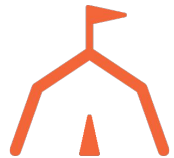
S.O.L.I.D. Principles

What is SOLID?

In OOP (Object-Oriented Programming), SOLID is an acronym for five design principles intended to make software design more understandable, flexible and more maintainable.

In other words, it’s a design philosophy adapted for OOP to make your code way easier to deal with.



What does SOLID stand for?

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

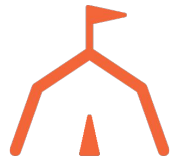
Single Responsibility Principle

SRP

Single responsibility principle

Every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

In other words, your class should be **responsible** for doing ONE thing and ONE thing only. If you find your class is providing multiple functionality, chances are your class is taken on TOO MUCH **responsibility** and should be refactored into multiple classes instead.



Single responsibility Example

```
public class Car
{
    0 references
    public void Accelerate(double accelerationRate)
    {
        // Accelerating logic
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }

    0 references
    public void GetDirections(string address)
    {
        // Get Directions
    }
}
```

In the example on the left, we have a car that has three methods, Accelerate, Brake, and Get Directions. When considering the single responsibilities of this car, one could make an argument that a Car is not actually responsible for Getting Directions.

Single responsibility Example Cont.

```
public class GPS
{
    0 references
    public void GetDirections(string address)
    {
        // Get Directions
    }
}
```

```
public class Car
{
    0 references
    public void Accelerate(double accelerationRate)
    {
        // Accelerating logic
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}
```

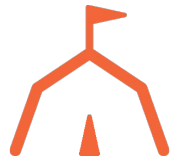
After some refactoring, we now have Two separate classes with their own responsibilities. A car is responsible for accelerating and braking, and a GPS for getting directions.

Single responsibility... why?

A reality in software development is you will often have to modify your code as business decisions and rules (business logic) changes. So, if you have classes doing just one thing, it is much easier to maintain by reducing the chances of breaking other classes that **depend** on your now changed class.

In contrast, if you have a class with many responsibilities, when changing said class, it could break several classes because they depend on the many responsibilities of your now modified class.

The latter is **NOT** desirable



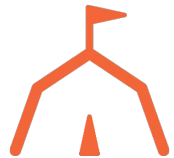
Open-Closed Principle

OCP

Open-Closed Principle

Classes should be open for extension, but closed for modification, that is, a class can allow its behavior to be extended without modifying its source code.

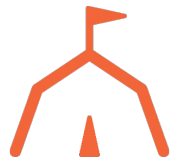
In other words, you should be able to add features/functionality to a class without directly affecting already existing methods logic. As a result, your class will be **CLOSED** to modification, and **OPEN** to extension (creating a new method(s))



Open-Closed Example

```
public class Car
{
    0 references
    public void Accelerate(double accelerationRate)
    {
        if (accelerationRate > 20)
        {
            // activate Turbo logic
        }
        else
        {
            // regular acceleration
        }
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
```

In this example, the business wants us to consider using turbo if the car's acceleration is greater than a certain threshold. The quick and dirty way to achieve this would be to modify the existing code for this car. However, if your car class is being referenced already, then you may be introducing a potential bug because this method is already in use. Therefore, this method should be **closed for modifying**.



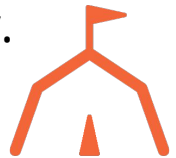
```
public interface ITurbo
{
    1 reference
    double AddTurbo();
}

1 reference
public class Turbo : ITurbo
{
    1 reference
    public double AddTurbo()
    {
        // Add some turbo
        return 2;
    }
}

0 references
public class Car
{
    1 reference
    private ITurbo _turbo = new Turbo();

    0 references
    public void Accelerate(double accelerationRate)
    {
        // regular acceleration
    }

    0 references
    public void TurboAccelerate(double accelerationRate)
    {
        var boost = _turbo.AddTurbo();
        // turbo acceleration logic
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}
```

Open-Closed Example - Composition

With composition, you could make use of interfaces that describes behavior that your car can reference. This idea promotes composition over inheritance.

The example on the left shows an Accelerate method that has **not been modified.** Instead, you can add a new method, thus **extending** the class.

Also, if you notice, the turbo logic has its own class, promoting single responsibility.



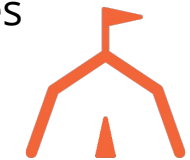
Open-Closed Example - Inheritance

```
public class Car
{
    0 references
    public virtual void Accelerate(double accelerationRate)
    {
        // regular acceleration
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}

0 references
public class TurboCar : Car
{
    0 references
    public override void Accelerate(double accelerationRate)
    {
        // some turbo logic
    }
}
```

With inheritance, you can make your Accelerate method virtual, then make a new class, and have this class override Accelerate.

By using Inheritance in this example, you are NOT modifying the original class at all and strictly enforcing open-closed.

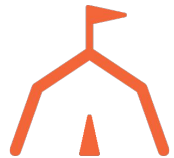
However, if you find that you are creating multiple layers of inheritance to enforce open-closed, you may be creating layers of complexity. The implications of these layers is; if for you need to change a parents classes logic, this could break classes down the inheritance chain.



