need to use the Correct Architecture (i.e., MVC, Layered, 3-tier, MVP, and so on) as per the Project Requirements.
2. As developers, we must follow the Design Principles (i.e., SOLID Principles, ONIO Design Principles, etc.).
3. Again, we must choose the correct Design Patterns (Creational Design Pattern, Structural Design Pattern, Behavioral Design Pattern, Dependency Injection Design Pattern, Repository Design Pattern, etc.) per the project requirements.

##### **What are SOLID Design Principles?**

##### The **SOLID Design Principles**are those used to manage most of the Software Design Problems we, as developers, generally encounter in our day-to-day programming. These design principles are tested and proven mechanisms to make the software designs more understandable, flexible, and maintainable. As a result, if we follow these principles while designing our application, we can develop better applications.

SOLID Design Principles represent five Design Principles used to make software designs more understandable, flexible, and maintainable. The Five SOLID Design Principles are as follows:

* **S**stands for the [**Single Responsibility Principle**](https://dotnettutorials.net/lesson/single-responsibility-principle/), also known as **SRP**: The Single Responsibility Principle states that Each software module or class should have only one reason to change. In other words, we can say that each module or class should have only one responsibility.
* **O** stands for the [**Open-Closed Principle**](https://dotnettutorials.net/lesson/open-closed-principle/), also known as **OCP**: The Open-Closed Principle states that software entities such as modules, classes, functions, etc., should be open for extension but closed for modification.
* **L**stands for the [**Liskov Substitution Principle**](https://dotnettutorials.net/lesson/liskov-substitution-principle/), also known as LSP. The Liskov Substitution Principle states that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class. That means the child class objects should be able to replace parent class objects without changing the correctness or behavior of the program.
* **I**stand for the [**Interface Segregation Principle**](https://dotnettutorials.net/lesson/interface-segregation-principle/)**, also** known as ISP: The Interface Segregation Principle states that Clients should not be forced to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods, with each interface serving one submodule.
* **D**stands for [**Dependency Inversion Principle**](https://dotnettutorials.net/lesson/dependency-inversion-principle/), also known as DIP: The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

**Single Responsibility Principle Real Time Example**

Without SRP

namespace SRPExample

{

public class BankAccount

{

public int AccountNumber { get; set; }

public double Balance { get; set; }

private List<string> Transactions = new List<string>();

public BankAccount(int accountNumber)

{

AccountNumber = accountNumber;

}

public void Deposit(double amount)

{

Balance += amount;

Transactions.Add($"Deposited Rs.{amount}. New Balance: Rs.{Balance}");

}

public void Withdraw(double amount)

{

Balance -= amount;

Transactions.Add($"Withdrew Rs.{amount}. New Balance: Rs.{Balance}");

}

public void PrintStatement()

{

Console.WriteLine($"Statement for Account: {AccountNumber}");

foreach (var transaction in Transactions)

{

Console.WriteLine(transaction);

}

}

}

//Testing the Single Responsibility Principle

public class Program

{

public static void Main()

{

BankAccount johnsAccount = new BankAccount(123456);

johnsAccount.Deposit(500);

johnsAccount.Withdraw(100);

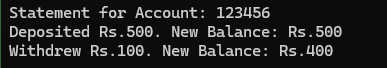
johnsAccount.PrintStatement();

Console.ReadKey();

}

}

}

****

Here, the BankAccount class handles:

* Transaction operations.
* Printing the transaction statement.

###### **Following SRP:**

* A cleaner approach would separate transaction management from statement printing:
* BankAccount manages the transactions of the account.
* StatementPrinter handles printing the transaction statement.

