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

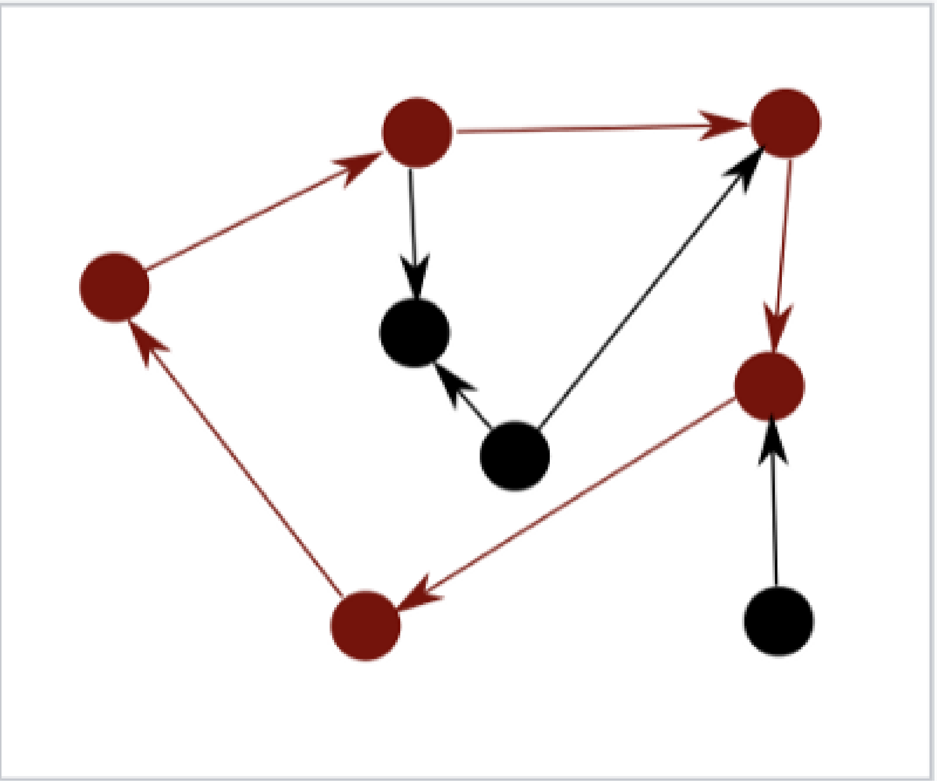
https://medium.com/better-programming/solid-principles-simple-and-easy-explanation-f57d86c47a7f
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/
https://medium.com/better-programming/what-is-bad-code-f963ca51c47a

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

When applied properly, 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.