Open-Closed... why?

What this solves is very similar to the problem we run into with Single Responsibility. You want to avoid making changes to your code base that could potentially break code elsewhere.

The more modern approach to Open-Closed is to favor **composition** over **inheritance**: By using composition, you are also promoting single responsibility which helps define your classes and keeps your code more organized as a result.



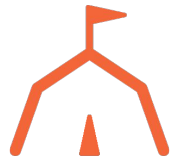
Liskov Substitution Principle

LSP

Liskov Substitution Principle

If **S** is a child type of **T**, then objects of type **T** may be *replaced* with objects of type **S** (i.e. an object of type **T** may be substituted with ANY object of a subtype **S**)

This is just stating that if a subclass inherits from a class, its child class should be able to be USED as a substitute for its parent class. In addition, if ALL child classes adhere to this principle, then each child class can easily substitute for its parent.



```
public class Car
{
    0 references
    public virtual void Accelerate(double accelerationRate)
    {
        // regular acceleration
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}

0 references
public class TurboCar : Car
{
    0 references
    public override void Accelerate(double accelerationRate)
    {
        // some turbo logic
    }
}

0 references
public class OffRoadCar: Car
{
    0 references
    public void OffRoadDriveAccelerate(double accelerationRate)
    {
        // some acceleration logic
    }
}
```

Liskov Substitution Example

Whenever you inherit from a class, again said child classes should be able to be used as a substitute for it's parent.

For example, TurboCar is overriding Car and also providing its own Accelerating logic. Therefore if we were to use TurboCar as a replacement for Car, we are doing just that because we are also **Substituting** the Accelerate Method.

```
public class Car
{
    1 reference
    public virtual void Accelerate(double accelerationRate)
    {
        // regular acceleration
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}

1 reference
public class OffRoadCar: Car
{
    0 references
    public void OffRoadDriveAccelerate(double accelerationRate)
    {
        // some acceleration logic
    }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Car car = new OffRoadCar();

        // This is calling the Base Car classes Accelerate!
        car.Accelerate(15);
    }
}
```

Liskov Substitution Example Cont.

In the following example, the OffRoadCar **cannot** be a full substitute for Car because the Accelerate Method is not being substituted.

In the Main method, whenever we instantiate a new OffRoadCar into memory, when calling the Accelerate Method, we are instead calling the Car class's method because it was never overridden in the OffRoadCar class.

This is especially hard to catch if you have a collection of type Car, and each element in the collection needs to Accelerate and each element is a subtype of car. (ie, a List of { TurboCar, OffRoadCar })

Liskov Substitution Example Cont.

In order to remedy this situation, simply make sure you are guaranteeing that your subclasses can be a replacement for its parent. In the example below, the Accelerate method is overriding, and we are using our own logic to determine what can make an off roading acceleration different, then reference the base to determine the car's acceleration.

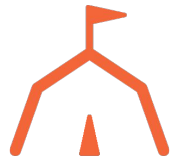
```
public class OffRoadCar: Car
{
    2 references
    public override void Accelerate(double accelerationRate)
    {
        // some Off roading logic
        var resistance = 10;
        base.Accelerate(accelerationRate - resistance);
    }
}
```

Liskov Substitution... why?

When solving problems using OOP, It's important that your classes are as modular and maintainable as possible. Part of that is to make sure your inheritance does NOT deviate too far away from its intended use.

In a way, this principle helps reinforce single responsibility by making sure any sub classes also have that same or close to a single responsibility.

This is just a simple example but helps explain the principle, this situation can be manifested in a broad variety of ways, and is not always easy to identify.



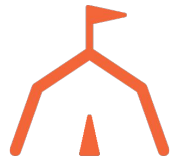
Interface Segregation Principle

ISP

Interface Segregation Principle

No client should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller more specific ones so that clients will only have to know about methods that are of interest to them.

Simply put, when creating interfaces and a class implements an interface, make sure that the methods you are implementing are USEFUL to the class. If the additional methods in the interface do **not** serve the class, consider refactoring or **segregating** your interface into multiple interfaces.



```
public interface ITurbo
{
    1 reference
    double AddTurbo();
    0 references
    bool TryHover();
}

1 reference
public class Turbo : ITurbo
{
    1 reference
    public double AddTurbo()
    {
        // Add some turbo
        return 2;
    }

    0 references
    public bool TryHover()
    {
        return false;
    }
}
```

Interface Segregation Example

The business wants to try adding a hover feature to its cars using turbo fuel. Given that this is kind of related to our current turbo interface, you could make an argument that this try hover behavior could exist in ITurbo.

However, your previous cars that used turbo will not have the ability to hover, so the current turbo class will now be forced to implement a method it does not care about.

In order to correct this problem, you should break the ITurbo interface into two separate interfaces.

```
public interface IAddTurbo
{
    1 reference
    double AddTurbo();
}

0 references
public interface ITryHover {
    0 references
    bool TryHover();
}

1 reference
public class Turbo : IAddTurbo
{
    1 reference
    public double AddTurbo()
    {
        // Add some turbo
        return 2;
    }
}
```

Interface Segregation Example Cont.

