S.O.L.I.D.

.NET

In Object-Oriented Programming, ***S.O.L.I.D.*** *is an acronym for five design principles intended to make software more understandable, flexible and maintainable.*

HTTPS://EN.WIKIPEDIA.ORG/WIKI/SOLID

S.O.L.I.D. – Overview

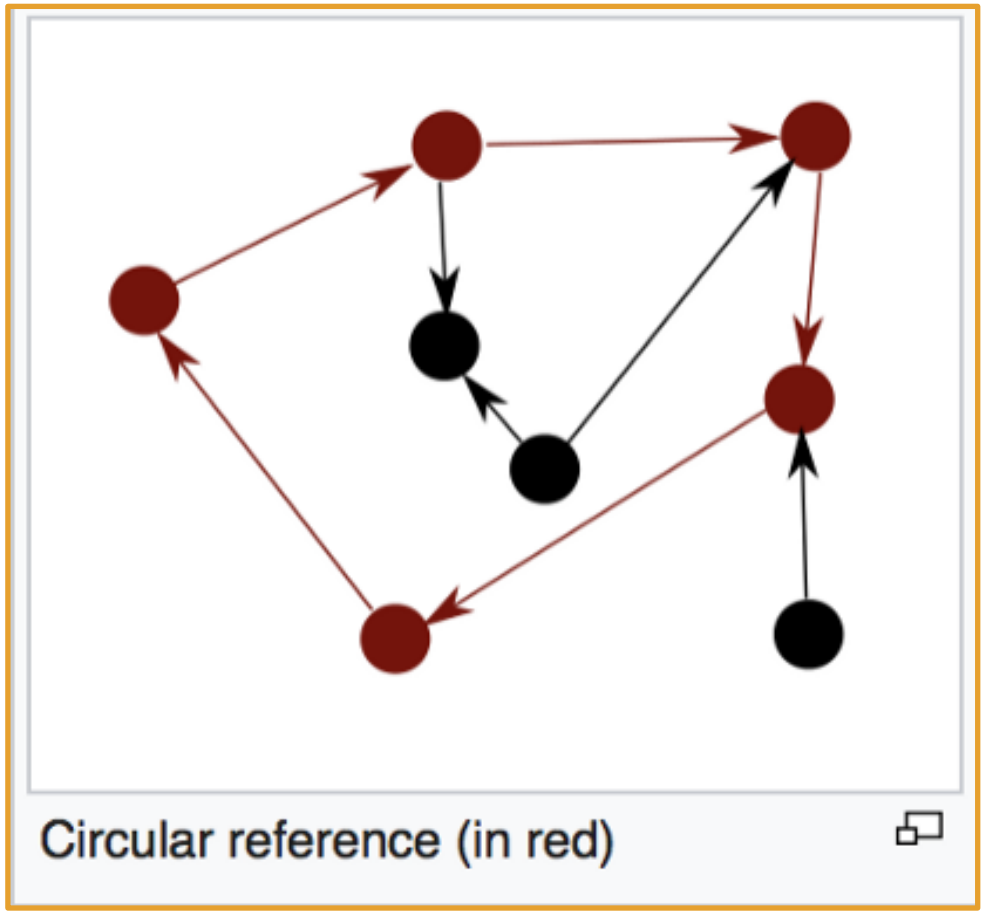
https://medium.com/better-programming/solid-principles-simple-and-easy-explanation-f57d86c47a7f
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/
https://medium.com/better-programming/what-is-bad-code-f963ca51c47a

---

SOLID Principles is a coding standard that helps developers avoid problematic design in software development.

When applied properly, SOLID principles make code easier to refactor, easier to debug, and easier to read.

Badly designed software is inflexible and brittle. Small changes can result in a cascade of problems that break various parts of the code.



Circular reference (in red)

Single Responsibility Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

A class should only have one responsibility. When classes have single responsibilities, changes to software should only affect one class at a time.

In this example, SendEmail() and ValidateEmail() serve a logically different purpose from the UserService class, which registers a new user.

