1. **Single responsibility** : a class should have only single responsibility and should have one reason for change

Example:

For Violate Single responsibility:

We will create a Resume class that has methods get & set the technology & years of experiences in each resume, and search the resume from the repository.

However, the Resume class violates Single Responsibility Principle because it have two responsibilities. First, get & set properties. Second, it searches for the resume in the repository.

For Single responsibility:

I decouple the two responsibilities. The resume class will only be responsible for getting & setting properties.

I create ResumeRepository is a single responsibility for manipulating Resume in table Resume in database

1. **Open closed principle**: a class should be open for extension, but closed for modifications

Example:

For violate Open closed principle:

I have classes that are Circle, Rectangle /ˈrek.tæŋ.ɡəl/, AreaCalculator that have two methods calculating area for Circle, Rectangle.

After time I add one more class pentagon /ˈpen.tə.ɡən/ then I want to calculate area for that class. Then I add one more method calculateAreaPentagon then consequently I violate Open Closed Principle.

For Open closed principle:

I have an interface Shape that has abstract method calculateArea and three class Rectangle, Circle, AreaCalculator which has only one method calculateShape with parameter is Shape. Two classes Rectangle, Circle implement Shape.

After time I add one more class pentagon that implement Shape then I want to calcuclate Area for pentagon . I don’t have to modify class AreaCalculator .

1. **Liskov substitution** /ˌsʌb.stɪˈtʃuː.ʃən/ **principle**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of program.

For violate Liskov:

I have three classes Bird, Pigeon /ˈpɪdʒ.ən/, Penguin /ˈpeŋ.ɡwɪn/ . Class Bird have method fly(). Classes Pigeon , Penguin inherit from Bird. But Penguin cannot fly so

it violate Liskov substitution.

For Liskov:

In class Bird. I remove method fly(). Then I create more two classes FlyingBird, Swimming Bird. First one has a fly() method, second one has a swim() method.

Then class Pigeon inherit from FlyingBird, class Penguin inherit from SwimmingBird.

1. **Interface segregation** /ˌseɡ.rəˈɡeɪ.ʃən/: Clients should not be forced to depend upon interfaces that they do not use, it would be bad for you to force the client to depend on a certain thing, which they don’t need.

For violate interface segragation :

I create a RestaurantInterface that has methods acceptOnlineOrder, acceptTelephoneOrder, acceptWalkInCustomerOrder, payOnline, payInPerson.

Then I create class OnlineCustomerImpl that implements RestaurantInterface.

But class OnlineCustomerImpl only supports acceptOnlineOrder, payOnline, not support methods acceptTelephoneOrder, acceptWalkInCustomerOrder, payInPerson then it violates interface segregation.

For interface segragation:

Instead of Create RestaurantInterface. I create two interfaces that are OrderInterface, PaymentInterface. OrderInterface has a abstract method placeOrder. PaymentInterface has a abstract method payForOrder.

Then I create OnlineCustomerImpl, WalkInCustomerImpl, TelephoneCustomerImpl. Each class implement action of it’s own.

1. **Dependency inversion principle:** the high-level module must not depend on the low-level module, but they should depend on abstractions.

For violating Dependency Inversion:

I have a EmailService has a method sendEmail()

UserController uses directly EmailService to send email.

It violate Dependency inversion because:

* **UserController depends on EmailService. When we modify code in EmailService, ex: add more params to constructor. It will effect to class UserController.**
* **When user wants to change the way to send message like sending message via sms or facebook.**
* **It hard to write unit test for UserController because it depends directly on EmailService.**

For Dependency Inversion:

I create an interface MessageService that has an abstract method sendMessage

I create two classes EmailService, SmsService implements MessageService.

In UserController I create contructor with param MessageService.

Then each time I want to use EmailService or SmsService. I just create it and then pass it to UserController