SOLID PRINCIPLES

SOLID principles are object-oriented design concepts relevant to software development, they are a set of rules and best practices to follow while designing a class structure.

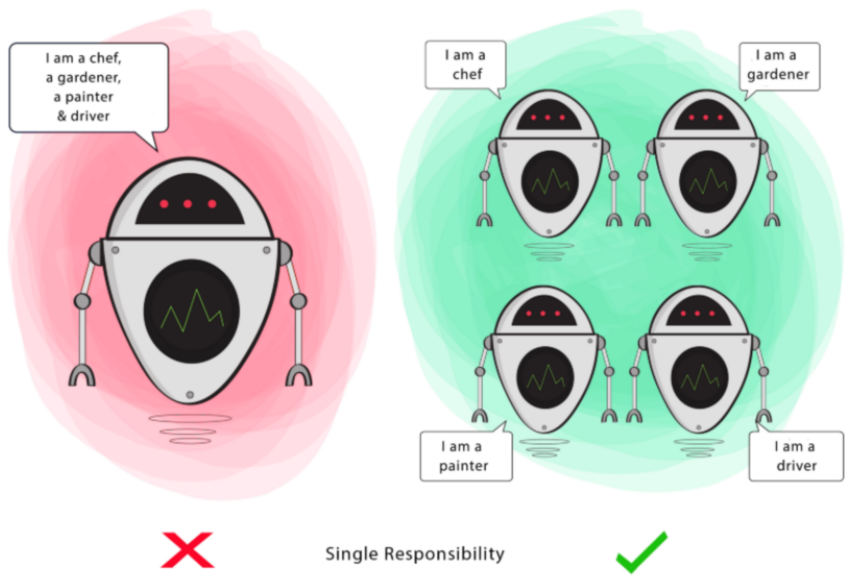
The SOLID principles were introduced by Robert C. Martin in his 2000 paper "Design Principles and Design Patterns." These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these five principles have revolutionized the world of object-oriented programming, changing the way that we write software.

SOLID is an acronym for five other class-design principles namely:

1. Single Responsibility Principle,
2. Open-Closed Principle,
3. Liskov Substitution Principle,
4. Interface Segregation Principle,
5. Dependency Inversion Principle.

Single Responsibility Principle

A class should have a single responsibility



This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to single responsibility.

If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities could affect the other ones without you knowing.

Goal
This principle aims to separate behaviors so that if bugs arise as a result of your change, it won't affect other unrelated behaviors.

IMPLEMENTATION

```
public class BankAccount
{
    public BankAccount()  {}

    public string AccountNumber { get; set; }
    public decimal AccountBalance { get; set; }

    public decimal CalculateInterest()
    {
        // Code to calculate Interest
    }
}
```

Here, the BankAccount class comprises the properties of the account and computes the interest of the account. Now, look at the change Request we received from the business:

- Please implement a new Property AccountHolderName.
- Some new rules have been incorporated to calculate interest.

These are very different kinds of change requests. One is changing features while the other one is affecting the functionality. We have 2 separate kinds of reasons to change one class, which violates SRP.

Let’s use SRP to resolve this violation. Let us look at the code below:

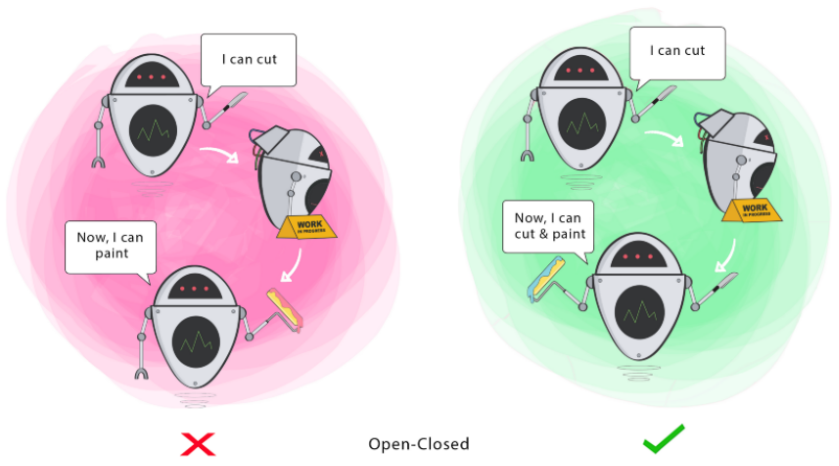
```
public interface IBankAccount
{
    string AccountNumber { get; set; }
    decimal AccountBalance { get; set; }
}

public interface IInterstCalculator
{
    decimal CalculateInterest();
}

public class BankAccount : IBankAccount
{
    public string AccountNumber { get; set; }
    public decimal AccountBalance { get; set; }
}

public class InterstCalculator : IInterstCalculator
{
    public decimal CalculateInterest(IBankAccount account)
    {
        // Write your logic here
        return 1000;
    }
}
```