Let us see how we can implement the above example following the Single Responsibility Principle in C#:

namespace SRPExample

{

public class BankAccount

{

public int AccountNumber { get; private set; }

public double Balance { get; private set; }

public List<string> Transactions = new List<string>();

public BankAccount(int accountNumber)

{

AccountNumber = accountNumber;

}

public void Deposit(double amount)

{

Balance += amount;

Transactions.Add($"Deposited Rs.{amount}. New Balance: Rs.{Balance}");

}

public void Withdraw(double amount)

{

Balance -= amount;

Transactions.Add($"Withdrew Rs.{amount}. New Balance: Rs.{Balance}");

}

}

public class StatementPrinter

{

public void Print(BankAccount account)

{

Console.WriteLine($"Statement for Account: {account.AccountNumber}");

foreach (var transaction in account.Transactions)

{

Console.WriteLine(transaction);

}

}

}

//Testing the Single Responsibility Principle

public class Program

{

public static void Main()

{

BankAccount johnsAccount = new BankAccount(123456);

johnsAccount.Deposit(500);

johnsAccount.Withdraw(100);

StatementPrinter printer = new StatementPrinter();

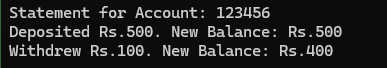
printer.Print(johnsAccount);

Console.ReadKey();

}

}

}

****

**Open Closed Principle Real Time Example**

Without OCP

namespace OCPDemo

{

public enum CustomerType

{

Regular,

Premium,

Newbie

}

public class DiscountCalculator

{

public double CalculateDiscount(double price, CustomerType customerType)

{

switch (customerType)

{

case CustomerType.Regular:

return price \* 0.1; // 10% discount for regular customers

case CustomerType.Premium:

return price \* 0.3; // 30% discount for premium customers

case CustomerType.Newbie:

return price \* 0.05; // 5% discount for new customers

default:

throw new ArgumentOutOfRangeException();

}

}

}

public class Program

{

public static void Main()

{

double discountedPrice = 0;

DiscountCalculator objdiscountCalculator = new DiscountCalculator();

discountedPrice = objdiscountCalculator.CalculateDiscount(100, CustomerType.Regular);

Console.WriteLine($"Discounted Price For {CustomerType.Regular} is {discountedPrice}");

discountedPrice = objdiscountCalculator.CalculateDiscount(100, CustomerType.Premium);

Console.WriteLine($"Discounted Price For {CustomerType.Premium} is {discountedPrice}");

discountedPrice = objdiscountCalculator.CalculateDiscount(100, CustomerType.Newbie);

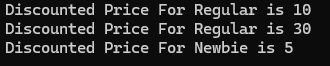
Console.WriteLine($"Discounted Price For {CustomerType.Newbie} is {discountedPrice}");

Console.ReadKey();

}

}

}



We can define a strategy pattern to adhere to open closed principle. Each discount type will have its own class, and adding a new discount would mean adding a new class without modifying the existing ones. Let us see how we can implement the above example following the Open-Closed Principle in C#:

namespace OCPDemo

{

//Create an interface for the discount strategy

public interface IDiscountStrategy

{

double CalculateDiscount(double price);

}

//Implement this interface for each discount type

public class RegularDiscount : IDiscountStrategy

{

public double CalculateDiscount(double price)

{

return price \* 0.1;

}

}

public class PremiumDiscount : IDiscountStrategy

{

public double CalculateDiscount(double price)

{

return price \* 0.3;

}

}

public class NewbieDiscount : IDiscountStrategy

{

public double CalculateDiscount(double price)

{

return price \* 0.05;

}

}

//Modify the DiscountCalculator class to accept an IDiscountStrategy

public class DiscountCalculator

{

private readonly IDiscountStrategy \_discountStrategy;

public DiscountCalculator(IDiscountStrategy discountStrategy)

{

\_discountStrategy = discountStrategy;

}

public double CalculateDiscount(double price)

{

return \_discountStrategy.CalculateDiscount(price);

}

}

//Testing the Open-Closed Principle

public class Program

{

public static void Main()

{

var regularDiscount = new RegularDiscount();

var calculator = new DiscountCalculator(regularDiscount);

double discountedPrice = calculator.CalculateDiscount(100); // 10% discount applied

var premiumDiscount = new PremiumDiscount();

calculator = new DiscountCalculator(premiumDiscount);

discountedPrice = calculator.CalculateDiscount(100); // 30% discount applied

Console.ReadKey();

}

}

}

**Liskov Substitution Principle Real Time Example**

Let’s see a real-time example, Vehicles and Their Engines, to understand the Liskov Substitution Principle (LSP). Vehicles can have different types of engines: gasoline, electric, or hybrid. Each engine type has a method to start it. However, while gasoline engines use ignition and fuel, electric engines need a battery check.

