S.O.L.I.D principles

helps in writing better code:

- avoid duplicate code
- easy to maintain
- software flexibility

1) Single Responsibility

A class should have only 1 reason to change

only 1 responsibility for each class

```
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }

    public void printInvoice() {
        //print the Invoice
    }

    public void saveToDB() {
        // Save the data into DB
    }

}
```

above class would have to be changed for multiple reasons

- If Price calculation logic changes : add discount, tax.... Not following "S"
- If Printing logic changes
- If db is changed : MySQL to mongodb migration then save logic changes

To fix we can have 3 classes instead of 1 — each class taking each of the 3 responsibilities

```
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }
}
```

2) Open | close

open for extension but closed for modification

consider the InvoiceDao class is being used in production code

Now a new requirement came up to save to file, we can update the class as follows ↓

```
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

     public void saveToDB() {
         // Save Invoice into DB
     }

    public void saveToFile(String filename) {
        // Save Invoice in the File with the given name
    }
}
```

but why do you want to modify a already tested class, it can introduce unnecessary bugs into Prod code.

Instead we can make the class extendable, rather than having to modify every time we need a new saveTo funtionality

```java
interface InvoiceDao {

    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {

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

class FileInvoiceDao implements InvoiceDao {

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

Any new saveTo funtionality in future will have a new class that extends InvoiceDao instead of modification

3) Liskov Substitution

If class B is a subtype of class A, then we should be able to replace obj of A with B without breaking the program

⇒ subclass should extend the capabilities of parent class not nallow it down

logic i.e., at runtime

```
interface Bike {

    void turnOnEngine();
    void accelerate();
}

class MotorCycle implements Bike {

    boolean isEngineOn;
    int speed;

    public void turnOnEngine() {
        //turn on the engine!
        isEngineOn = true;
    }

    public void accelerate() {
        //increase  the speed
        speed = speed + 10;
    }
}
```

In above ex, Motorcycle class satisfies L

but Bicycle class doesn't as it narrowed down the capabilities by removing "turnOnEngine" capability

replacing `Bike b = new MotorCycle();`

with `Bike b = new Bicycle();`

will break `b.turnOnEngine()`

see end of page for a way to solve this

4) Interface Segmented

Interfaces should be defined such that client shouldn't have to implement unnecessary functions that they don't need

```java
interface RestaurantEmployee {
    void washDishes();
    void serveCustomers();
    void cookFood();
}

class Waiter implements RestaurantEmployee {

    public void washDishes(){
        //not my job
    }

    public void serveCustomers() {
        //yes and here is my implemenation
        System.out.println("serving the customer");
    }

    public void cookFood(){
        // not my job
    }
}
```

To fix this we can have interfaces like WaiterInterface, CookInterface .....

7) Dependency Inversion

class should depend on interfaces not on concrete classes

The Properties of class should be of interface type
with value assigned through Polymorphism [run-time]

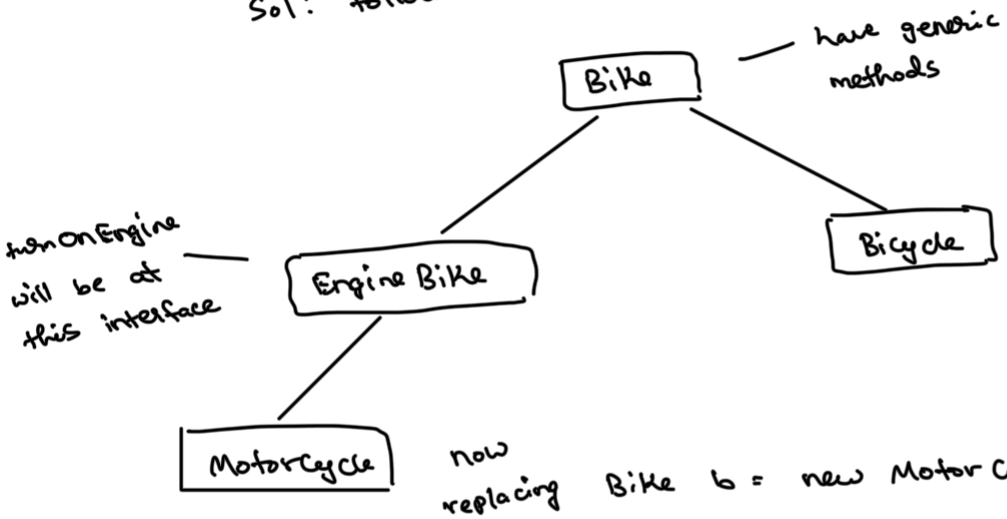
```java
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

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

interfaces

can use different keyboard & mouse types

More on Liskov Substitution

Sol: follow this obj hierarchy



now replacing `Bike b = new MotorCycle();`

with `Bike b = new Bicycle();`

will break `b.turnOnEngine()` at compile time itself

code won't break when it is ran in Prod