S.O.L.I.D.

.NET CORE

In Object-Oriented Programming, ***S.O.L.I.D.*** *is five design principles intended to make software designs 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

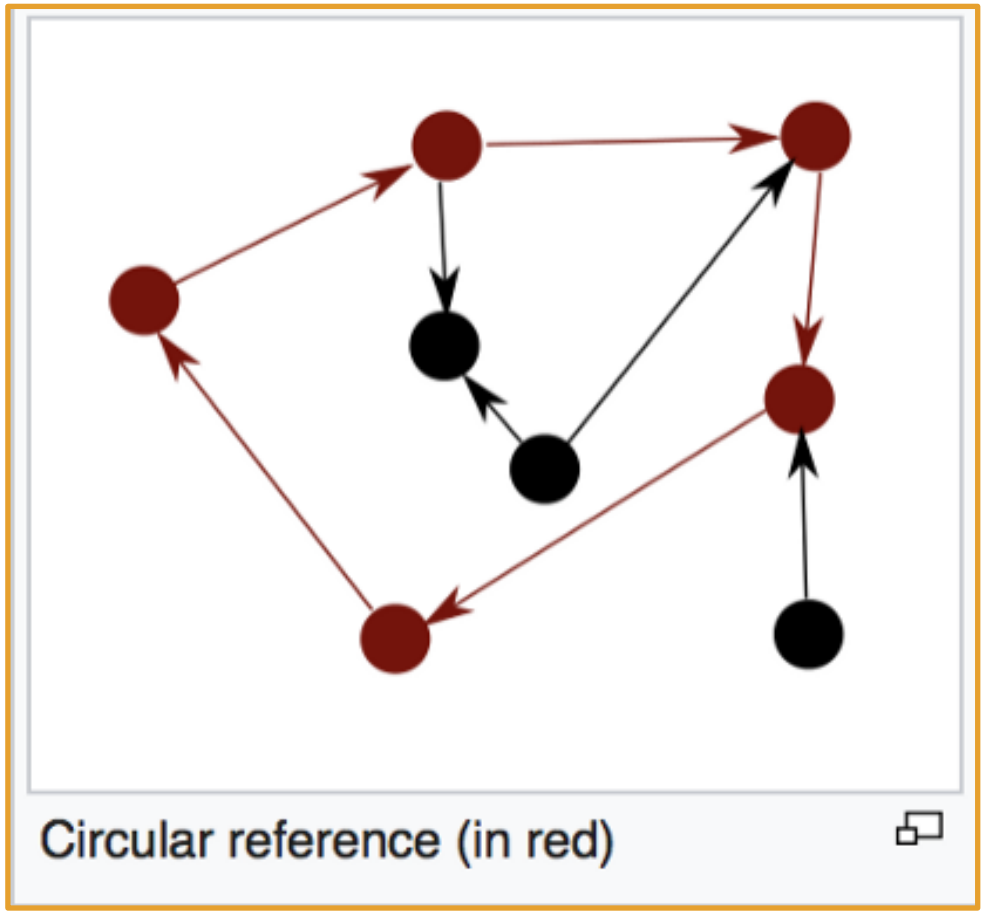
---

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

When applied properly, it makes code more extensible, more logical, and easier to read.

Badly designed software can become inflexible and brittle.

Small changes in the software can result in bugs that break other 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. Changes to a part of the software should only be able to affect the specification of one class.
On examination, we see that SendEmail() and ValidateEmail() have nothing to do with the UserService class.
They should be moved into their own class. Let's refactor it on the next slide.

```
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/

Now, UserService only creates a new user. It leverages EmailService for anything email related.
EmailService is a class that is injected into any other class that needs to handle emails.
EmailService is very basic. It only verifies the email address and sends the email. In a real situation, you could add as much related functionality as you needed.

