S.O.L.I.D.

.NET

In object-oriented computer programming, **S.O.L.I.D.** is a mnemonic acronym for 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/

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design.

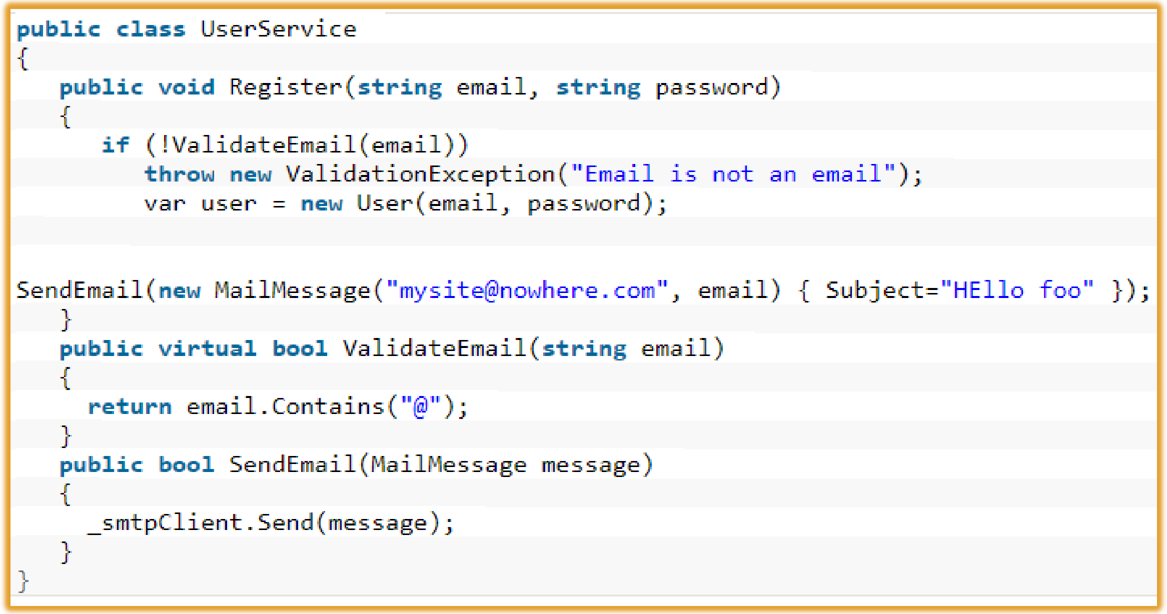
When applied properly it makes your code more extensible, logical, and easier to read.

When the developer builds software following a bad design, the code can become inflexible and brittle. Small changes in the software can result in bugs that break other parts of the code.

