**Design principles-**

Design principles are the guidelines to follow in order to make application maintainable and understandable.

SOLID design principles-

i) single responsibility

ii) Open closed principle

iii)Liskov's substitution principle

iv) Interface segregation

v) Dependency inversion

**i) Single Responsibility Principle-**

Single responsibility principle state that every method, module or class should have single responsibility.

that is there should be only one reason to change the class. If a function is performing some action, then it should not return a value

or vice versa. if we find any class or method having multiple responsibility, we should divide class or method.

**ii) Open Closed Principle-**

A class should open for extension but close for modification. Intention behind open closed principle is to reduce changes in code during maintenance phase.

Let’s take example-

Class Rectangle {

………………

Public float area () {

…………………

}

}

Class square {

Public float area (){

…………………

}

}

Class Main {

Public static void main () {

// here if we want to calculate area of every shape we need to explicitly mention that shape.

}

}

When we add a new shape, we need to change everything.

Solution to this is define an interface called Shape.

Interface Shape {

Public float area ();

}

Now those classes which come under shape will implement it and we can later add more shapes by implementing shape interface.

**iii) Liskov's Substitution principle-**

A derived class object should be substitutable for its base class object subjected to not change in the functionality of code.

Class Rectangle {

float length;

float width;

Public float area (){

Return length\*width;

}

}

Class square extends Rectangle{

float side;

}

Above example violates Liskov’s substitution principle as square class extends rectangle due to which area calculated would be wrong.

**iv) Interface Segregation-**

Interface Segregation principle states that a class should not be forced to implement interface methods that are not relevant to them. if interface contains unnecessary methods then that interface is called as fat interface. if fat interface is there then we can split that interface into smaller ones and using appropriate way we can avoid violation of interface segregation principle.

ex. violation of interface segregation-

interface Animal {

public void walk ();

public void eat ();

}

class Cat implements Animal {

.......

}

class Crocodile implements Animal {

....

}

in above example, even though crocodile can't walk he is forced to implement eat () method. That should not be case. What we can do here is

make a new interface

interface Animal {

public void eats ();

}

interface crawlable {

public void crawl ();

}

class Crocodile implements Crawlable, Animal {

.......

}

int this way we can split main interface to prevent violation of interface segregation principle.

**v) Dependency inversion-**

This principle states that high level methods should not depend on low level function. There should be a abstract layer between high level and low level methods. High level function and low level function should depend on abstraction, that will provide loose coupling.

ex.

class Address {

int zipCode;

String city;

....

}

class User {

Address address;

}

here user has dependency of address which is lower level class.

In order to prevent the violation of principle we introduce a abstraction layer as below

interface IAddress {

......

}

class Address implements IAddress {

.........

}

class User {

IAddress address;

}

now dependency structure would be User----> IAddress -----> Address.