Without LSP

using System;

namespace LSPDemo

{

public class Vehicle

{

public virtual void StartEngine()

{

Console.WriteLine("Starting engine using ignition and fuel.");

}

}

public class ElectricVehicle : Vehicle

{

public override void StartEngine()

{

Console.WriteLine("Checking battery and starting electric motor.");

}

}

public class Program

{

public static void Main()

{

Vehicle vehicle = new Vehicle();

vehicle.StartEngine();

vehicle = new ElectricVehicle();

vehicle.StartEngine();

Console.ReadKey();

}

}

}

When working with the base Vehicle class, it’s important to note that the method StartEngine assumes an ignition mechanism. However, ElectricVehicle overrides this method to provide its own mechanism. If the client assumes that all vehicles start using an ignition mechanism, replacing the Vehicle with an ElectricVehicle would be incorrect and violate the Liskov Substitution Principle (LSP).

using System;

namespace LSPDemo

{

public abstract class Vehicle

{

public abstract void StartEngine();

}

public class GasolineVehicle : Vehicle

{

public override void StartEngine()

{

Console.WriteLine("Starting engine using ignition and fuel.");

}

}

public class ElectricVehicle : Vehicle

{

public override void StartEngine()

{

Console.WriteLine("Checking battery and starting electric motor.");

}

}

//Testing the Liskov Substitution Principle

public class Program

{

public static void Main()

{

Vehicle vehicle = new GasolineVehicle();

vehicle.StartEngine();

vehicle = new ElectricVehicle();

vehicle.StartEngine();

Console.ReadKey();

}

}

}

**Interface Segregation Principle Real Time Example**

###### **Violating ISP**

Creating a universal interface for all order types leads to unnecessary implementations. Let us first see how we can implement the above example without following the Interface Segregation Principle (ISP) in C#. Here’s an approach where a single interface tries to encompass all functionalities

namespace ISPDemo

{

public interface IOrder

{

void ProcessOnlinePayment();

void PrintTicket();

void ConfirmOverPhone();

}

public class OnlineOrder : IOrder

{

public void ProcessOnlinePayment()

{

Console.WriteLine("Implementation for processing online payment.");

}

public void PrintTicket()

{

throw new NotImplementedException("Online orders do not print tickets.");

}

public void ConfirmOverPhone()

{

throw new NotImplementedException("Online orders do not confirm over the phone.");

}

}

//Testing the ISP

public class Program

{

public static void Main()

{

OnlineOrder onlineOrder = new OnlineOrder();

onlineOrder.ProcessOnlinePayment();

onlineOrder.PrintTicket();

onlineOrder.ConfirmOverPhone();

Console.ReadKey();

}

}

}



Let us see how we can rewrite the above example following the Interface Segregation Principle (ISP) in C#. By segregating the interfaces, we can make the system more modular and avoid unnecessary implementations.

namespace ISPDemo

{

public interface IOnlineOrder

{

void ProcessOnlinePayment();

void GenerateReceipt();

}

public interface IInHouseOrder

{

void PrintTicket();

}

public interface IPhoneOrder

{

void ConfirmOverPhone();

}

// Implementing segregated interfaces

public class OnlineOrder : IOnlineOrder

{

public void ProcessOnlinePayment()

{

// Implementation for processing online payment.

Console.WriteLine("Processing Online Payment");

}

public void GenerateReceipt()

{

// Implementation for generating a receipt.

Console.WriteLine("Generating Receipt");

}

}

public class InHouseOrder : IInHouseOrder

{

public void PrintTicket()

{

// Implementation for printing a ticket.