Our BankAccount class is just responsible for the properties of the bank account. If we wish to incorporate any additional business rule for the Calculation of Interest, we aren't required to change the BankAccount class. Also the InterestCalculator class needs no changes, in case we have to add a new Property AccountHolderName. Thus, this is the implementation of the Single Responsibility Principle.

Open-Closed Principle

Classes should be open for modification but closed for extension.

Modification means changing the code of an existing class, and extension means adding new functionality.



Changing the current behavior of a Class will affect all the systems using that Class.

So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.

Goal

This principle aims to extend a Class's behavior without changing the existing behavior of that Class. This is to avoid causing bugs wherever the Class is being used.

IMPLEMENTATION

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

public class Circle{
 public double Radious {get; set;}
}

public double getArea (object[] shapes){
 double totalArea = 0;

 foreach(var shape in shapes){
  if(shape is Rectangle){
   Rectangle rectangle = (Rectangle)shape;
   totalArea += rectangle.Width * rectangle.Height;
  }
  else{
   Circle circle = (Circle)shape;
   totalArea += circle.Radious * circle.Radious * Math.PI;
  }
 }
}
```

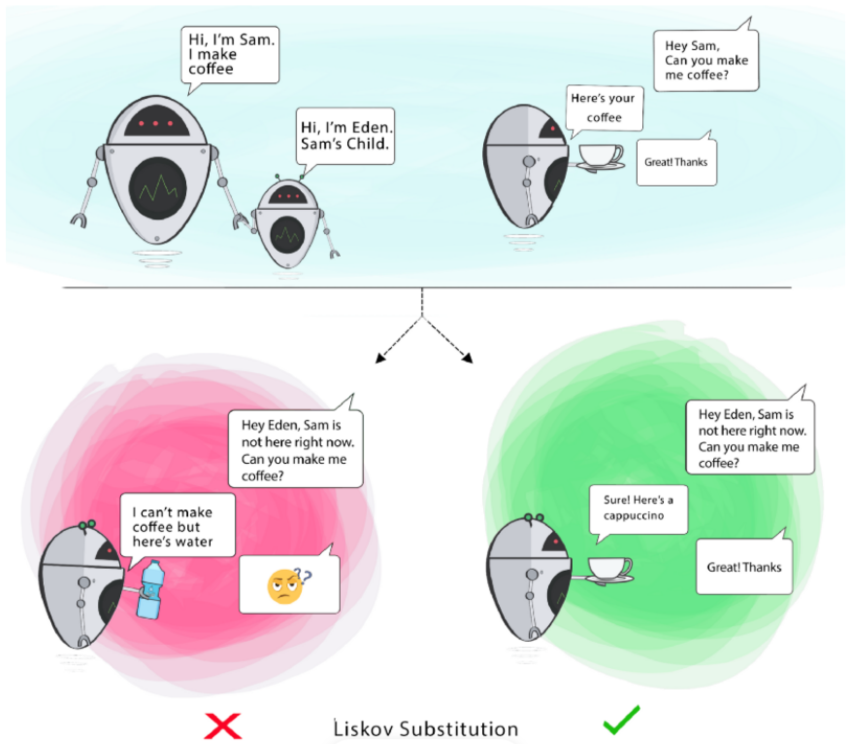
This violates OCP because If we have to calculate another type of object (say, Trapezium) then we’ve to add another condition. But from the rules of OCP, we have an idea that Software entities should be closed for modification. Hence it violates OCP. Let’s try to resolve this violation by using OCP.

