* **Why Design Principle ?**

Well it all came when **Robert Martin** addressed three major issue which software developer face time to time . they are as follows ,

* **Rigidity**

When a developer writes a rigid code , whenever anyone changes that code then it will definitely effect another part of the code.

* **Fragility**

If a developer writes a fragile code then whenever a new code is written that automatically breaks the old code .

* **Immobility**

Basically effects the reusability of the code .

* **So what’s the solution this ??**

Well its simple the most popular approach is to use **SOLID principle.**

* **So what is solid principle ??**

So SOLID stands for ,

**S:Single responsibility principle.**

**O:open close Principle**

## L: [Liskov's Substitution Principle](https://www.oodesign.com/liskov-s-substitution-principle.html)

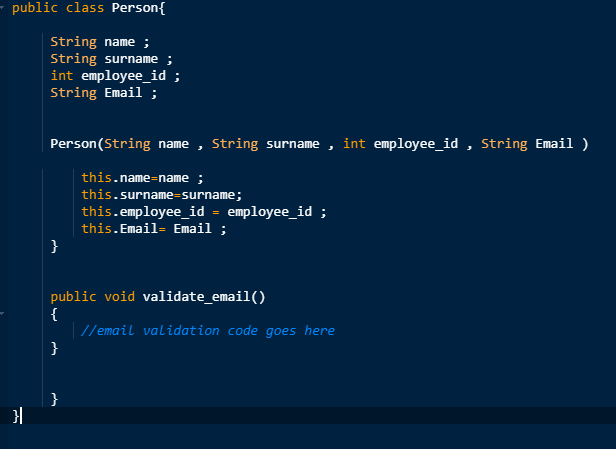
## I: [Interface Segregation Principle](https://www.oodesign.com/interface-segregation-principle.html)

**D: Dependency Inversion Principle**

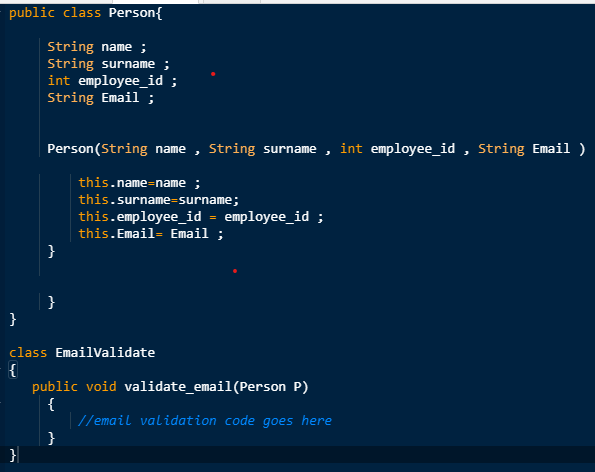
* **Single responsibility principle :**

Single responsibility is the concept of a Class doing one specific thing (responsibility) and not trying to do more than it should, which is also referred to as High Cohesion.

The most effective way to break applications is to create God classes.



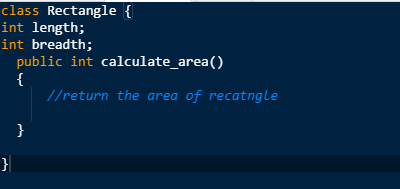
The above example shows an example of god class basically the class is related to person but it also has another functionality to validate email .



* **Open close principle**

This principle states that a new functionality to existing code should be added in such a way that there should be minimum modification in the existing code .

So lets suppose we have a customer who wants us to calculate the area of rectangle . its an easy task .

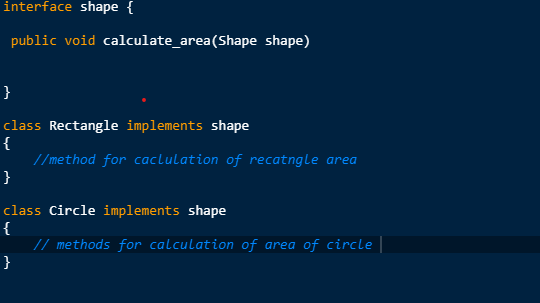


Its simple right ?

Now the customer has another demand or extension in functionality that he wants us to include calculation of area of the circle .

Things will start getting difficult with every modification .

One common solution to this would be create and interface shape and extend it by doing this we will always write a new class without even modifying the existing class.



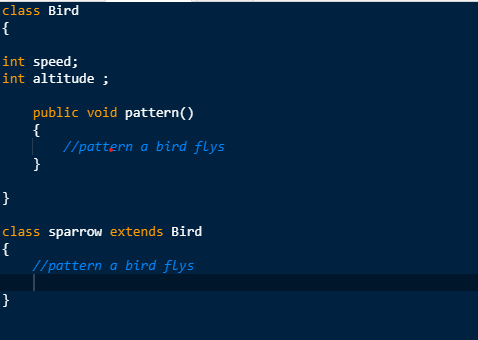
* Liskov’s Principle :

We know “is a relationship “ often called as an inheritance .