Console.WriteLine("Printing Ticket");

}

}

public class PhoneOrder : IPhoneOrder

{

public void ConfirmOverPhone()

{

// Implementation to confirm order over the phone.

// Implementation for printing a ticket.

Console.WriteLine("Confirming order over the phone");

}

}

//Testing the Interface Segregation Principle

public class Program

{

public static void Main()

{

Console.WriteLine("OnlineOrder:");

OnlineOrder onlineOrder = new OnlineOrder();

onlineOrder.ProcessOnlinePayment();

onlineOrder.GenerateReceipt();

Console.WriteLine("\nInHouseOrder:");

InHouseOrder inHouseOrder = new InHouseOrder();

inHouseOrder.PrintTicket();

Console.WriteLine("\nPhoneOrder:");

PhoneOrder phoneOrder = new PhoneOrder();

phoneOrder.ConfirmOverPhone();

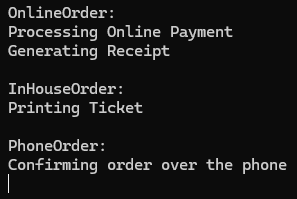
Console.ReadKey();

}

}

}

With this design approach, each class only implements the interfaces relevant to its behavior, adhering to the Interface Segregation Principle. This helps in maintaining a cleaner and more intuitive codebase.



**Example to Understand Dependency Inversion Principle in C#**

Without DIP

##### **Employee.cs**

Create a class file named **Employee.cs** and copy and paste the following code. The following is a simple class having 4 properties. The following class is going to hold the employee data.

namespace SOLID\_PRINCIPLES.DIP

{

public class Employee

{

public int ID { get; set; }

public string Name { get; set; }

public string Department { get; set; }

public int Salary { get; set; }

}

}

##### **EmployeeDataAccessLogic.cs**

namespace SOLID\_PRINCIPLES.DIP

{

public class EmployeeDataAccessLogic

{

public Employee GetEmployeeDetails(int id)

{

//In real time get the employee details from database

//but here we have hard coded the employee details

Employee emp = new Employee()

{

ID = id,

Name = "Pranaya",

Department = "IT",

Salary = 10000

};

return emp;

}

}

}

##### **DataAccessFactory.cs**

The following class contains one static method, returning an instance of the EmployeeDataAccessLogic class. If you want to consume any method of the EmployeeDataAccessLogic class, then you need to create an instance of that class. In our example, the following class, the GetEmployeeDataAccessObj() static method, is going to return an instance of the EmployeeDataAccessLogic class, and using that instance, we can access the GetEmployeeDetails(int id) method. So, this class will return an instance of the EmployeeDataAccessLogic class, using which we can do the database operations.

namespace SOLID\_PRINCIPLES.DIP

{

public class DataAccessFactory

{

public static EmployeeDataAccessLogic GetEmployeeDataAccessObj()

{

return new EmployeeDataAccessLogic();

}

}

}

##### **EmployeeBusinessLogic.cs**

The following class has one constructor used to create an instance of the EmployeeDataAccessLogic class. Here, within the constructor, we call the static GetEmployeeDataAccessObj() method on the DataAccessFactory class, which will return an instance of EmployeeDataAccessLogic, and we initialize the \_EmployeeDataAccessLogic property with the return instance. We also have one method, i.e., GetEmployeeDetails, which calls the GetEmployeeDetails method on the EmployeeDataAccessLogic instance to get the employee details by employee ID.

namespace SOLID\_PRINCIPLES.DIP

{

public class EmployeeBusinessLogic

{

EmployeeDataAccessLogic \_EmployeeDataAccessLogic;

public EmployeeBusinessLogic()

{

\_EmployeeDataAccessLogic = DataAccessFactory.GetEmployeeDataAccessObj();

}

public Employee GetEmployeeDetails(int id)

{

return \_EmployeeDataAccessLogic.GetEmployeeDetails(id);