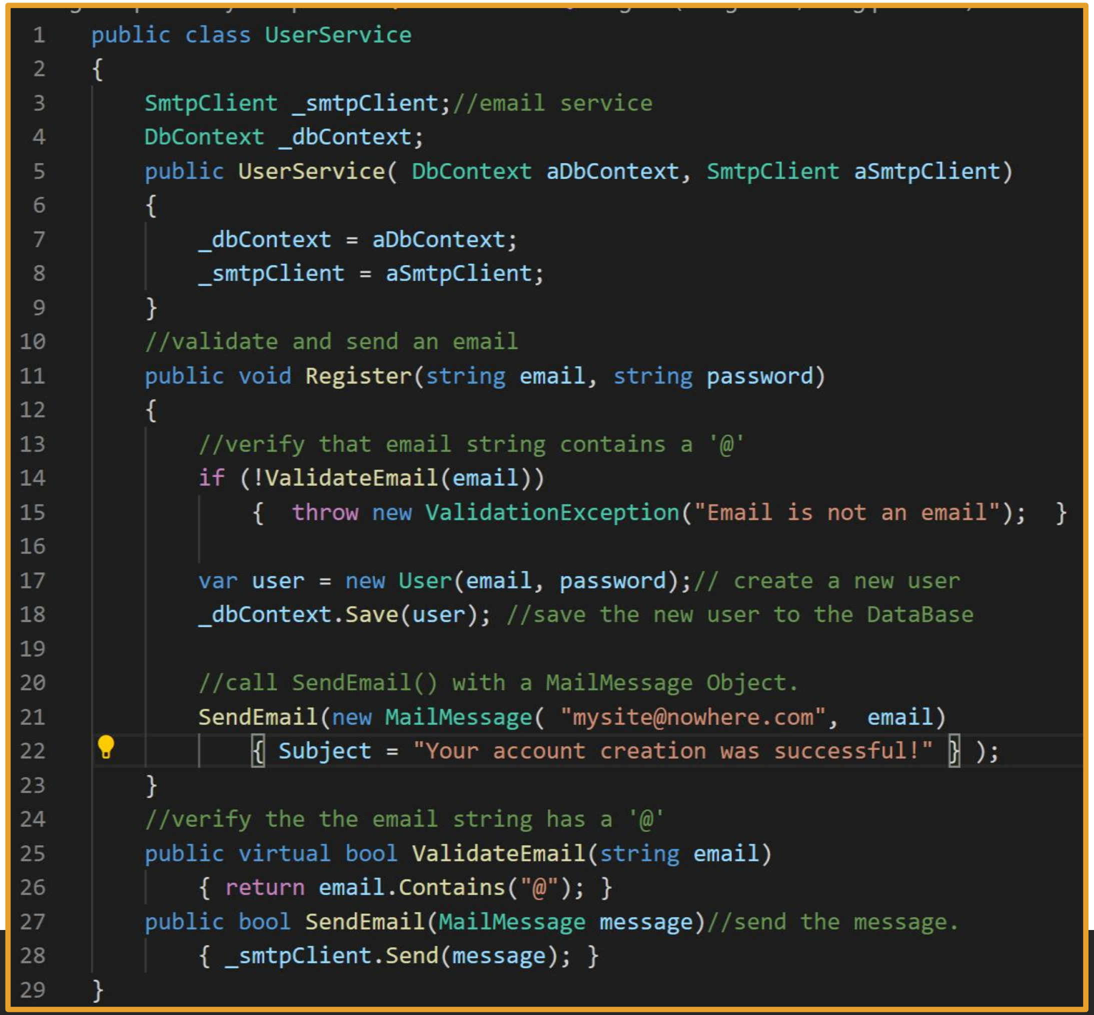
Circular reference (in red)

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.

In this example, we see that SendEmail() and ValidateEmail() serve a logically different purpose from the UserService class.
They should not contain the logic to be sending and validating emails.



```
public class UserService
{
    SmtpClient _smtpClient;//email service
    DbContext _dbContext;
    public UserService( DbContext aDbContext, SmtpClient aSmtpClient)
    {
        _dbContext = aDbContext;
        _smtpClient = aSmtpClient;
    }
    //validate and send an email
    public void Register(string email, string password)
    {
        //verify that email string contains a '@'
        if (!ValidateEmail(email))
            {  throw new ValidationException("Email is not an email");  }

        var user = new User(email, password);// create a new user
        _dbContext.Save(user); //save the new user to the DataBase

        //call SendEmail() with a MailMessage Object.
        SendEmail(new MailMessage( "mysite@nowhere.com",  email)
            { Subject = "Your account creation was successful!" } );
    }
    //verify the the email string has a '@'
    public virtual bool ValidateEmail(string email)
        { return email.Contains("@"); }
    public bool SendEmail(MailMessage message)//send the message.
        { _smtpClient.Send(message); }
}
```