```
public abstract class shape{
 public abstract double Area();
}

public class Rectangle : shape{
 public double Width {get; set;}
 public double Height {get; set;}

 public override double Area(){
  return Width * Height;
 }
}

public class Circle : shape{
 public double Radious {get; set;}

 public override double Area(){
  return Radious * Radious * Math.PI;
 }
}

public double getArea (shape[] shapes){
 double totalArea = 0;
 foreach(var shape in shapes){
  totalArea += shape.Area();
 }

 return totalArea;
}
```

Now if we have to calculate another type of object, we don’t have to alter logic (in getArea()), we just have to add another class like Rectangle or Circle.

Liskov Substitution Principle

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

When a child Class cannot perform the same actions as its parent Class, this can cause bugs.

If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a child. The child Class should be able to do everything the parent Class can do. This process is called Inheritance.

The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.

The picture shows that the parent Class delivers Coffee(it could be any type of coffee). It is acceptable for the child Class to deliver Cappuccino because it is a specific type of Coffee, but it is NOT acceptable to deliver Water.

If the child Class doesn't meet these requirements, it means the child Class is changed completely and violates this principle.

Goal

This principle aims to ensure consistency so that the parent's class or its child class can be used in the same way without any errors.

IMPLEMENTATION

Supposedly we are designing the payment module for our eCommerce website. Customers order products on the site and pay with the payment instruments like a credit card or a debit card.
When a customer provides their card details, we want to

- validate it,
- run it through a third-party fraud detection system,
- and then send the details to a payment gateway for processing.

Here is an example of a class structure which violates LSP:

```
public class Project
{
    public Collection<ProjectFile> ProjectFiles { get; set; }

    public void LoadAllFiles()
    {
        foreach (ProjectFile file in ProjectFiles)
        {
            file.LoadFileData();
        }
    }

    public void SaveAllFiles()
    {
        foreach (ProjectFile file in ProjectFiles)
        {
            if (file as ReadOnlyFile == null)
                file.SaveFileData();
        }
    }
}
```

```
public class ProjectFile
{
    public string FilePath { get; set; }

    public byte[] FileData { get; set; }

    public void LoadFileData()
    {
        // Retrieve FileData from disk
    }

    public virtual void SaveFileData()
    {
        // Write FileData to disk
    }
}


public class ReadOnlyFile : ProjectFile
{
    public override void SaveFileData()
    {
        throw new InvalidOperationException();
    }
}
```

Here the Project class has been changed to include two collections rather than one. One collection comprises all of the files in the project and one holds references to writable files only. The LoadAllFiles method loads data into all of the files in the AllFiles collection. As the files in the WriteableFiles collection will be a subset of the same references, the data will be visible via these also. The SaveAllFiles method has been replaced with a method that saves only the writable files.

The ProjectFile class now comprises just one method, which loads the file data. This method is needed for both writable and read-only files. The new WriteableFile class extends ProjectFile, incorporating a method that saves the file data. This reversal of the hierarchy means that the code now complies with the LSP.

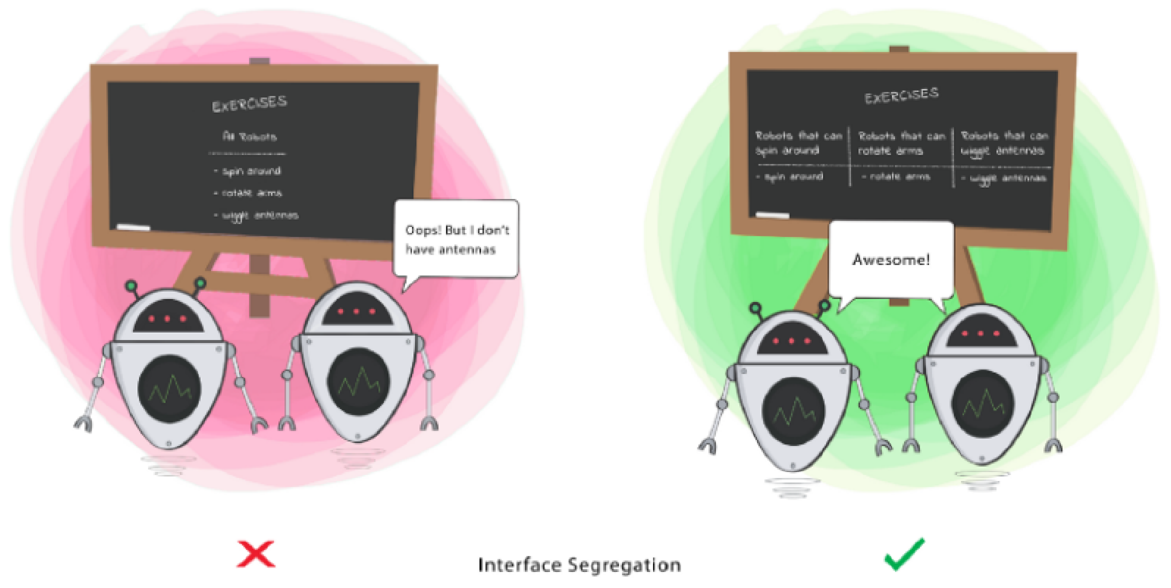
The refactored code is as follows:

```
public class Project
{
    public Collection<ProjectFile> AllFiles { get; set; }
    public Collection<WriteableFile> WriteableFiles { get; set; }

    public void LoadAllFiles()
    {
        foreach (ProjectFile file in AllFiles)
        {
            file.LoadFileData();
        }
    }

    public void SaveAllWriteableFiles()
    {
        foreach (WriteableFile file in WriteableFiles)
        {
            file.SaveFileData();
        }
    }
}
```

```
public class ProjectFile
{
    public string FilePath { get; set; }

    public byte[] FileData { get; set; }

    public void LoadFileData()
    {
        // Retrieve FileData from disk
    }
}


public class WriteableFile : ProjectFile
{
    public void SaveFileData()
    {
        // Write FileData to disk
    }
}
```

Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.



When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

A Class should perform only actions that are needed to fulfill its role. Any other action should be removed completely or moved somewhere else if it might be used by another Class in the future.

Goal

This principle aims at splitting a set of actions into smaller sets so that a class executes only the set of actions it requires.

Implementation

This is a simple principle to understand and apply, so let's see an example.

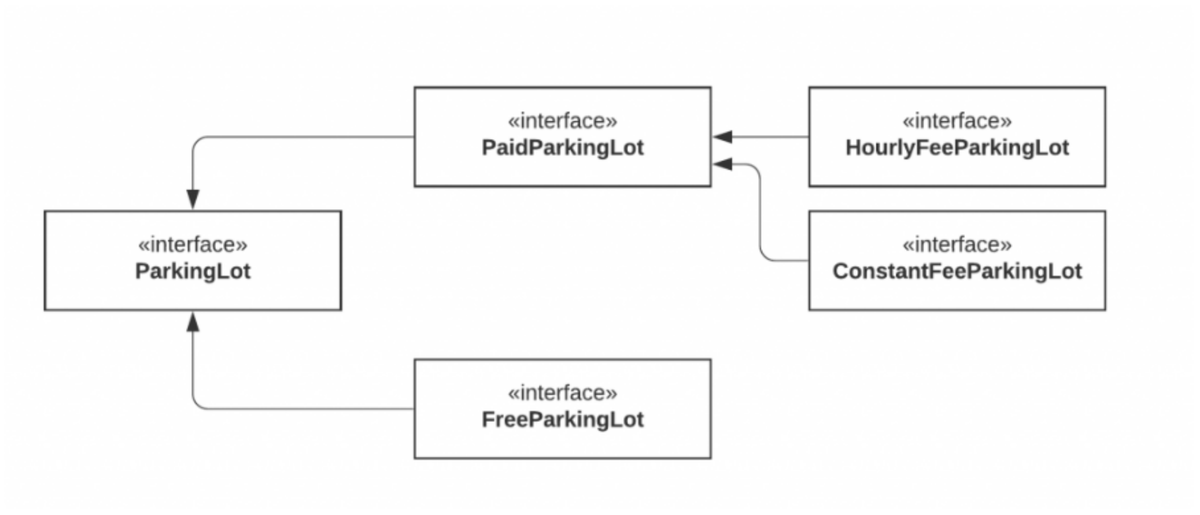
```
public interface ParkingLot {