}

}

}

Program.CS

using System;

namespace SOLID\_PRINCIPLES.DIP

{

public class Program

{

static void Main(string[] args)

{

EmployeeBusinessLogic employeeBusinessLogic = new EmployeeBusinessLogic();

Employee emp = employeeBusinessLogic.GetEmployeeDetails(1001);

Console.WriteLine($"ID: {emp.ID}, Name: {emp.Name}, Department: {emp.Department}, Salary: {emp.Salary}");

Console.ReadKey();

}

}

}



##### **Comparing the above Example with the Dependency Inversion Principle in C#**

As per the **Dependency Inversion Principle**definition**, “a High-Level module should not depend on Low-Level modules. Both should depend on the abstraction”.**

So, first, we need to figure out the High-Level Module (class) and the Low-Level Module (class) in our example. A High-Level Module is a module that always depends on other modules. So, in our example, the EmployeeBusinessLogic class depends on the EmployeeDataAccessLogic class, so here, the EmployeeBusinessLogic class is the high-level module, and the EmployeeDataAccessLogic class is the low-level module.

So, as per the first rule of the Dependency Inversion Principle in C#, the EmployeeBusinessLogic class/module should not depend on the concrete EmployeeDataAccessLogic class/module. Instead, both classes should depend on abstraction. But, in our example, the way we have implemented the code, the EmployeeBusinessLogic, depending on the EmployeeDataAccessLogic class, means the first rule we are not following. In the later part of this article, I will modify the example to follow the Dependency Inversion Principle.

The second rule of the **Dependency Inversion Principle**states that **“Abstractions should not depend on details. Details should depend on Abstractions”.**Before understanding this, let us first understand what is an abstraction.

With DIP

##### **Employee.cs**

Create a class file named **Employee.cs** and copy and paste the following code. The following is a simple class having 4 properties. The following class is going to hold the employee data.

namespace SOLID\_PRINCIPLES.DIP

{

public class Employee

{

public int ID { get; set; }

public string Name { get; set; }

public string Department { get; set; }

public int Salary { get; set; }

}

}

##### **IEmployeeDataAccessLogic.cs**

we created the interface with one abstract method, i.e., GetEmployeeDetails. You must declare those methods here if you have multiple employee-related methods.

namespace SOLID\_PRINCIPLES.DIP

{

public interface IEmployeeDataAccessLogic

{

Employee GetEmployeeDetails(int id);

//Any Other Employee Related Method Declarations

}

}

##### **EmployeeDataAccessLogic.cs**

namespace SOLID\_PRINCIPLES.DIP

{

public class EmployeeDataAccessLogic : IEmployeeDataAccessLogic

{

public Employee GetEmployeeDetails(int id)

{

//In real time get the employee details from database

//but here we have hard coded the employee details

Employee emp = new Employee()

{

ID = id,

Name = "Pranaya",

Department = "IT",

Salary = 10000

};

return emp;

}

}

}

Next, we need to change the DataAccessFactory class. Here, we need to change the return type of the GetEmployeeDataAccessObj to IEmployeeDataAccessLogic instead of EmployeeDataAccessLogic. Internally, the method creates an instance of the EmployeeDataAccessLogic class, but we return that instance to the user using the Parent Interface, i.e., IEmployeeDataAccessLogic. This is possible because a Parent Class Reference Variable can hold the child class object reference. And here, IEmployeeDataAccessLogic is the Parent class, and EmployeeDataAccessLogic is the Child class of the IEmployeeDataAccessLogic Parent class.

##### **DataAccessFactory.cs**

namespace SOLID\_PRINCIPLES.DIP

{

public class DataAccessFactory

{

public static IEmployeeDataAccessLogic GetEmployeeDataAccessObj()

{

return new EmployeeDataAccessLogic();

}

}

}

We need to change the EmployeeBusinessLogic class, which will use the IEmployeeDataAccessLogic instead of the concrete EmployeeDataAccessLogic class, as shown below. You can see the EmployeeBusinessLogic class is not using the concrete EmployeeDataAccessLogic class. Instead, it uses the nonconcrete IEmployeeDataAccessLogic class.