Single Responsibility Principle

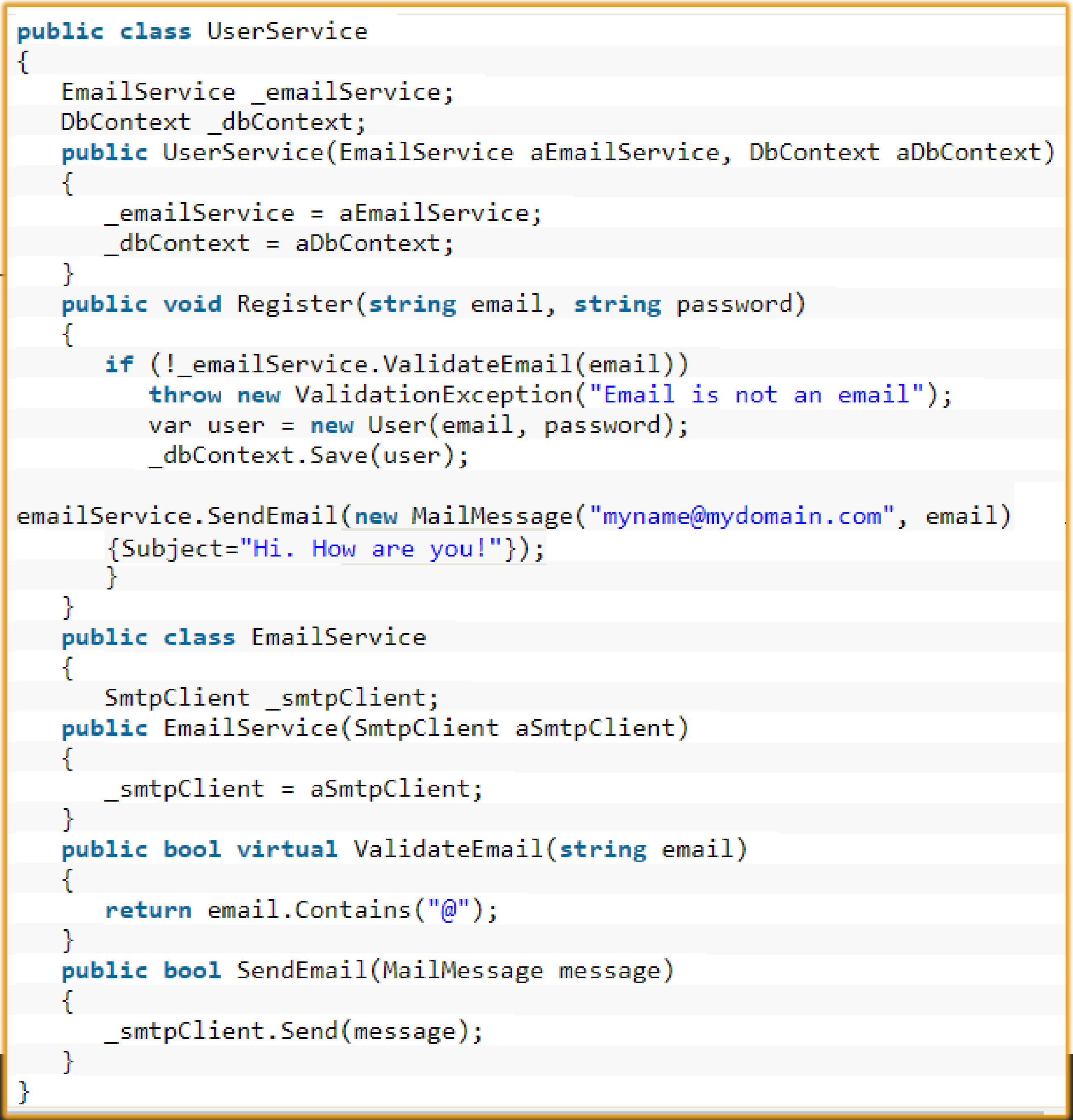
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

A class should only have a single responsibility. Only changes to one part of the software's specification should be able to affect the specification of the class.



```
public class UserService
{
   public void Register(string email, string password)
   {
      if (!ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);


SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
   }
   public virtual bool ValidateEmail(string email)
   {
     return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
     _smtpClient.Send(message);
   }
}
```

The SendEmail and ValidateEmail methods have nothing to do within the UserService class. Let's refactor it. →→→



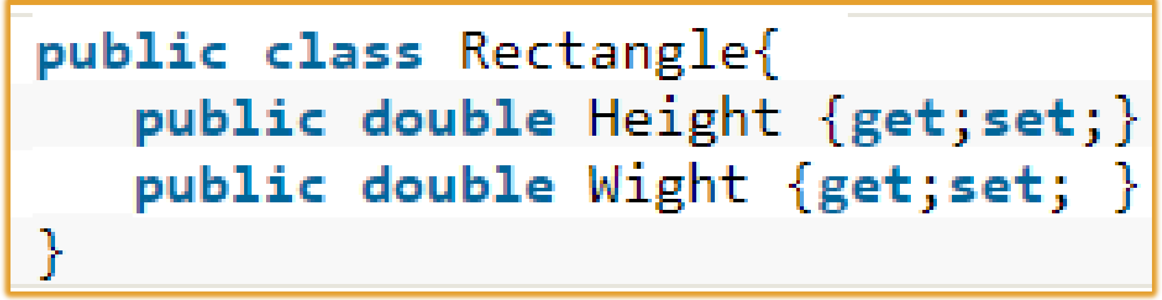
```
public class UserService
{
   EmailService _emailService;
   DbContext _dbContext;
   public UserService(EmailService aEmailService, DbContext aDbContext)
   {
      _emailService = aEmailService;
      _dbContext = aDbContext;
   }
   public void Register(string email, string password)
   {
      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 bool virtual ValidateEmail(string email)
   {
      return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
      _smtpClient.Send(message);
   }
}
```

Open-Closed Principle

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

We need to design our module/class in such a way that the new functionality can be added only when new requirements are generated.
A class should be open for extensions. We can use ***inheritance*** to do this.