```
1  public class UserService
2  {
3      EmailService _emailService;//DI to get the EmailService
4      DbContext _dbContext;//DI to get access to the DB
5
6      public UserService(EmailService aEmailService, DbContext aDbContext)//constructor
7      {
8          _emailService = aEmailService;
9          _dbContext = aDbContext;
10     }
11
12     public void Register(string email, string password)
13     {
14         if (!_emailService.ValidateEmail(email)) //verify that the email is valid.
15          throw new ValidationException("Email is not an email");//throw an error
16
17         var user = new User(email, password);//create a new user
18         _dbContext.Save(user);//save the new user to the DB
19
20          //call the email service to send the email.
21          emailService.SendEmail(new MailMessage("myname@mydomain.com", email)
22                                                 { Subject = "Hi. How are you!" });
23     }
24 }
25 public class EmailService
26 {
27     SmtpClient _smtpClient;
28
29     public EmailService(SmtpClient aSmtpClient)
30        {  _smtpClient = aSmtpClient;  }
31
32     public virtual bool ValidateEmail(string email)
33        {  return email.Contains("@");  }
34
35     public bool SendEmail(MailMessage message)
36        {  _smtpClient.Send(message);  }
37 }
```

Open-Closed Principle

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

"A class should be open for extensions but closed to modifications".

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

This app needs the ability to calculate the total area of a collection of Rectangles. Because of the ***Single Responsibility Principle***, we can't put the total area calculation code inside the rectangle!

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

How can this problem be solved?

Open-Closed Principle

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

We create another class specifically for the calculation of the area of a Rectangle object array?

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

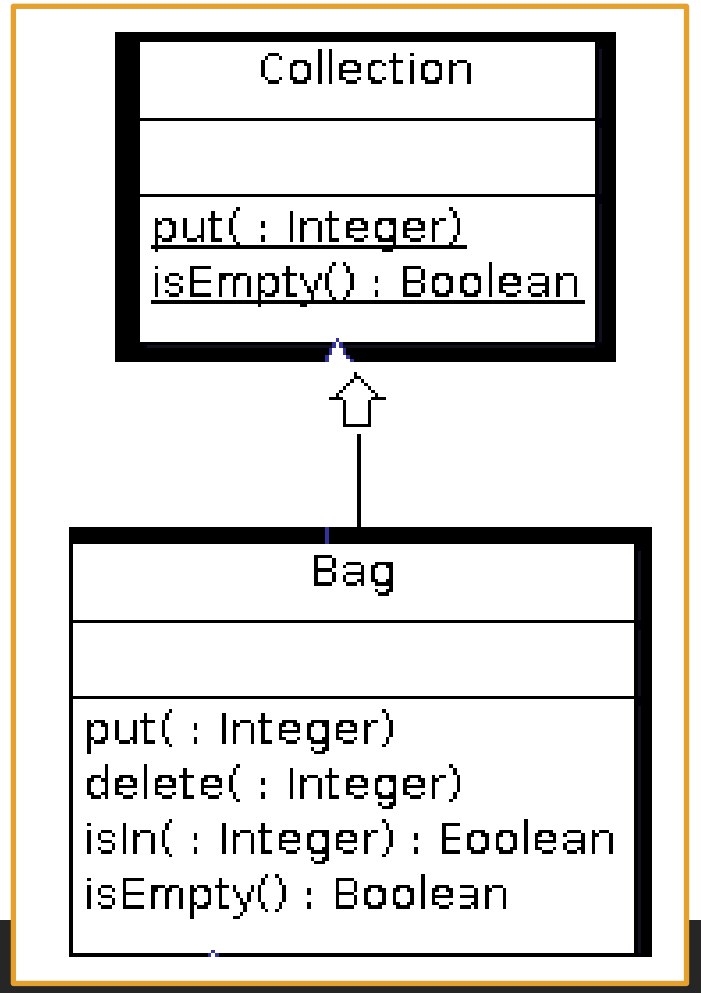
EVEN BETTER. Create one class for the calculation of the area of any shape in the program. →→→→→→→→→→→→

```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        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*** class.
Functions that use pointers of references to base classes must be able to use objects of ***derived*** classes without knowing it.



Interface Segregation Principle

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

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

A class shouldn't be forced to implement an ***interface*** when the class doesn't share the ***interfaces*** purpose.

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

Clients should not be forced to depend upon ***interfaces*** whose methods they don't use.

Dependency Inversion Principle

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

High-level modules/classes implement business rules or logic in a system (front-end).

Low-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 too closely uses the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/classes ***loosely coupled*** as much as possible.

To do that, we need to make both of them dependent on abstractions instead of knowing each other.