Single Responsibility Principle

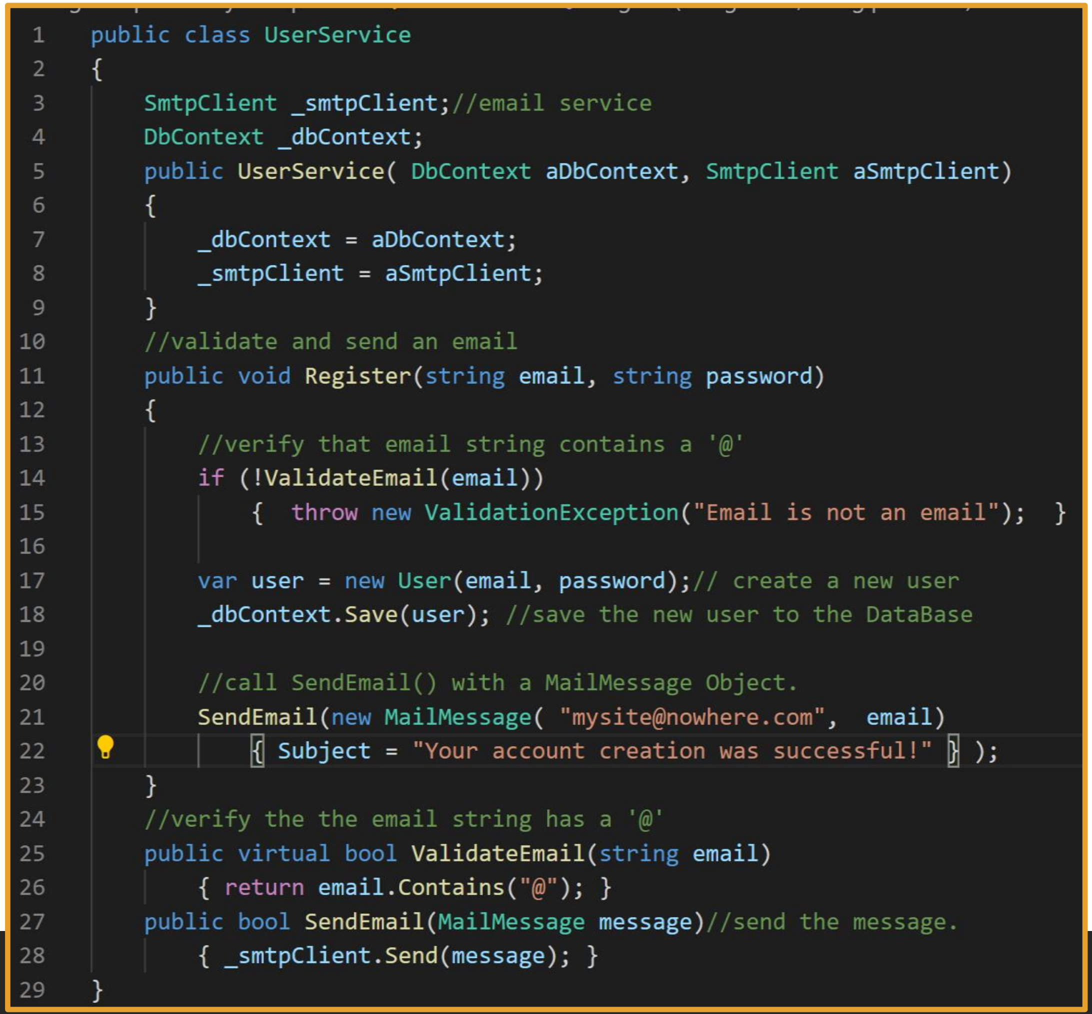
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

To fulfill the ***Single Responsibility Principal***, 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 needed to each class.



```
public class UserService
{
    SmtpClient _smtpClient;//email service
    DbContext _dbContext;
    public UserService( DbContext aDbContext, SmtpClient aSmtpClient)
    {
        _dbContext = aDbContext;
        _smtpClient = aSmtpClient;
    }
    //validate and send an email
    public void Register(string email, string password)
    {
        //verify that email string contains a '@'
        if (!ValidateEmail(email))
            {  throw new ValidationException("Email is not an email");  }

        var user = new User(email, password);// create a new user
        _dbContext.Save(user); //save the new user to the DataBase

        //call SendEmail() with a MailMessage Object.
        SendEmail(new MailMessage( "mysite@nowhere.com",  email)
            { Subject = "Your account creation was successful!" });
    }
    //verify the the email string has a '@'
    public virtual bool ValidateEmail(string email)
        { return email.Contains("@"); }
    public bool SendEmail(MailMessage message)//send the message.
        { _smtpClient.Send(message); }
}
```

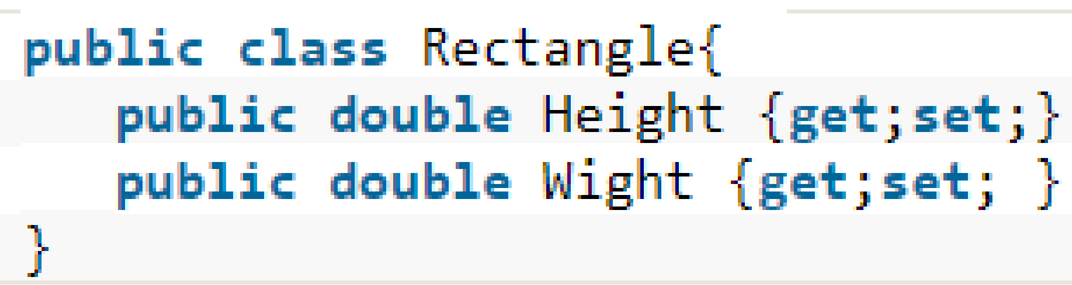
Open-Closed Principle

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 can implement interfaces and use ***inheritance*** 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 shouldn’t put the total area calculation code inside the rectangle.



