WEEK - 1 :-

DESIGN PRINCIPLES AND PATTERNS :-

DESIGN PRINCIPLES :-

S – Single responsibility principle.
O – Open/closed principle
L – Liskov substitution principle
I – Interface segregation principle.
D – Dependency Inversion principle

Introduction to solid principles :-

→ Solid principles are introduced by Robert C. Martin in his 2000 Paper "Design Patterns and Design Principles"

→ Later built by Michel feathers who introduced SOLID acronym.

→ These 5 principles revolutionized the object oriented programming changing the way we write software.

→ Martin & feathers "Design Principles & design Patterns" encourage us to create more maintainable, understandable and flexible software

→ As our application grows these principles helps us to reduce Complexity.

Single Responsibility Principle :-

→ SRP (Single responsibility principle) state that "A class should only have one responsibility. Furthermore it should have only one reason to change".

How does SRP helps us to buid better software :-

[1] Testing :- A class with one responsibility have fewer TC.

[2] Lower coupling :- A class with one responsibility have fewer dependen

[3] Smaller, well organised casses are easir to search.

Example :- (with SRP principal).

```
Class Zoo Entity
{
    // Staff
    String name;
    String gender
    int age;
    Double salary;
    String department;

    // Animal
    String name;
    String gender;
    int age;
    String species;
    boolean canFly;
    boolean eatsMeat;

    //visitors.
    String name;
    int age;
    String gender.
    Long TicketId;
    DateTime Date;

//Methods

    // staff methods.
    void sleep();
    Void eat();
    Void walk();
    void feedAnimal();
    Void cleanPremises();

    // Animal methods.
    Void sleep();
    void eat();
    Void fly();
    void fight();

    // visitor methods.
    void eat();
    Void roamAround();
    Void clickPictures();
}
```

Problems with above code:-

→ Variable naming conflict

→ Difficult to test

→ Tightly coupled

→ Comes under Violation of SRP.

Code :- (with SRP Principle)

```
class ZooEntity
{
    String name;
    String gender;
    int age;

    Void eat();
    Void sleep();
}
```

```
class Staff extends ZooEntity
{
    double salary;
    String designation;

    Void feedAnimal();
    Void cleanPremises();
}
```

```
class Animal extends ZooEntity
{
    boolean canFly;
    boolean eatsMeat;

    Void fight();
    Void fly();
}
```

```
class Vistor Extends ZooEntity
{
    String ticketId();
    DateTime date;

    Void roamAround();
    Void clickPictures();
}
```

[2] Open / Closed Principle :-

→ Open/closed Principle stated that "A class should open for extension but closed for modification.

```
[Public ZooLibrary]
{
    class Animal {
        String species;
    }
    class Bird Extends Animal {
        void fly() {
            if (species == "sparrow") ...
            else if (species = "pegion") ...
            else if (species == "eagle") ...
        }
```

```
import ZooLibrary.Bird
class Awesome
{
    void main() {
        Bird B = new Bird ("Pigeon")
        B.fly();
    }
}
```

→ what if client add new Bird.
→ Extension is not possible if we use if-else ladder.

[3] Liskov Substitution Principle :-

→ An object of Parent class should be replicable with any object of child class extends Parent without causing error.

(or)

→ If S is a subtype of T, the object of type T may be replaced with object of type S without altering the correctness of the program.

→ A subclass should behave in a way that it should not break the expectations set by its Parent class

→ The derived class must be completly substituable for the base class.

```
class Bird extends Animal
{
    abstract void eat();
}

Interface IcanFly()
{
    void fly()
}

class Sparrow extends Bird implements IcanFly {
    —
}

class Kiwi extends Bird{
    —
}
```

[4] Interface segregation:-

→ ISP states that large interfaces should be split into smaller ones.

→ By doing this we can ensure that implementing classes only need to concerned about the methods.

```
Public interface BearKeeper
{
    void FeedTheBear();
    void WashTheBear();
}
```

```
Public interface BearFeeder
{
    void FeedTheBear();
}
Public interface BearCleaner {
    void WashTheBear();
}
```

[5] Dependency Inversion:

→ Dependency Inversion Principle refers to decoupling of software modules.

→ High-level modules should not depend on low-level modules.

→ Both should depend on abstractions.

→ Abstraction should not depend on details.

→ Details should depend on abstraction.