**SOLID**

**Each of the various letters in the S.O.L.I.D. acronym is yet another acronym**

**SRP: Single Responsibility Principle**

*THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.*

**OCP: Open Closed Principle**

*SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION*.

**LSP: Liskov Substitution Principle**

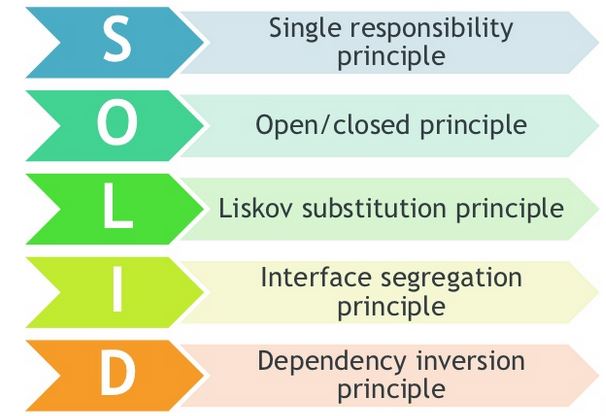
*FUNCTIONS THAT USE ... REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.*

**ISP: Interface Segregation Principle**

*CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE*

**DIP: Dependency Inversion Principle**

1. *HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS*
2. *ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS*



Single Responsibility Principle



There should only be one reason for a class to change.  
Don’t mix responsibilities, you can’t add any other tools to this tool you need to change the base

**Inheritance: Inheritance means that if you create a class Car with a public field Tank Size then you derive from it a class Supercar the last one has inherited the field Tank Size from Car**

**Polymorphism**: The ability to treat objects of different types in a similar manner

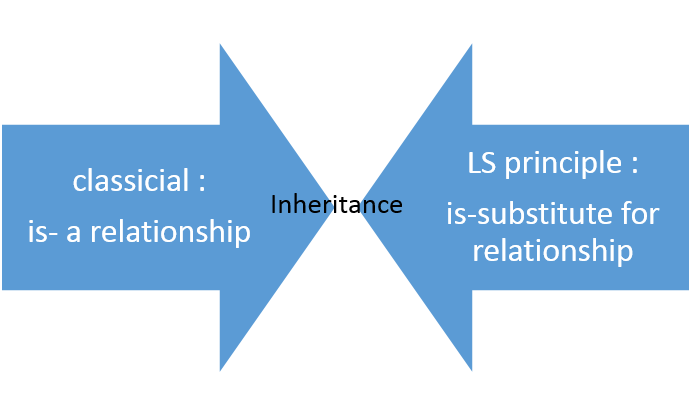
Abstraction – Using public and private to separate important exposed methods from private which is used to hide methods.

Open Closed Principle

Open for extension Closed for modification (should be able to extend from a class without having to modify it).  
  
To state the open closes principle very straightforward way you can say:  
- You should design modules that never change.   
- When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Liskov substitution principle 

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.



In short: if S is subset of T, an object of T could be replaced by object of S without impacting the program and bringing any error in the system. Let’s say you have a class Rectangle and another class Square. Square is as Rectangle, or in other words, it inherits the Rectangle class. So as the Liskov Substitution principle states, we should able to replace object of Rectangle by the object of Square without bringing any undesirable change or error in the system.

**Solve problem with No-Inheritance**

We can solve the above problem by following the below steps:

1.       Get rid of the AreaCalculator class.

2.       Let each shape define its own Area method.

3.       Rather than Square class will inherit Rectangle class, let us create a common abstract base class Shape and both classes will inherit that.

public  abstract class Shape

        {

        public abstract int Area();

        }

public class Rectangle :Shape

  {

      public  int Height { get; set; }

      public  int Width { get; set; }

      public override int Area()

      {

          return Height \* Width;

      }

  }

public class Square : Shape

  {

      public int Sides;

      public override int Area()

      {

          return Sides \* Sides;

      }

  }

Interface segregation principle

A client should not be forced to use an interface which is irrelevant to it.

* ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
* ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

**Problem:**

public interface IOrder

    {

        void Purchase();

        void ProcessCreditCard();

    }

    public class OnlineOrder : IOrder

    {

        public void Purchase()

        {

            //Do purchase

        }

        public void ProcessCreditCard()

        {

            //process through credit card

        }

    }

    public class InpersonOrder : IOrder

    {

        public void Purchase()

        {

            //Do purchase

        }

        public void ProcessCreditCard()

        {

            //Not required for inperson purchase

            throw new NotImplementedException();

        }

    }

**FIX**

public interface IOrder

{

void Purchase();

}

public interface IOnlineOrder

{

void ProcessCreditCard();

}

public class OnlineOrder : IOrder, IOnlineOrder

{

public void Purchase()

{

//Do purchase

}

public void ProcessCreditCard()

{

//process through credit card

}

}

public class InpersonOrder : IOrder

{

public void Purchase()

{

//Do purchase

}

}

Dependency inversion principle

* Do not write any tightly coupled code
  + Nightmare to maintain
  + Especially when the application gets bigger
  + There are 3 types to Dependency Injection,
    - Constructor injection,
    - Property injection
    - Method injection.

Constructor Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification(Imessenger pMessenger)

{

\_ iMessenger = pMessenger;

}

public void DoNotify()

{

\_ iMessenger.SendMessage();

}

}

Property Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification()

{

}

public IMessenger MessageService

{

private get;

set

{

\_ iMessenger = value;

}

}

public void DoNotify()

{

\_ iMessenger.SendMessage();

}

}

Method Injection

public class Notification

{

public void DoNotify(IMessenger pMessenger)

{

pMessenger.SendMessage();

}

}