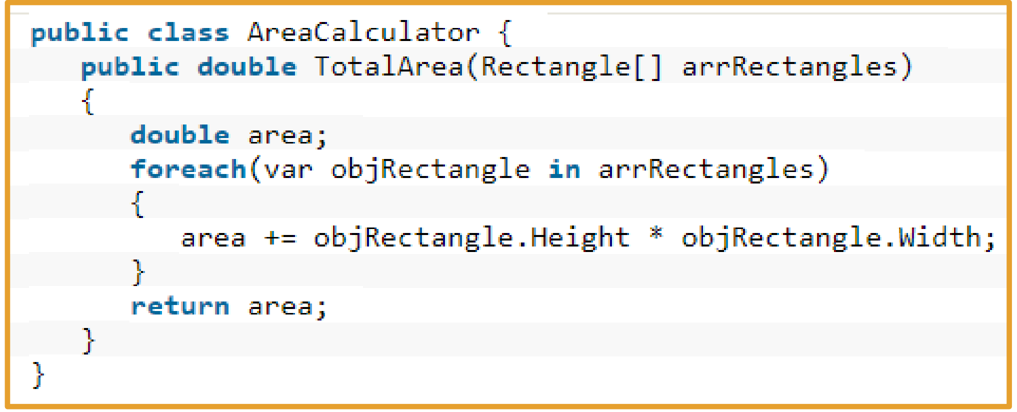
```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
```

How can this problem be solved?

Open-Closed Principle

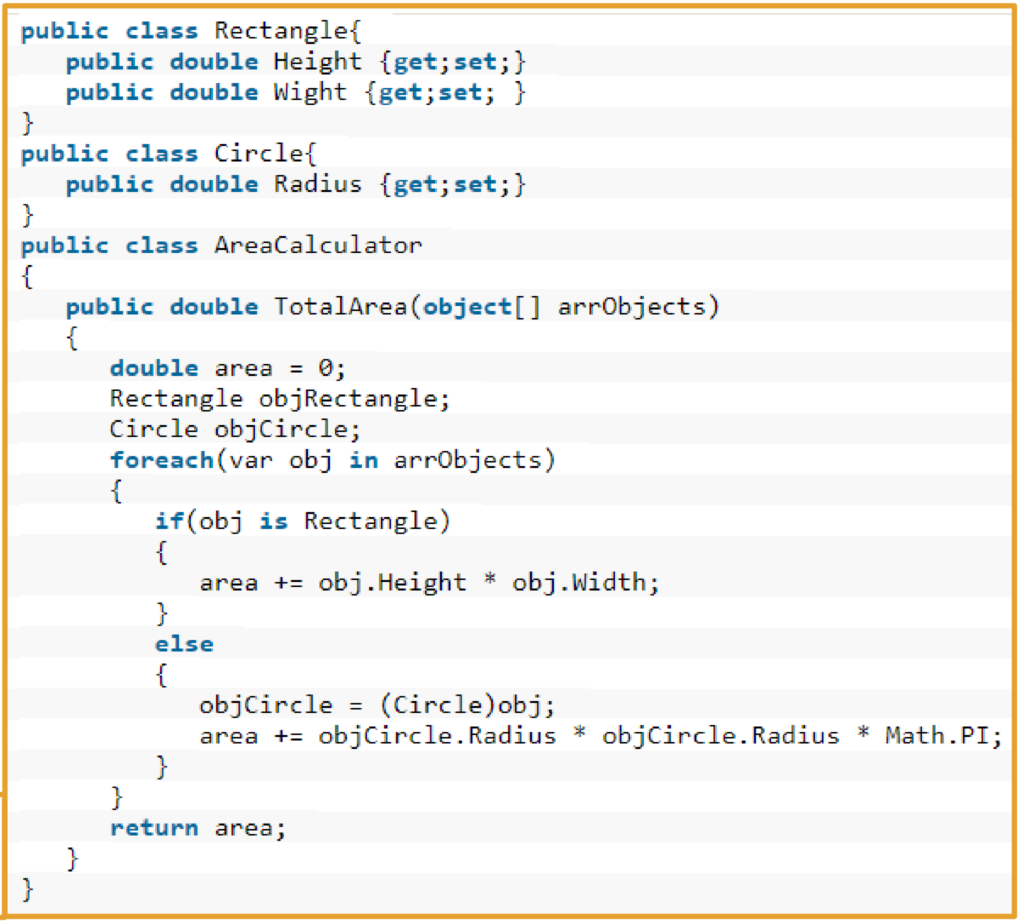
https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