##### **EmployeeBusinessLogic.cs**

namespace SOLID\_PRINCIPLES.DIP

{

public class EmployeeBusinessLogic

{

IEmployeeDataAccessLogic \_IEmployeeDataAccessLogic;

public EmployeeBusinessLogic()

{

\_IEmployeeDataAccessLogic = DataAccessFactory.GetEmployeeDataAccessObj();

}

public Employee GetEmployeeDetails(int id)

{

return \_IEmployeeDataAccessLogic.GetEmployeeDetails(id);

}

}

}

Program.CS

using System;

namespace SOLID\_PRINCIPLES.DIP

{

public class Program

{

static void Main(string[] args)

{

EmployeeBusinessLogic employeeBusinessLogic = new EmployeeBusinessLogic();

Employee emp = employeeBusinessLogic.GetEmployeeDetails(1001);

Console.WriteLine($"ID: {emp.ID}, Name: {emp.Name}, Department: {emp.Department}, Salary: {emp.Salary}");

Console.ReadKey();

}

}

}



**Design Patterns in C#**

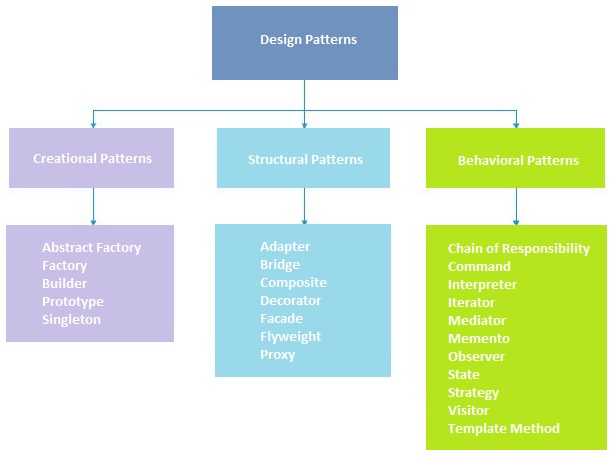
Design patterns provide general solutions or a flexible way to solve common design problems.

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.

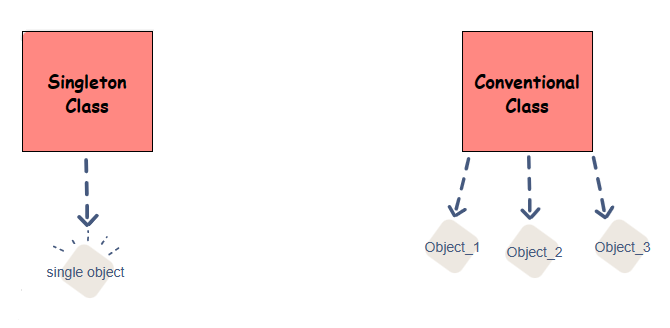
**Behavioral Design Patterns:**

**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());

}

}

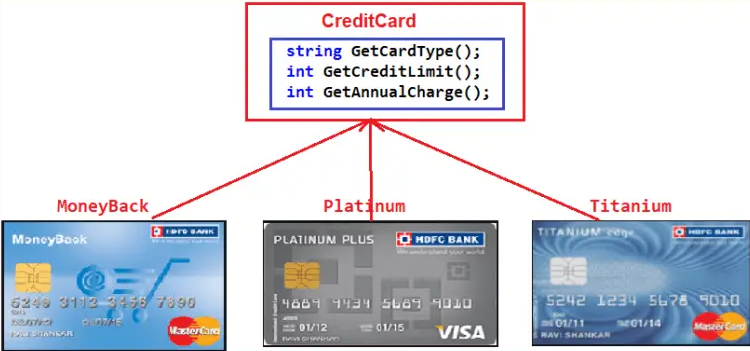
}

##### **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");

}

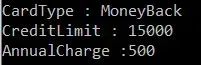
Console.ReadLine();

}

}

}

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.