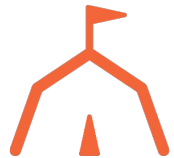
By breaking your Turbo interface into separate interfaces, you are now being more specific on what behavior you want your class(es) to implement.

Now the current Turbo class will not be forced to implement a hovering method. In Addition, if you create a new turbo class that needs to implement both interfaces, you can just do so because classes can implement as many interfaces as you want.

Interface Segregation... why?

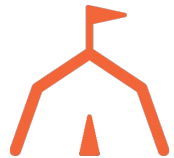
When designing interfaces, it may be tempting to add many methods to it because they all share the same responsibility. However, when composing classes with the selected interfaces, it's very important that the class in use does not implement behavior it will never use. This can lead to unpredictable bugs.

Because a class can implement many interfaces, it's encouraged to make your interfaces as lean as possible.



Dependency Inversion Principle

DIP

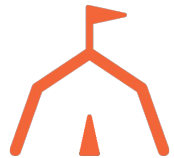


Dependency Inversion Principle

A form of decoupling modules (classes).

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces)
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions

When we say abstractions, we just mean interfaces in this case, details will be the classes implementing the interface. This principle is basically saying your classes should NOT care about the implementation details of its dependencies, AND our dependencies can be replaced by different implementations of said interfaces as a result.



Dependency Inversion explained

```
public class Car
{
    // I am a dependency!
    1 reference
    private Turbo _turbo = new Turbo();

    0 references
    public void Accelerate(double accelerationRate)
    {
        // regular acceleration
    }

    0 references
    public void TurboAccelerate(double accelerationRate)
    {
        var boost = _turbo.AddTurbo();
        // turbo acceleration logic
    }

    0 references
    public void Brake(double decelerationRate)
    {
        // Breaking logic
    }
}
```

In order to understand Dependency Inversion, you must first understand the idea of a dependency. A dependency is a class that is depended on by another class. Dependencies are created as a side effect of the other solid principles.

Notice the comment I am creating a dependency on the left. Car depends on Turbo in order to add turbo, as a result, Car depends on Turbo.

Dependency Inversion example

```
public class Car
{
    // I am a dependency!
    1 reference
    private ITurbo _turbo = new Turbo();
```

One of the problems that dependency inversion solves is caring only for abstractions, not the implementations. In other words, instead of storing a Turbo as a private field, we can instead use an ITurbo. ITurbo is considered an abstraction of Turbo.

In the example above, we changed our type from Turbo to ITurbo, thus caring only about abstraction. The only problem is we are STILL forcing our car class to CARE about the type of Turbo we are going to use by assigning _turbo a new Turbo().

Dependency Inversion example cont

```
public class Car
{
    // I am a dependency, and an interface, my
    // concrete implementation of ITurbo can
    // be whatever I want!
    2 references
    private ITurbo _turbo;

    // Dependency is being injected into the constructor!
    0 references
    public Car(ITurbo turbo)
    {
        _turbo = turbo;
    }
```

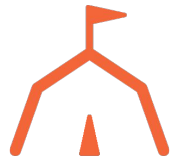
Next thing to do is store a type of ITurbo into the field without instantiating a Turbo. A way of doing this is to use whats called Dependency Injection, where we inject our dependency into the constructor of the car class, using the ITurbo type instead of the Turbo type.

Dependency Inversion example cont

```
public class Turbo : ITurbo
{
    1 reference
    public double AddTurbo()
    {
        // Add some turbo
        return 2;
    }
}

1 reference
public class SuperTurbo : ITurbo
{
    1 reference
    public double AddTurbo()
    {
        // Add some turbo
        return 4;
    }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        var turboCar = new Car(new Turbo());
        var superTurboCar = new Car(new SuperTurbo());
    }
}
```

By passing an abstracted type (ITurbo) into the constructor instead of the concrete type (Turbo), we are now able to pass in ANY class that is implementing ITurbo into car because of polymorphism. This is a decoupling technique, now the Turbo class is NO LONGER coupled to the Car class.

In the example on the left, we have two car classes, both of them using a different implemented type of ITurbo. Since this makes our dependencies not bound to its calling class, our dependence has been “inverted.”

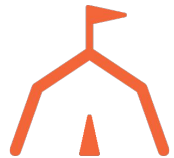


Dependency Inversion... why?

This principle makes our code way more modular. When writing your code with abstraction in mind, you can swap out implementations without breaking your code because each implementation is just that, an implementation of the interface.

In modern OOP, it is more common to design your classes with composition instead of inheritance. By using dependency inversion, it's easier to compose your class by passing in dependencies instead of inheriting from a base class.

Dependency Injection in dotnet core is a much larger topic and will be covered in separate slides.



Summary

You should now be able to identify and explain each letter of SOLID:

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SOLID is a series of software engineering principles designed and used to make your code more understandable, flexible and maintainable. The more applications you create/maintain, the more intuitive the SOLID principles will start to feel!