We create class specifically to calculate the area of a Rectangle object.



```
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. →→→→→→→→→→→



```
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/

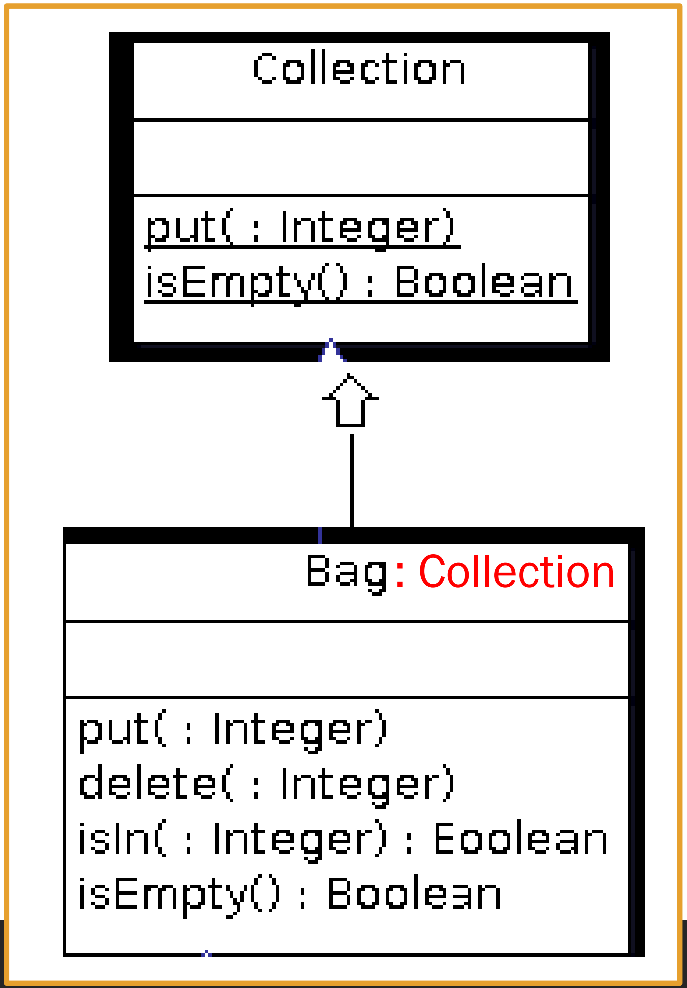
Derived classes must implement all the methods and fields of their ***parent***.

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 (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 under test.



Collection

put(: Integer)
isEmpty() : Boolean

Bag : Collection

put(: Integer)
delete(: Integer)
isIn(: Integer) : Eoolean
isEmpty() : Boolean

How Liskov SP Works.

// 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.

~~myBag.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.

Clients should not be forced to depend upon *interfaces* whose methods they won'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/

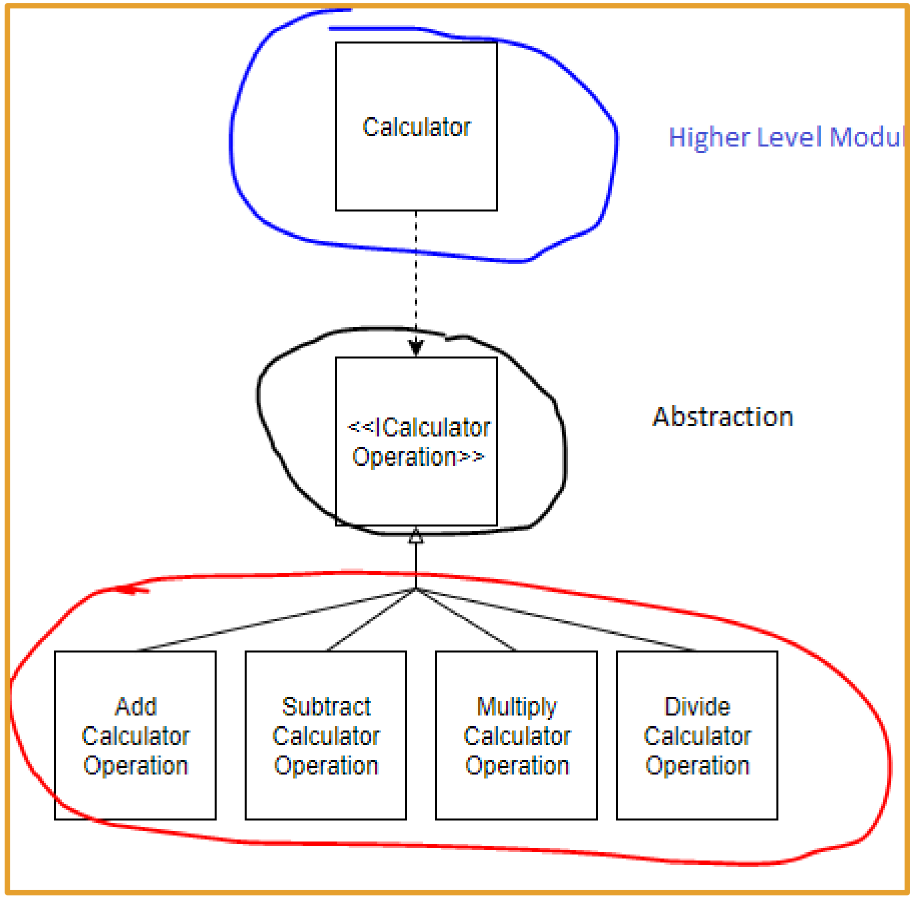
"High-level modules should not depend on low-level modules. Both should depend upon abstractions."