Suppose we have a Rectangle class with the properties Height and Width.



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

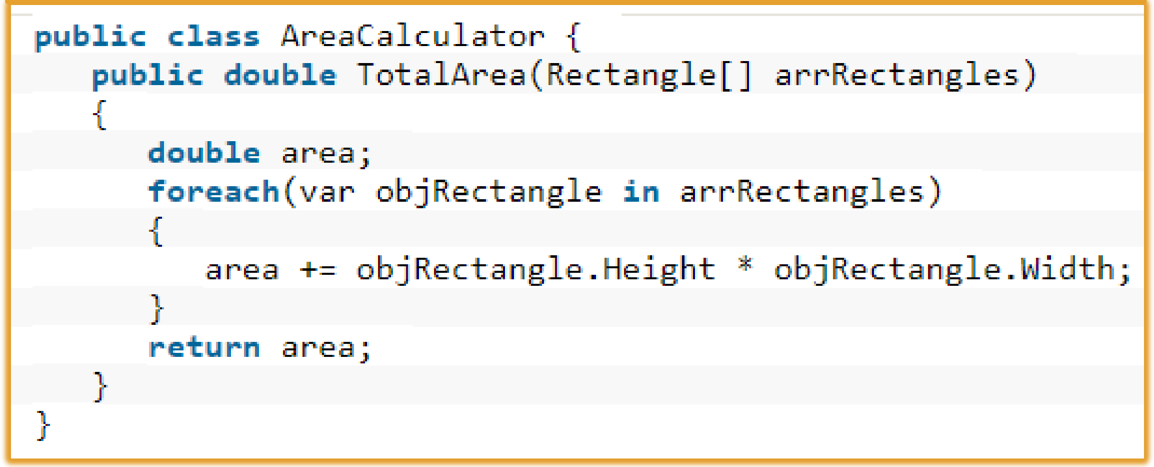
An app needs the ability to calculate the total area of a collection of Rectangles.
Because of the ***Single Responsibility Principle***, we can't put the total area calculation code inside the rectangle!

How can this problem be solved?

Open-Closed Principle

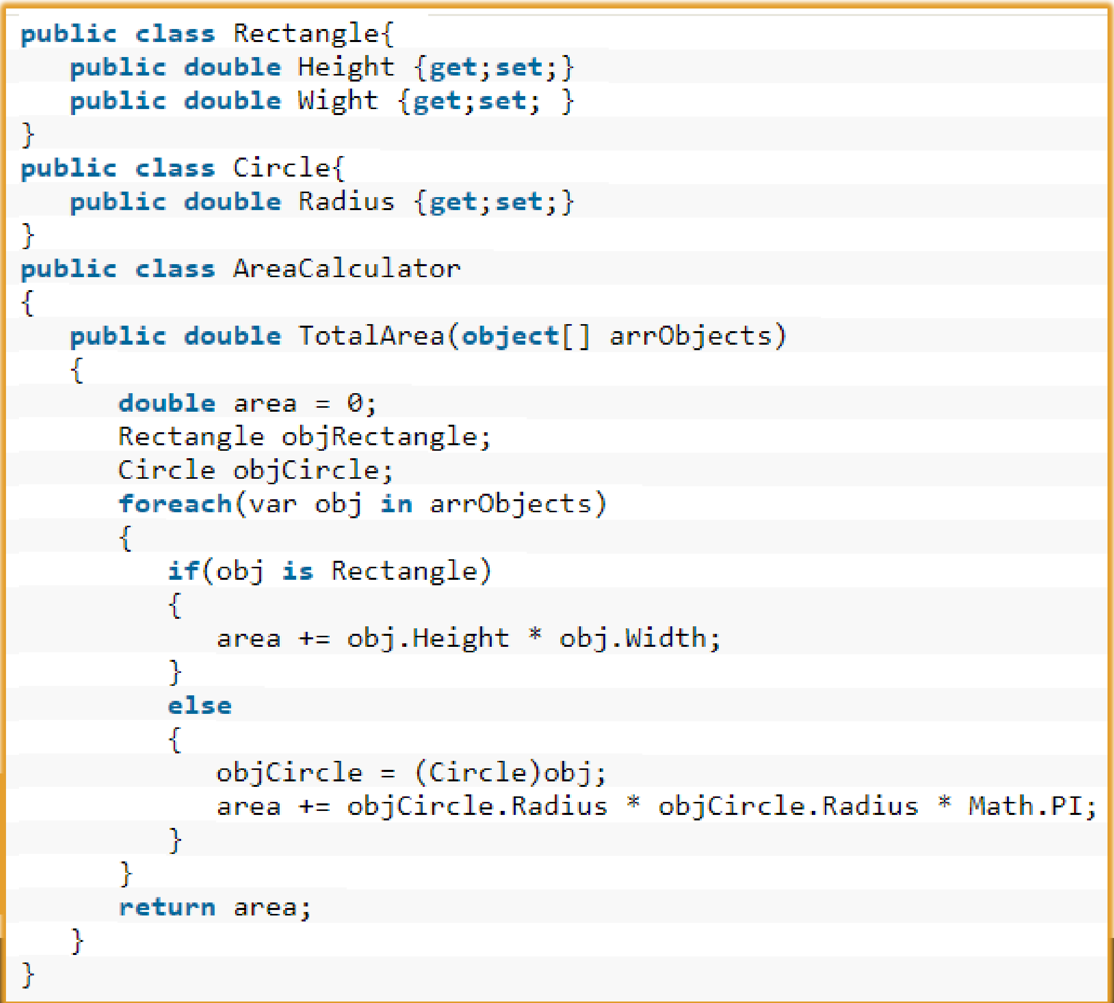
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

