Single Responsibility principle

A class should have only one reason to change,meaning it should have only one job or responsibility

[Devaraj-Umapathi/Solid Principles/src/SingleResponsibilityWithViolation/SRPViolation.java at SOLID-Principles · Devaraj-Umapathi/Devaraj-Umapathi](https://github.com/Devaraj-Umapathi/Devaraj-Umapathi/blob/SOLID-Principles/Solid%20Principles/src/SingleResponsibilityWithViolation/SRPViolation.java)

Salary calculation not work of employee class it is work of other class

Better wy handle is to have separate clss that is sarygenerator and reportgenrator if nay change in cculation we modify that class

Open/Closed Principle:

Software entitens should be open for extennstion and closed for modification .This llow adding new feature without altering the existing code

Accept shape as parmeter ,radius length, brdth in inside u use if clss to check whether the shape is circle it perform some calculation and rectangle in elseif and and square in another elsif if in future if we try to add new shape then we need to add another elseif for that

So define interface for calacuate area

And circle class implement shape and we area method and define

Likewise we do if new shape create and implement shape

interface and define area so no modifaction of old code we create new clss for this

Liskov Substitution Principle:

Objects of superclass should be replaceable with objects of subclass without affecting the correctness of the program

[Devaraj-Umapathi/Solid Principles/src/LiskovSubstitutionWithViolation/LSPViolation.java at SOLID-Principles · Devaraj-Umapathi/Devaraj-Umapathi](https://github.com/Devaraj-Umapathi/Devaraj-Umapathi/blob/SOLID-Principles/Solid%20Principles/src/LiskovSubstitutionWithViolation/LSPViolation.java)

In inheritance we have parent and child .when child class inherit prent method its funvtion not affected by parent

Exaample we have parent class bird with function fly

And another child aand sprrow inhert bird nd it cana use the fly method or override

If penquin extends bird but normally child penguin not fly so we throw error when we try to call fly of penguin

Hey we define separate interface one for bird with make sound functions and flting bird with fly function by this penguin only implemts bird interface not flyingbird

[Devaraj-Umapathi/Solid Principles/src/LiskovSubstitutionWithoutViolation at SOLID-Principles · Devaraj-Umapathi/Devaraj-Umapathi](https://github.com/Devaraj-Umapathi/Devaraj-Umapathi/tree/SOLID-Principles/Solid%20Principles/src/LiskovSubstitutionWithoutViolation)

Interface segregation Principle

Clients should not be forced to depend on interfaces they do not use

[Devaraj-Umapathi/Solid Principles/src/InterfaceSegregationWithViolation/ISPViolation.java at SOLID-Principles · Devaraj-Umapathi/Devaraj-Umapathi](https://github.com/Devaraj-Umapathi/Devaraj-Umapathi/blob/SOLID-Principles/Solid%20Principles/src/InterfaceSegregationWithViolation/ISPViolation.java)

In worker interface we have work and eat function

Human class implements worker and do implementation

But robot implement worker but robot can’t eat so to handle that

Make separate interface

* Interface Segregation Principle (ISP): This principle states that no client should be forced to depend on methods it does not use. In other words, interfaces should be small and fine-grained, tailored to specific clients, so that implementing classes do not end up with unnecessary or irrelevant methods.
* Liskov Substitution Principle (LSP): This principle tells us that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. That means derived classes must honor the expectations (contract) set by their base class (or interface), maintaining consistent behavior and properties

Dependency inversion principles

High level module should not depend on low level module

This approach inverts the usual pattern where higher-level code is tightly coupled to the lower-level implementation. By introducing an abstract layer (like an interface), both high- and low-level modules depend on contracts, rather than each other’s concrete details. This results in *loose coupling*, making code more flexible, easier to modify or extend, and more maintainable.

his example solves the Dependency Inversion Principle (DIP) problem by ensuring that the high-level module (Computer) does not depend on the concrete low-level modules (WiredKeyboard, WirelessKeyboard) but instead depends on an abstraction (Keyboard interface).

* The Computer class depends on the Keyboard interface, not any particular implementation. This means Computer can work with *any* kind of Keyboard, as long as it implements the Keyboard interface.
* Both WiredKeyboard and WirelessKeyboard are low-level modules and depend on the abstraction (they implement the Keyboard interface), not the Computer class or each other.
* Because of this design, if you want to add a new keyboard type (like a GamingKeyboard), you only add a new class implementing Keyboard—no changes are needed to the Computer class. This promotes loose coupling and makes your system more extendable and maintainabl

https://youtu.be/8x7-IlibqeQ?si=XxbH9fNlycw3qaC3