Low Level Design

Date __/__/__

1) SOLID Principles.

→ Advantages

Avoid duplicate code, Flexible Software

Easy to maintain, understand, Reduce Complexity

S → Single Responsibility.

→ A class should have only 1 reason to change.

(means a class should have only one responsibility)

Ex

```
class Marker {
    string name;
    String color;
    int year;
    int price;

public Marker (String name, String color, int year, int price)
    { this.name = name;
      this.color = color;
      this.year = year;
      this.price = price;
    }
}

class Invoice {
    private Marker marker; // Invoice has a Marker
    private int quantity;
public Invoice (Marker marker, int quantity)
    { this.marker = marker;
      this.quantity = quantity;
    }
  public void int calculateTotal () {
    int price = (marker.price * this.quantity)
    return price;
  }
```

Date __/__/__

```
Public void printInvoice() {
    // print invoice
}

public void SaveToDB() {
    // save data into DB
}
}
```

=> now suppose in calculateTotal we want to add GDP & discount then the logic will change right?!

=> similarly if we want to change how the printInvoice() is printed then also logic is changed.

=> same with saveToDB() logic is changed if we want some other thing.

=> So in total we have 3 reasons to change, But Single Responsibilty (S) says a class should have only one 1 reason to change.

Now Invoice class is not following Single Responsibilty so to correct it make diff classes for each function such that Each class is having only 1 responsibility to change.

Ex

```
class InVoice {
    private Marker marker;
    private int quantity;
    public Invoice ( Marker marker, int quantity)
        { ... } // constructor.
    Public int calculateTotal () { ... logic }.
}
```

//vijeta//

Date __/__/__

```
class InvoiceDataAccesslayer {
  Invoice invoice;
  public InvoiceDataAccesslayer ( Invoice invoice )
    { this.invoice = invoice ; }

  public void SaveToDB () {
      // logic for save into DB
  }

class InvoicePrinter {
  private Invoice invoice;
  public InvoicePrinter ( Invoice invoice)
    { this.invoice = invoice ; }
  public void print() {
      // print invoice
  }
}
```

=> Now we can see each class has single responsibility to change.

=> So it is useful like if I want to change calculation logic I need only Invoice class to change.

② O → Open / Close Principle

→ Open for Extension But Close for Modification

=> Now above we have a class InvoiceDataAcessLayer right ?! , suppose a requirement comes & says saveToFile also so what can we do ? We can ~~no~~ add function ~~of~~ to the class.

```
class InvoiceDataAccesslayer {
    // Constructor
  Public void SaveToDB { ... }
  public void SaveToFile { ... }
```

=> But is it following (O), Ans is No. Date / /

as it says that a Class is open for Extensions means it can be extended (or Inherited) but cannot be modified (means cannot add more functionality).

=> So what we can do is make another class that Extends Invoice DataAccess layer to save to File.

=> Solution :

```
interface InvoiceDataAccessLayer {
    public void save (Invoice invoice);
}
```

① Class DatabaseInvoiceDAL implements InvoiceDataAccesslayer

```
{
    @Override
    public void save (Invoice invoice)
    { // Save to DB }
}
```

② Class FileInvoiceDAL implements InvoiceDataAccessLayer

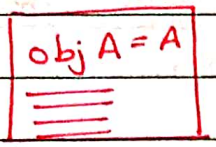
```
{   @Override
    public void save (Invoice invoice)
    { // save to file }
}
```

=> So in futur if another function comes like SaveToXYZ we we simply extend it & write logic.

Date __/__/__

(3) L → Liskov Substitution Principle

if Class B is subtype of A, then we should be able to replace object of A with B without breaking the behaviour of Program

Ex Suppose 

we have object A & after that some operations are done on A

Now as we know B extends A so if we replace obj A with obj B then the further operations should not cause any error.

=> i.e subclass should extend the capability of parent class & not narrow it down.

Ex
```
interface Bike {
    void turnOnEngine();
    void accelerate();
}

class MotorCycle implements Bike {
    boolean isEngineON;
    int speed;
    public void turnOnEngine() {
        isEngineOn = True;
    }
    void accelerate() {
        accelerate = accelerate + 18;
    }
}

class Bicycle implements Bike {
    public void turnOnEngine() {
        throw new Error AssertionError("there is no engine");
    }
    public void accelerate() {
    }
}
```

Date __/__/__

So in above Ex assume Bike as (A)
Motorcycle as (C) & Bicycle as (B)
now if we make
A a = new C();
that is we make object a of type motor cycle
then we can perform a.turnOnEngine(). &
a.accelerate()
So MotorCycle Class (C) is following Liskov principle

But if we do
A a = new B()
i.e we here make obj a of type Bicycle
& then we do a.turnOnEnigine() → this would throw error, which should not happen.
, this example denotes Bicycle is not following Liskov principle because it is narrowing down the functionality.

④ I → Interface Segmented Principle
Interface should be such that Client should not implement unnecessary functions they do not need.
Ex
```
interface RestaurantEmployee {
    void washDishes();
    void serveCustomers();
    void cookFood();
}

class waitor implements RestaurantEmployee {
    public void washDishes() { → not req
        // not my Job
    }
    public void serveCustomers() {
    }
    public void cookFood() { → not req.
    }
}
```

Date __/__/__

→ In above Example we can see that waiter has to implement all functions of Interface, But wait! does a waiter cooks Food? or clean Dishes? No right?! so why to implement unnecessary functions. this is what (I) says in SOLID principle.

=> Solution: is to breakDown Interfaces into seperate interface.

```
interface WaiterInterface {
    void serveCustomers();
    void takeOrder();
}
interface ChefInterface {
    void cookFood();
    void decideMenu();
}
public Waiter implements WaiterInterface {....}
```

⑤ D → Dependency Inversion Principle.

→ Class should depend on Interfaces rather than Concrete Classes

Ex: Interface Mouse → Classes: WiredMouse, BluetoothMouse.
Interface Keyboard → Classes: WiredKeyboard, Bluetooth.

```
Class MacBook {
    private final WiredKeyBoard keyboard;
    private final WiredMouse mouse;
    public MacBook() {
        Keyboard = new WiredKeyBoard();
        mouse = new WiredMouse();
    }
}
```

this is not preferred.

// coz. it might happen we change keyboard to Bluetooth keyboard, but we will not be able to do this

=> so it is adviced to do with constructor injecton. i.e.

Date __/__/__

```
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

public MacBook (Keyboard keyboard; Mouse mouse)
  {   this.keyboard = keyboard; }
      this.mouse = mouse;
}
```

pass like this.

this is not hardcoded it is dynamic