## What is SOLID?

SOLID is an acronym for five principles of architecture.

S – Single Responsibility Principle

O – Open Close Principle

L – Liskov Substitution Principle

I –Interface Segregation Principle

D – Dependency Inversion Principle

## Single Responsibility Principle (SRP)

## Own Way:

This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose.

Means a class is used for only one purpose.

Eg:

1. **public** **class** UserService
2. {
3. EmailService \_emailService;
4. DbContext \_dbContext;
5. **public** UserService(EmailService aEmailService, DbContext aDbContext)
6. {
7. \_emailService = aEmailService;
8. \_dbContext = aDbContext;
9. }
10. **public** **void** Register(**string** email, **string** password)
11. {
12. **if** (!\_emailService.ValidateEmail(email))
13. **throw** **new** ValidationException("Email is not an email");
14. var user = **new** User(email, password);
15. \_dbContext.Save(user);
16. emailService.SendEmail(**new** MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
18. }
19. }
20. **public** **class** EmailService
21. {
22. SmtpClient \_smtpClient;
23. **public** EmailService(SmtpClient aSmtpClient)
24. {
25. \_smtpClient = aSmtpClient;
26. }
27. **public** **bool** **virtual** ValidateEmail(**string** email)
28. {
29. **return** email.Contains("@");
30. }
31. **public** **bool** SendEmail(MailMessage message)
32. {
33. \_smtpClient.Send(message);
34. }
35. }

Above: To validate emails, we have defined another class for this so that actual class is separated from the other methods which need not be bound together to Serve Single Responsibility.

## New Learning

It says that every class should have single responsibility. A class should not have more than one reason to change.

**Example**  
Suppose, you have created a class XmlValidator for XML validation, which has the responsibility to validate XML.

If there is a need to update XML, then a separate class should be created for the same. XmlValidator class should not be used for updating XML.

1. **public** **class** XmlValidator
2. {
4. **public** **void** Validate()
5. {

8. }
10. }

For updating a new class, it should be created.

1. **public** **class** XmlUpdate
2. {
4. **public** **void** DoUpdate()
5. {

8. }

11. }

## Open Close Principle (OCP)

A class should be open for an extension and closed for the modification.

**Example**  
Suppose, we have a class name Customer, which has a property InvoiceNumber, which has an integer type

1. **public** **class** Customer
2. {
3. **public** **int** InvoiceNumber
4. {
5. get;
6. set;
8. }
9. }

In the future, if the requirement changes now, InvoiceNumber should be alphanumeric rather than only an integer. Hence, in this case, you should create a subclass CustomerNew with a same property but different datatype rather than modifying the previous one.

1. **public** **class** CustomerNew : Customer
2. {
4. **public** **new** String InvoiceNumber
5. {
6. get;
7. set;
9. }
10. }

## Liskov Substitution Principle (LSP) - Abstraction

## This is like adding abstract class to make sure "Derived classes must be extending the base class without changing its behavior"

## You should be able to use any derived class instead of a parent class and have it behave in the same manner without modification

## This principle is just an extension of the Open Closed Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior.

A parent object should be able to replace its child during runtime polymorphism.

**Example**  
Suppose, you have two classes, Cooler and Fan, both are inherited from a common interface named ISwitch, which has three methods- On, Off and Regulate.

1. **public** **interface** ISwitch
2. {
3. **void** On();
4. **void** Off();
5. }
7. **public** **class** Cooler : ISwitch
8. {
9. **public** **void** On()
10. {
12. }
13. **public** **void** Off()
14. {
16. }
18. **public** **void** Regulate()
19. {
21. }
23. }
25. **public** **class** Fan : ISwitch
26. {
27. **public** **void** On()
28. {
30. }
32. **public** **void** Off()
33. {
35. }
36. **public** **void** Regulate()
37. {
39. }
41. }
43. **public** **class** MainClass
44. {
45. **public** **void** AddObject()
46. {
47. List<ISwitch> Switch = **new** List<ISwitch>();
48. Switch.Add(**new** Cooler());
49. Switch.Add(**new** Fan());
51. foreach (**var** o **in** Switch)
52. {
53. o.Regulate();
54. }
55. }
56. }

Everything was fine until a new class introduced for same interface named Bulb, which has only two methods On and Off. It does not have Regulate method. Thus Bulb class is given below.

1. **public** **class** Bulb : ISwitch
2. {
3. **public** **void** On()
4. {
6. }
8. **public** **void** Off()
9. {
11. }
12. **public** **void** Regulate()
13. {
14. **throw** **new** NotImplementedException();
15. }
17. }

Now, AddObject method will be updated, as shown below.

1. **public** **void** AddObject()
2. {
3. List<ISwitch> Switch = **new** List<ISwitch>();
4. Switch.Add(**new** Cooler());
5. Switch.Add(**new** Fan());
6. Switch.Add(**new** Bulb());
8. foreach (**var** o **in** Switch)
9. {
10. o.Regulate();
11. }
12. }

In this case, Regulate method will throw an error.

One horrible solution to this problem is to put an if condition.

1. foreach (**var** o **in** Switch)
2. {
4. **if**(o is Bulb)
5. **continue**;
7. o.Regulate();
8. }

This is an example of bad design, if above condition is used somewhere, it clearly means that there is a violation of LSK principle.

## Interface Segregation Principle (ISP)

Client specific interfaces are better than general purpose interfaces.

Suppose, we have one interface for clicking.

1. **public** **interface** IClick{
2. **void** onClick(Object obj);
3. }

As time passes, new requirement comes for adding one more function onLongClick. You need to add this method in already created interface.

1. **public** **interface** IClick{
2. **void** onClick(Object obj);
3. **void** onLongClick(Object obj);
4. }

After some time, one new requirement comes for adding function for touch also and you need to add the method in the same interface

1. **public** **interface** IClick{
2. **void** onClick(Object obj);
3. **void** onLongClick(Object obj);
4. **void** onTouch(Object obj);
5. }

At this point, you need to decide to change the name of interface too because touch is different than click.

In this way, this interface becomes a problem—generic and polluted. At this stage, ISP comes into play.

## Why Generic Interface creates problem?

Suppose, some clients need only onClick function and some need only onTouch function, then one will be useless for both. Hence ISP gives the solution, which splits the interface into two interfaces.

ITouch and IClick. The client which has required onClick can implement IClick, which needs onTouch. It can implement ITouch and when it needs both, it can implement both.

1. **public** **interface** IClick{
2. **void** onClick(Object obj);
3. **void** onLongClick(Object obj);
4. }
6. **public** **interface** ITouch{
7. **void** onClick(Object obj);
8. **void** onTouch(Object obj);
9. }

## Dependency Inversion Principle (ISP)

It states two points, where the first point is a higher level module, which should not depend on a low level module. Both should depend on abstraction or interfaces. The second point is abstraction, which should not depend on detail. Detail should depend on abstraction.

In other words, no object should be created inside a class. They should be passed or injected from outside. When it  is received, it will be an interface rather than a class.

We have our eg in my own application.

Its more like adding Interfaces and calling them instead of calling the main class all the time. This helps in decouple it. Also for any new implementation of the Interface, you can easily switch it to the new class.  
  
eg: EmployeeBusinessLogic does not depend on concrete EmployeeDataAccess class, instead, it includes a reference of IEmployeeDataAccess interface