The UserService class should not contain logic to be sending and validating emails.

Sending and validating emails should be done by a Service classes.

```
1  public class UserService
2  {
3      SmtpClient _smtpClient;//email service
4      DbContext _dbContext;
5      public UserService( DbContext aDbContext, SmtpClient aSmtpClient)
6      {
7          _dbContext = aDbContext;
8          _smtpClient = aSmtpClient;
9      }
10     //validate and send an email
11     public void Register(string email, string password)
12     {
13         //verify that email string contains a '@'
14         if (!ValidateEmail(email))
15             {  throw new ValidationException("Email is not an email");  }
16
17         var user = new User(email, password);// create a new user
18         _dbContext.Save(user); //save the new user to the DataBase
19
20         //call SendEmail() with a MailMessage Object.
21         SendEmail(new MailMessage( "mysite@nowhere.com",  email)
22             { Subject = "Your account creation was successful!" } );
23     }
24     //verify the the email string has a '@'
25     public virtual bool ValidateEmail(string email)
26         { return email.Contains("@"); }
27     public bool SendEmail(MailMessage message)//send the message.
28         { _smtpClient.Send(message); }
29 }
```

Single Responsibility Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

To fulfill the ***Single Responsibility Principal***, now UserService only creates a new user. It leverages EmailService for anything email related.
EmailService is a service class that is ***injected*** into any other class that needs to handle emails. It is very basic. It only verifies the email address and sends the email.
You can add as much related functionality as needed to each service class.

```
using System.ComponentModel.DataAnnotations;
using System.Net.Mail;

public class UserService
{
    EmailService _emailService;
    DbContext _dbContext;
    public UserService(EmailService aEmailService, DbContext aDbContext)
    {
        _emailService = aEmailService;
        _dbContext = aDbContext;
    }
    public void Register(string email, string password)
    {
        if (!_emailService.ValidateEmail(email))
            throw new ValidationException("Email is not an email");
        var user = new User(email, password);
        _dbContext.Save(user);
        _emailService.SendEmail(new MailMessage("myname@mydomain.com", email) { Subject = "Hi. How are you!" });

    }
}
public class EmailService
{
    SmtpClient _smtpClient;
    public EmailService(SmtpClient aSmtpClient)
    {
        _smtpClient = aSmtpClient;
    }
    public virtual bool ValidateEmail(string email)
    {
        return email.Contains("@");
    }
    public void SendEmail(MailMessage message)
    {
        _smtpClient.Send(message);
    }
}
```

Open-Closed Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

"A class should be open for extension but closed to modification".

Modules and classes must be designed in such a way that new functionality can be added when new requirements are generated. We implement ***interfaces*** and use ***inheritance*** to do this.

```
public class Rectangle
{
    public double Height { get; set; }
    public double Width { get; set; }
}
```

The Rectangle class needs to be able to calculate the total area of a collection of Rectangles. The ***Single Responsibility Principle*** dictates that we should not put the total area calculation code inside the Rectangle class.

How can this problem be solved?

Open-Closed Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

We create a class specifically to calculate the area of objects.

```
public class AreaCalculator
{
    public double TotalArea(Rectangle[] arrRectangles)
    {
        double area = 0;
        foreach (var objRectangle in arrRectangles)
        {
            area += objRectangle.Height * objRectangle.Width;
        }
        return area;
    }
}
```