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. →→→→→→→→→→→

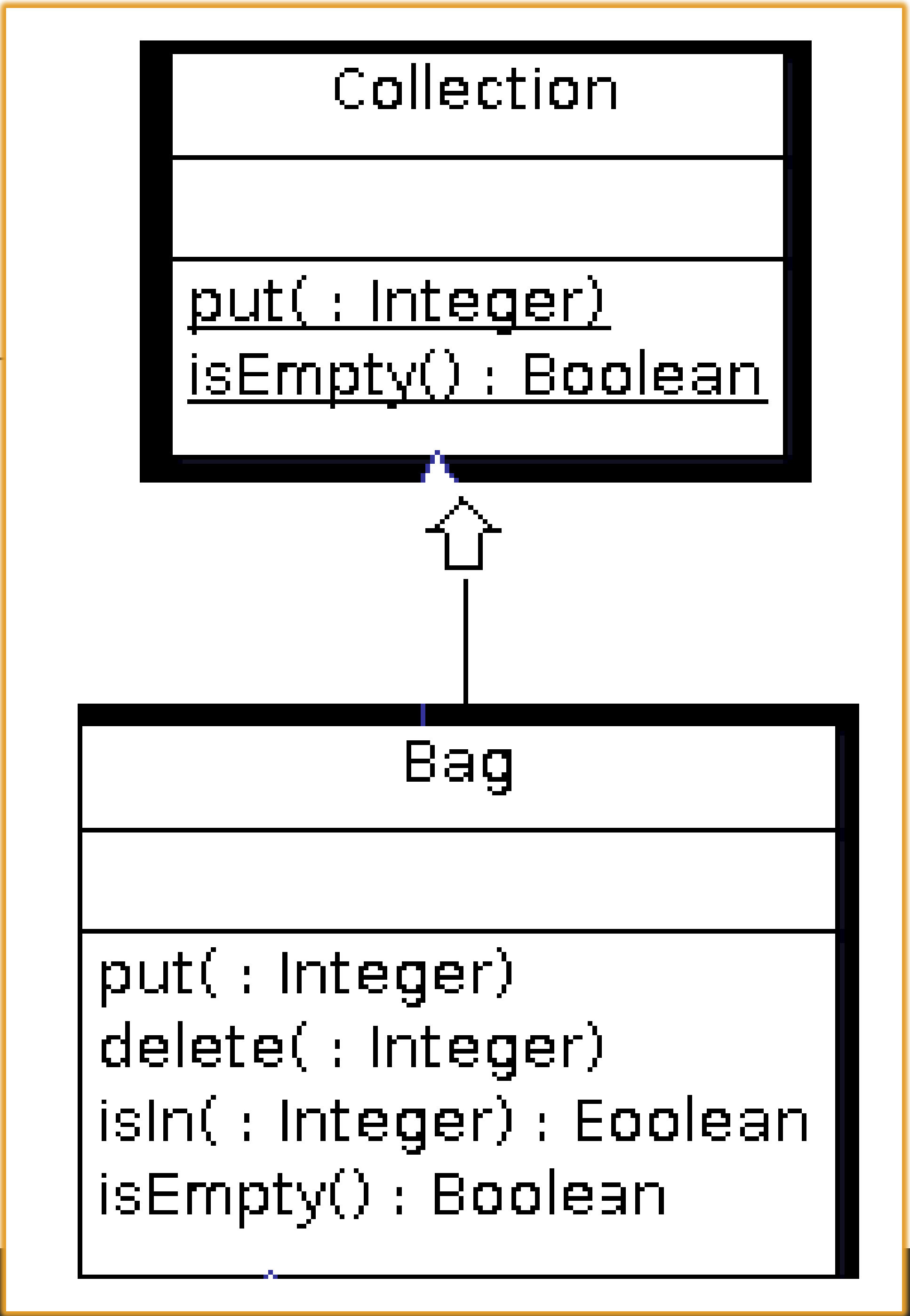


```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        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/

More ***derived*** classes should 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/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.

