**SOLID**

is an acronym and stands for 5 important object oriented principles. The SOLID principles help in making the object oriented application source code robust, scalable, extensible and non-fragile. It is absolutely necessary when you have a layered architecture and they are bound to changes over the period of time. Following are the 5 principles.

1.Single Responsibility Principle

2.Open Closed Principle

3.Liskov Substitution Principle

4.Interface Segregation Principle

5.Dependency Inversion Principle

In this article I will take you through all three all five SOLID principles and also provide sample C# code to demonstrate them.

**Single Responsibility Principle (SRP)**

SRP states that *every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. There should only be a single reason for making the change to a class.*

It means that a class should not be loaded with multiple responsibilities and a single responsibility should not be scattered across multiple classes and mixed with other responsibilities. The reason is that the more changes requested in the future, the more changes the class needs to undergo.

Below is a code violating the SRP. In the sample code, SRP is violated by mixing the OpenGate and CloseGate responsibility with the core vehicle service functionality.

1. public class ServiceStation
2. {
3. public void OpenGate()
4. {
5. //Open the gate if the time is later than 9 AM
6. }
8. public void DoService(Vehicle vehicle)
9. {
10. //Check if service station is opened and then
11. //complete the vehicle service
12. }
14. public void CloseGate()
15. {
16. //Close the gate if the time has crossed 6PM
17. }
18. }

The re-factored code sample is as follows. A new interface is created and the gate related utility methods are moved to a different class called ServiceStationUtility.

1. public class ServiceStation
2. {
3. IGateUtility \_gateUtility;
5. public ServiceStation(IGateUtility gateUtility)
6. {
7. this.\_gateUtility = gateUtility;
8. }
9. public void OpenForService()
10. {
11. \_gateUtility.OpenGate();
12. }
14. public void DoService()
15. {
16. //Check if service station is opened and then
17. //complete the vehicle service
18. }
20. public void CloseForDay()
21. {
22. \_gateUtility.CloseGate();
23. }
24. }
26. public class ServiceStationUtility : IGateUtility
27. {
28. public void OpenGate()
29. {
30. //Open the shop if the time is later than 9 AM
31. }
33. public void CloseGate()
34. {
35. //Close the shop if the time has crossed 6PM
36. }
37. }

40. public interface IGateUtility
41. {
42. void OpenGate();
43. void CloseGate();
44. }

**Open Closed Principle (OCP)**

OCP states that *software application source codes should be open for extension but should be closed for modification.*

According to the OCP principle the code should be easily extensible but it should not need any changes to be done to the core implementations. Following is a C# source code violating OCP where a new car has to be added then it will require changes in the core function CalculateMileage.

1. public class MileageCalculator
2. {
3. IEnumerable<Car> \_cars;
4. public MileageCalculator(IEnumerable<Car> cars) { this.\_cars = cars; }
6. public void CalculateMileage()
7. {
8. foreach (var car in \_cars)
9. {
10. if (car.Name == "Audi")
11. Console.WriteLine("Mileage of the car {0} is {1}", car.Name, "10M");
12. else if (car.Name == "Mercedes")
13. Console.WriteLine("Mileage of the car {0} is {1}", car.Name, "20M");
14. }
15. }
16. }

The OCP violation can be fixed as shown below, using an interface and creating classes for each car there, by reducing making the CalculateMileage method more generic and extensible.