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

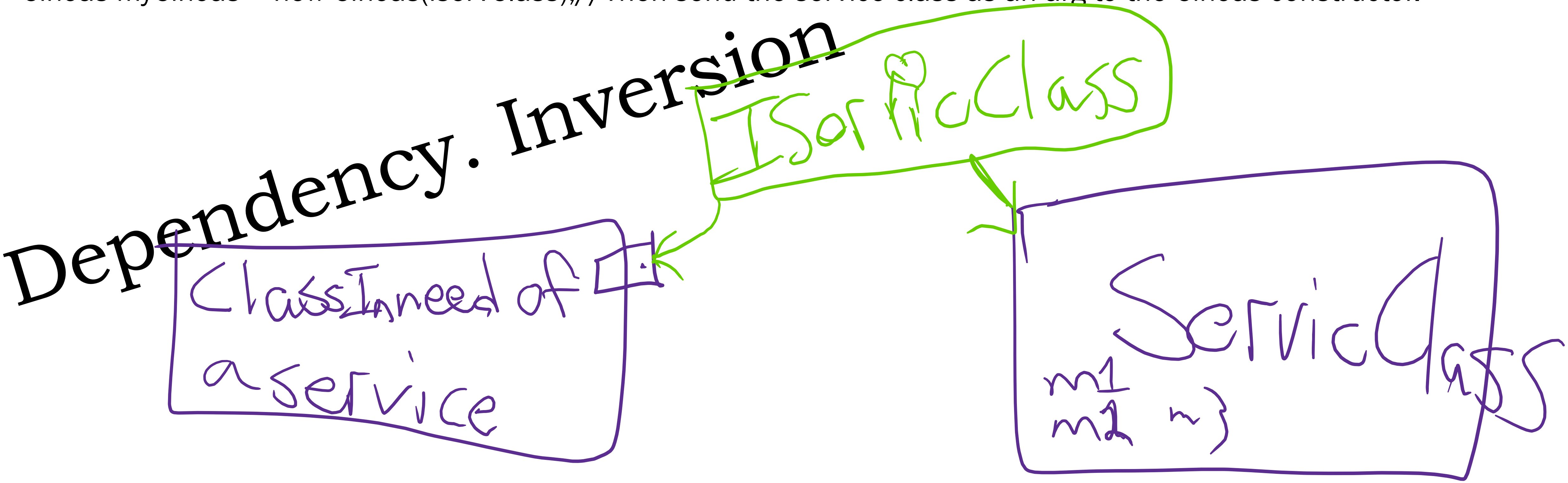
Low-level modules/classes deal with more detailed operations. They may deal with writing information to databases or passing messages to the operating system or services.

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 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

IServiceClass iservClass = new ServiceClass();// create the service class.
Cinoas myCinoas = new Cinoas(iservClass);//Then send the service class as an arg to the Cinoas constructor.
//NOW the private variable in Cinoas class can use the injected instance to access the Service class methods, etc
//WHEN TESTING
IServiceClass iservClass = new MockServiceClass();// create the mock service class that implements the IserviceClass.
// the methods in the MockServiceClass have a
Cinoas myCinoas = new Cinoas(iservClass);//Then send the service class as an arg to the Cinoas constructor.



Dependency. Inversion

A Class gets a service through Dependency Injection