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Dependency Inversion Principle

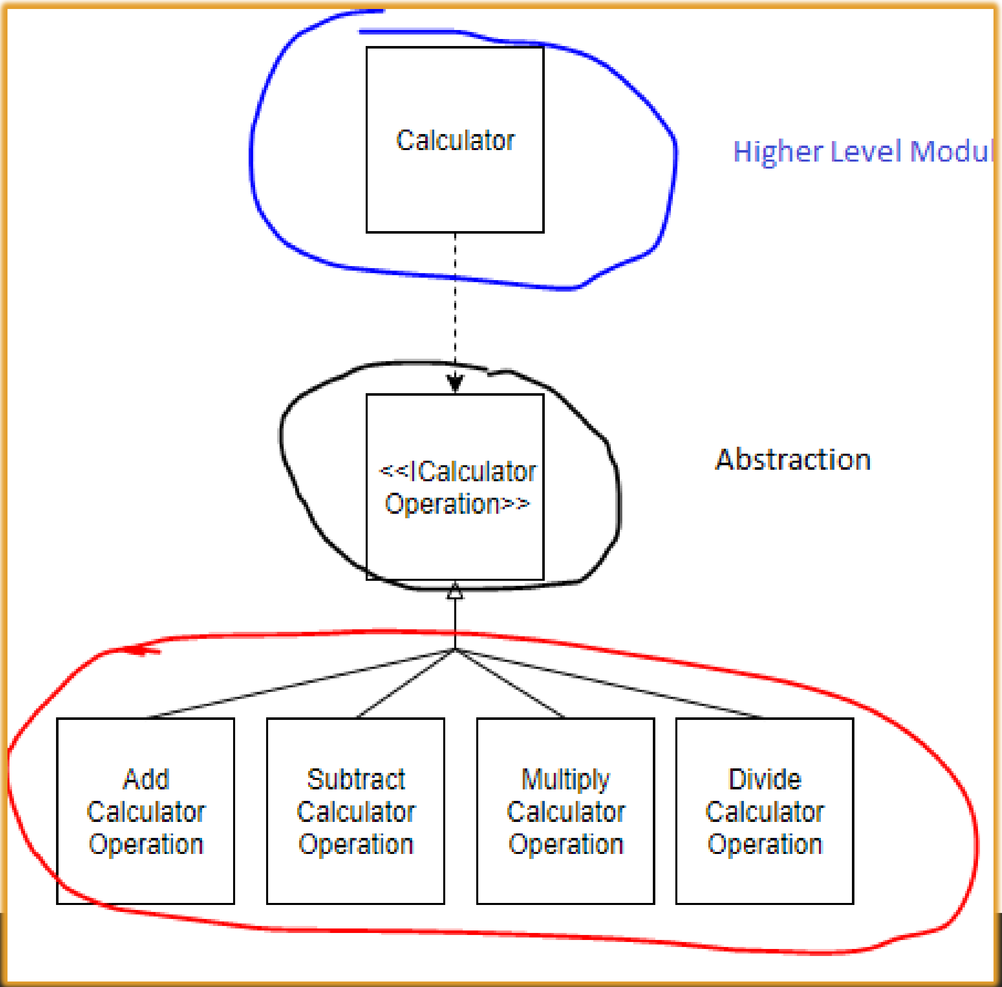
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

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.


Calculator
Higher Level Modul
<<ICalculator Operation>>
Abstraction
Add Calculator Operation
Subtract Calculator Operation
Multiply Calculator Operation
Divide Calculator Operation