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();

}

}

}

**Why Do We Need the Dependency Injection Design Pattern in C#?**

The Dependency Injection Design Pattern in C# allows us to develop Loosely Coupled Software Components. In other words, we can say that Dependency Injection Design Pattern is used to reduce the Tight Coupling between the Software Components. As a result, we can easily manage future changes and other complexities in our application. In this case, if we change one component, then it will not impact the other components.

##### **What is Tight Coupling in Software Design?**

**Tight Coupling means two objects are dependent on each other.** That means when a class is dependent on another class, then it is said to be a tight coupling between these two classes. In that case, if we change the Dependent Object, then we also need to change the classes where this dependent object is being used. If your application is a small one, then it is not that difficult to handle the changes but if you have a big Enterprise-Level application, then it’s really very difficult to handle these changes.

##### **What is Loose Coupling in Software Design?**

**Loosely Coupling means two objects are independent of each other.** That means if we change one object then it will not affect another object. The loosely coupled nature of software development allows us to manage future changes easily and also allows us to manage the complexity of the application.

##### **What is Dependency Injection Design Pattern in C#?**

The Dependency Injection Design Pattern in C# is a process in which we are injecting the dependent object of a class into a class that depends on that object. The Dependency Injection Design Pattern is the most commonly used design pattern nowadays to remove the dependencies between the objects.

 The Dependency Injection Design Pattern involves 3 types of classes:

* + 1. **Client Class:** The Client Class (dependent class) is a class that depends on the Service Class. That means the Client Class wants to use the Services (Methods) of the Service Class.
    2. **Service Class:** The Service Class (dependency) is a class that provides the actual services to the client class.
    3. **Injector Class:** The Injector Class is a class that injects the Service Class object into the Client Class.

##### **Different Types of Dependency Injection in C#:**

1. **Constructor Injection:** When the Injector Injects the Dependency Object (i.e. Service Object) into the Client Class through the Client Class Constructor, then it is called Constructor Dependency Injection.
2. **Property Injection:** When the Injector Injects the Dependency Object (i.e. Service Object) into the Client Class through the public Property of the Client Class, then it is called Property Dependency Injection. This is also called the Setter Injection.
3. **Method Injection:** When the Injector Injects the Dependency Object (i.e. Service Object) into the Client Class through a public Method of the Client Class, then it is called Method Dependency Injection.

**Example to Understand Dependency Injection Design Pattern in C#:**

Let us understand the Dependency Injection Design Pattern in C# with an Example. Let us first create a Console Application with the name DependencyInjectionExample. Once you create the Console Application, next we are going to create 3 classes **Employee, EmployeeDAL,**and **EmployeeBL.**

**Employee.cs**

namespace DependencyInjectionDesignPattern

{

//This is going to be our Model class which holds the Model data

//This class is going to be used by both EmployeeDAL and EmployeeBL

public class Employee

{

public int ID { get; set; }

public string Name { get; set; }

public string Department { get; set; }

}

}

**EmployeeDAL.cs (Service)**

using System.Collections.Generic;

namespace DependencyInjectionDesignPattern

{

//Service Class or Dependency Object

//This is the class that is responsible for Interacting with the Database

//This class is going to be used by the EmpoloyeeBL class

//That means it is going to be the Dependency Object

public class EmployeeDAL

{

public List<Employee> SelectAllEmployees()

{

List<Employee> ListEmployees = new List<Employee>

{

//Get the Employees from the Database

//for now we are hard coded the employees

new Employee() { ID = 1, Name = "Pranaya", Department = "IT" },

new Employee() { ID = 2, Name = "Kumar", Department = "HR" },

new Employee() { ID = 3, Name = "Rout", Department = "Payroll" }

};

return ListEmployees;

}

}

}

**EmployeeBL.cs (Client)**

using System.Collections.Generic;

namespace DependencyInjectionDesignPattern

{

//Client Class or Dependent Object

//This is the Class that is going to consume the services provided by the EmployeeDAL Class