“penguin” is a bird right ??

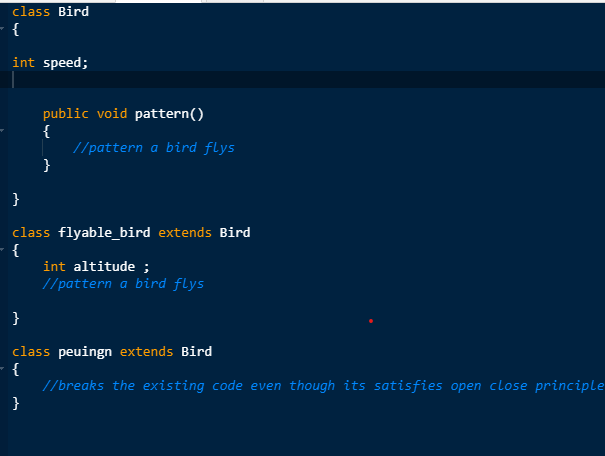
Ofcourse it is a bird .

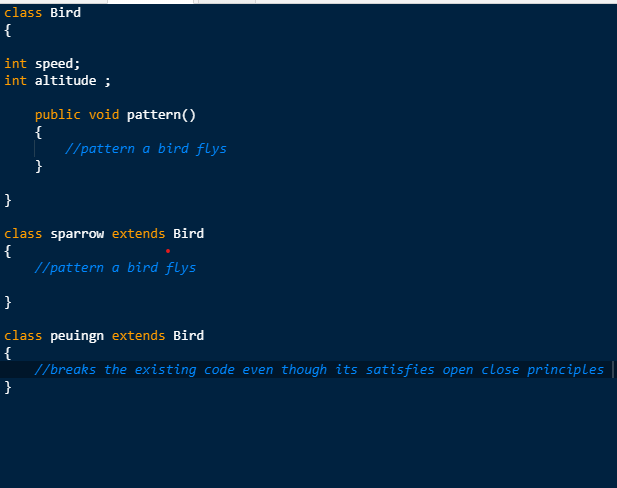
So we are writing a class functionality of pattern the bird fly .



goes well till the penguin class comes into picture and breaks the existing code as penguin cant fly the improper inheritance satisfies the open close principle but breaks the existing code

So the correct way of implementation is as follows .

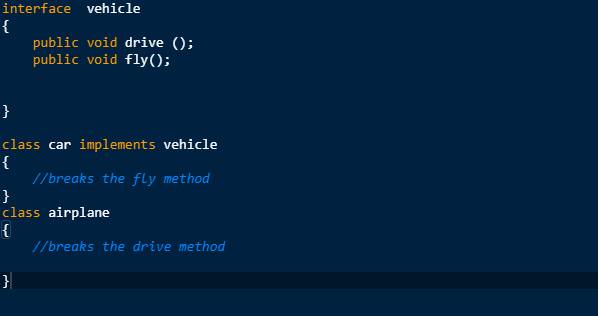




* Interface segregation principle

The idea is a client should not be forced to implement a fat Interface or a polluted interface which they don’t use .

For example see the below code .

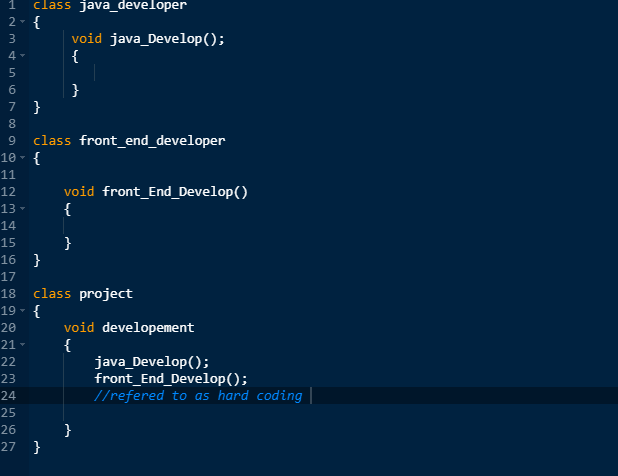


* **Dependency inversion Principle :**

Dependency Inversion is one of the last principles we are going to look at. The principle states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Lets see the below example



In the above example we have a dependency of two classes in project class .

So we can make an interface ProjectDevelopment and it contains devlop method and while implementing that interface we just have to override that method and directly call that method from Project class to reduce dependency .