1. public class MileageCalculator
2. {
3. IEnumerable<Car> \_cars;
4. public MileageCalculator(IEnumerable<Car> cars) { this.\_cars = cars; }
6. public void CalculateMileage()
7. {
8. CarController controller = new CarController();
9. foreach (var car in \_cars)
10. {
11. Console.WriteLine("Mileage of the car {0} is {1}", car.Name, controller.GetCarMileage(car.Name));
12. }
13. }
14. }
16. public class CarController
17. {
18. List<ICar> cars;
19. public CarController()
20. {
21. cars = new List<ICar>();
22. cars.Add(new Audi());
23. cars.Add(new Mercedes());
24. }
26. public string GetCarMileage(string name)
27. {
28. return cars.First(car => car.Name == name).GetMileage();
29. }
30. }
32. public interface ICar
33. {
34. string Name { get; set; }
35. string GetMileage();
36. }
38. public class Audi : ICar
39. {
40. public string Name { get; set; }
42. public string GetMileage()
43. {
44. return "10M";
45. }
46. }
48. public class Mercedes : ICar
49. {
50. public string Name { get; set; }
52. public string GetMileage()
53. {
54. return "20M";
55. }
56. }

**Liskov Substitution Principle (LSP)**

LSP states that *the derived classes should be perfectly substitutable for their base classes.* If class D is derived from A then D should be substitutable for A.

Look at the following C# code sample where the LSP is broken. Simply, an Orange cannot substitute an Apple, which results in printing the color of apple as Orange.

1. namespace SolidDemo
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Apple apple = new Orange();
8. Console.WriteLine(apple.GetColor());
9. }
10. }
12. public class Apple
13. {
14. public virtual string GetColor()
15. {
16. return "Red";
17. }
18. }
20. public class Orange : Apple
21. {
22. public override string GetColor()
23. {
24. return "Orange";
25. }
26. }
27. }

Now let us re-factor and make it comply with LSP by having a generic base class for both Apple and Orange.

1. namespace SolidDemo
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Fruit fruit = new Orange();
8. Console.WriteLine(fruit.GetColor());
9. fruit = new Apple();
10. Console.WriteLine(fruit.GetColor());
11. }
12. }
14. public abstract class Fruit
15. {
16. public abstract string GetColor();
17. }
19. public class Apple : Fruit
20. {
21. public override string GetColor()
22. {
23. return "Red";
24. }
25. }
27. public class Orange : Apple
28. {
29. public override string GetColor()
30. {
31. return "Orange";
32. }
33. }
34. }

**Interface Segregation Principle (ISP)**

ISP states that *no clients should be forced to implement methods which it does not use and the contracts should be broken down to thin ones.*

Say for example when a thick interface is defined declaring a wide responsibility of members then there will be opportunities where some clients may have to implement members, which they don’t even use. In the below mentioned example ISP is violated where ProcessCreditCard method is not required by InpersonOrder class but is forced to implement.

1. public interface IOrder
2. {
3. void Purchase();
4. void ProcessCreditCard();
5. }
7. public class OnlineOrder : IOrder
8. {
9. public void Purchase()
10. {
11. //Do purchase
12. }
14. public void ProcessCreditCard()
15. {
16. //process through credit card
17. }
18. }
20. public class InpersionOrder : IOrder
21. {
22. public void Purchase()
23. {
24. //Do purchase
25. }
27. public void ProcessCreditCard()
28. {
29. //Not required for inperson purchase
30. throw new NotImplementedException();
31. }
32. }

Now let us fix the violation by breaking down the IOrder interface.

1. public interface IOrder
2. {
3. void Purchase();
4. }
6. public interface IOnlineOrder
7. {
8. void ProcessCreditCard();
9. }
11. public class OnlineOrder : IOrder, IOnlineOrder
12. {
13. public void Purchase()
14. {
15. //Do purchase
16. }
18. public void ProcessCreditCard()
19. {
20. //process through credit card
21. }
22. }
24. public class InpersionOrder : IOrder
25. {
26. public void Purchase()
27. {
28. //Do purchase
29. }
30. }

**Dependency Inversion Principle (DIP)**

DIP states that *the higher level modules should be coupled with the lower level modules with complete abstraction*.

You can read more about DIP and IoC in my earlier article where I have covered it in-depth. [Please check here](http://www.codeguru.com/columns/experts/implementing-the-inversion-of-control-pattern-in-c.htm).

I hope this article took you through SOLID principles with appropriate C# code samples. Happy reading!