**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.
* stands for the [**Open-Closed Principle**](https://dotnettutorials.net/lesson/open-closed-principle/), also known as **OSP**: 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/" \t "_blank)**, 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);

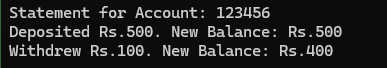
johnsAccount.PrintStatement();

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();

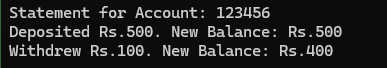
printer.Print(johnsAccount);

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);

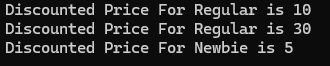
Console.WriteLine($"Discounted Price For {CustomerType.Newbie} is {discountedPrice}");

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();

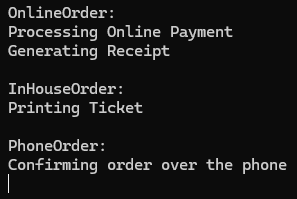
Console.ReadKey();

}

}

}

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();

}

}

}