EVEN BETTER. Create one class for the calculation of the area of ANY shape. →→→→→→→→→→→→

```
public class Rectangle
{
    public double Height { get; set; }
    public double Width { get; set; }
}
public class Circle
{
    public double Radius { get; set; }
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Circle objCircle;
        foreach (var obj in arrObjects)
        {
            if (obj is Rectangle)
            {
                area += obj.Height * obj.Width;
            }
            else
            {
                objCircle = (Circle)obj;
                area += objCircle.Radius * objCircle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

Liskov Substitution Principle

https://medium.com/better-programming/solid-principles-simple-and-easy-explanation-f57d86c47a7f
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

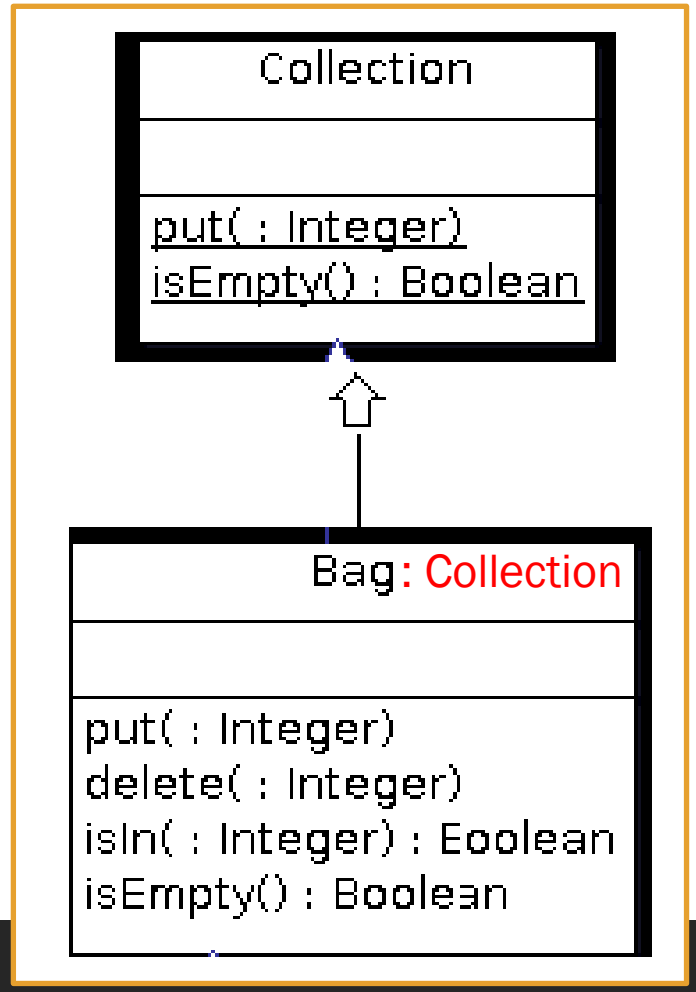
Derived classes must implement all the methods and fields of their ***parent***.

After implementing the methods and fields of the parent, you will be able to use any ***derived*** class instead of a ***parent*** class and it will behave in the same manner.

This ensures that a ***derived*** class does not affect the behavior of the ***parent*** class.

A ***derived*** class must be substitutable for its ***base (parent/super)*** class.

Interfaces help us implement this principle by defining methods but leaving the implementation to the developer. This allows you to abstract away dependencies of the class when testing.



How Liskov SP Works IRL.

```
// Bag inherits from Collection
Bag myBag = new Bag();//This Bag TYPE inherits (derives from) from Collections TYPE
myBag.BagMethod();// BagMethod is a new method on Bag only
myBag.CollectionMethod();//CollecitonsMethod is a Collection class method inherited by Bag
//myBags' actual value is the memory location on the heap of the Bag object.
Collection myCollection = myBag;//assign the memory location to the collection TYPE variable.
~~myCollection.BagMethod();~~// a Collection TYPE variable cannot access a Bag TYPE method.
myCollection.CollectionMethod();// This still works.
```

Interface Segregation Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

Each ***interface*** should have a specific purpose or responsibility.

Large ***interfaces*** are more likely to include methods that not all classes can implement.

Client classes should not be forced to depend on ***interfaces*** with methods they will never use.

Dependency Inversion Principle

https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

"High-level modules should not depend on low-level modules. Both should depend upon abstractions."

Higher-level modules/classes implement business rules or logic in a system.

Lower-level modules/classes deal with more detailed operations. They may deal with writing information to databases or passing messages to the operating system or services.

When a class is directly dependent on another class, changes to the dependency class will break the dependent class. We keep these high-level and low-level classes as ***loosely coupled*** as possible.

To do that, make both classes dependent on abstractions (interfaces) instead of on each other.