//That means it is the Dependent Class which is Depending on the EmployeeDAL Class

public class EmployeeBL

{

public EmployeeDAL employeeDAL;

public List<Employee> GetAllEmployees()

{

//Creating an Instance of Dependency Class means it is a Tight Coupling

employeeDAL = new EmployeeDAL();

return employeeDAL.SelectAllEmployees();

}

}

}

This is a Tight Coupling because the EmployeeDAL is tightly coupled with the EmployeeBL class. Every time the EmployeeDAL class changes, the EmployeeBL class also needs to change. This is the problem. Let us see how to use the Constructor Dependency Injection to make these classes Loosely coupled.

##### **Using Constructor Dependency Injection Design Pattern in C#**

##### 

Let us see how we can use the Constructor Dependency Injection Design Pattern in C# to make these classes loosely coupled. So, first Modify the **EmployeeDAL.cs** class file as shown below. As you can see in the below code, first we create one interface i.e **IEmployeeDAL** with one abstract method i.e. **SelectAllEmployees**. Then that interface is implemented by the EmployeeDAL class and provides implementations for the abstract SelectAllEmployees method.

**EmployeeDAL.cs (Service)**

using System.Collections.Generic;

namespace DependencyInjectionDesignPattern

{

//Service Class or Dependency Object

//Dependency Object should be Interface-Based

public interface IEmployeeDAL

{

List<Employee> SelectAllEmployees();

}

//This is the class that is responsible for Interacting with the Database

//This class is going to be used by the EmpoloyeeBL class

//That means it is going to be the Dependency Object

public class EmployeeDAL : IEmployeeDAL

{

public List<Employee> SelectAllEmployees()

{

List<Employee> ListEmployees = new List<Employee>

{

//Get the Employees from the Database

//for now we are hard coded the employees

new Employee() { ID = 1, Name = "Pranaya", Department = "IT" },

new Employee() { ID = 2, Name = "Kumar", Department = "HR" },

new Employee() { ID = 3, Name = "Rout", Department = "Payroll" }

};

return ListEmployees;

}

}

}

**EmployeeBL.cs (Client)**

using System.Collections.Generic;

namespace DependencyInjectionDesignPattern

{

//Client Class or Dependent Object

//This is the Class that is going to consume the services provided by the IEmployeeDAL Class

//That means it is the Dependent Class which Depending on the IEmployeeDAL Class

public class EmployeeBL

{

public IEmployeeDAL employeeDAL;

//Injecting the Dependency Object using Constructor means it is a Loose Coupling

public EmployeeBL(IEmployeeDAL employeeDAL)

{

this.employeeDAL = employeeDAL;

}

public List<Employee> GetAllEmployees()

{

return employeeDAL.SelectAllEmployees();

}

}

}

So here in the EmployeeBL class, we are not creating the object of the EmployeeDAL class. Instead, we are passing it as a parameter to the constructor of the EmployeeBL class. As we are Injecting the Dependency Object through the constructor, it is called Constructor Dependency Injection in C#.

**Injector Class**

The Injector class will Inject the Dependency Object (EmplpyeeDAL Object) into the Client Class (EmplpyeeBL Class). In our example, the Main method of the Program class is going to be the Injector Class. So, the Injector class will decide which EmployeeDAL instance to be used by the EmployeeBL class, and then the Injector will Inject that instance into the EmployeeBL class through the EmployeeBL class constructor as we are using Constructor Dependency Injection.

using System;

using System.Collections.Generic;

namespace DependencyInjectionDesignPattern

{

class Program

{

static void Main(string[] args)

{

//Create an Instance of EmployeeBL and Inject the Dependency Object as an Argument to the Constructor

EmployeeBL employeeBL = new EmployeeBL(new EmployeeDAL());

List<Employee> ListEmployee = employeeBL.GetAllEmployees();

foreach (Employee emp in ListEmployee)

{

Console.WriteLine($"ID = {emp.ID}, Name = {emp.Name}, Department = {emp.Department}");

}

Console.ReadKey();

}

}

}