    void parkCar(); // Decrease empty spot count by 1
    void unparkCar(); // Increase empty spots by 1
    void getCapacity(); // Returns car capacity
    double calculateFee(Car car); // Returns the price based on number of hours
    void doPayment(Car car);
}

class Car {

}
```

We modeled a very simplified parking lot. It is the type of parking lot where you pay an hourly fee. Now consider that we want to implement a parking lot that is free.

```
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {

    }

    @Override
    public void unparkCar() {

    }

    @Override
    public void getCapacity() {

    }

    @Override
    public double calculateFee(Car car) {
        return 0;
    }

    @Override
    public void doPayment(Car car) {
        throw new Exception("Parking lot is free");
    }
}
```

Our parking lot interface was composed of 2 things: Parking related logic (park car, unpark car, get capacity) and payment related logic.

But it is too specific. Because of that, our FreeParking class was forced to implement payment-related methods that are irrelevant. Let's separate or segregate the interfaces.



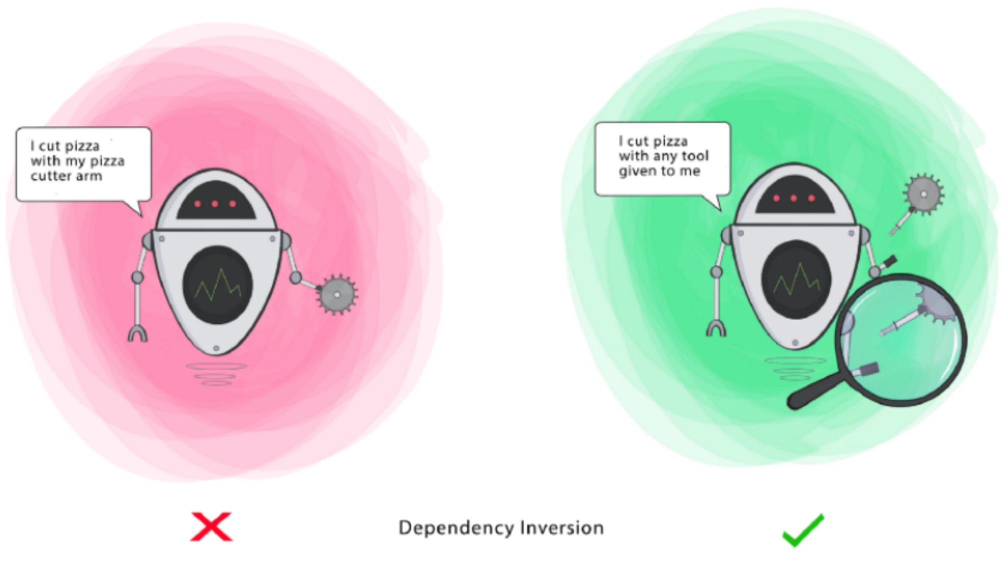
We've now separated the parking lot. With this new model, we can even go further and split the **PaidParkingLot** to support different types of payment.

Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.

Dependency Inversion Principle.

High Level modules should not depend on low-level modules. Both should depend on the abstraction.

Abstractions should not depend on details. Details should depend on abstractions.


I cut pizza with my pizza cutter arm
I cut pizza with any tool given to me
Dependency Inversion

Firstly, let’s define the terms used here more simply

High-level Module(or Class): Class that executes an action with a tool.

Low-level Module (or Class): The tool that is needed to execute the action

Abstraction: Represents an interface that connects the two Classes.

Details: How the tool works

This principle says a Class should not be fused with the tool it uses to execute an action. Rather, it should be fused to the interface that will allow the tool to connect to the Class.

It also says that both the Class and the interface should not know how the tool works. However, the tool needs to meet the specification of the interface.

Goal
The principle aims at reducing the dependency of a high-level class on the low-level class by introducing an interface.