SOLID Principle

S - Single Responsibility Principle

O - Open / Closed Principle

L - Liskov Substitution Principle

I - Interface Segmented Principle

D - Dependency Inversion Principle

a class should have only one reason to change.

we should not modify any class, instead we should just extend other classes to override.

child class should only improve capabilities of par class and not narrow it down

client should not need to implement any unnessarary methods.

class should depend on interfaces rather than concrete classes.

1) Single Responsibility Principle

problem

```
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
    }
}
```

we might need to change calc" in future due to GST calc" This is one reason to change and we might need to change print Inovice in future, (like now we need to download invoice as well)

Also, saveToDB maybe in future we need to save in file instead of DB.

So, there are multiple reasons in this class to change in future.

So, this class is not following Single responsibility Principle

Solution

⇒ So try to create each class so that each class will have only one reason to change.

```java
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;
    }
}
```

```java
class InvoiceDao {
    Invoice invoice;

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

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

```java
class InvoicePrinter {
    private Invoice invoice;

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

    public void print() {
        //print the invoice
    }
}
```

Note :- DAO :- data access layer

2) Open for extension but closed for modification

previous example :-

```java
class InvoiceDao {
    Invoice invoice;

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

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

```java
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
    }
}
```

Note :- modified the same class for a new method, but class was already tested and live in production, so now production might have error.

So, instead of modifying the same class, extend it

Solution:-

```java
interface InvoiceDao {

    public void save(Invoice invoice);
}
```

```java
class DatabaseInvoiceDao implements InvoiceDao {

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

```java
class FileInvoiceDao implements InvoiceDao {

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

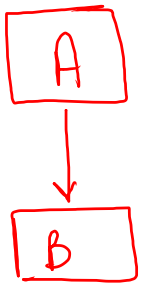
Now, same interface is implemented by both classes, either for DB or for file

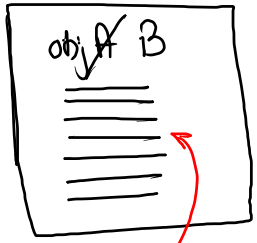
---

So if anything extra comes later, then I will just extend the interface for another class instead of modifying any class.

3) Liskov Substitution Principle :-

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





all of it should still be working.

gnp Subclass should extend the capabilities of parent class not narrows it down.

Ex:-

```
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;
    }
}
```

```
class Bicycle implements Bike {

    public void turnOnEngine() {
        throw new AssertionError( detailMessage: "there is no engine");
    }

    public void accelerate() {
        //do something
    }
}
```

here, Bicycle class narrowed down properties of Bike because now we dont have turnOnEngine and we are getting error.

Problem :-

```java
public class Vehicle {

    public Integer getNumberOfWheels(){
        return 2;
    }

    public Boolean hasEngine(){
        return true;
    }
}
```

returning null

```java
public class MotorCycle extends Vehicle{
}
```

```java
public class Car extends Vehicle{

    @Override
    public Integer getNumberOfWheels() {
        return 4;
    }
}
```

```java
public class Bicycle extends Vehicle{

    public Boolean hasEngine(){
        return null;
    }
}
```

client code

```java
public class Main {
    public static void main(String args[]){

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        vehicleList.add(new Bicycle());

        for(Vehicle vehicle : vehicleList){
            System.out.println(vehicle.hasEngine().toString());
        }
    }
}
```

will give null pointer exception for Bicycle (this has broke the code)

Solution:-

```java
public class Vehicle {

    public Integer getNumberOfWheels(){
        return 2;
    }
}
```

Note:- Parent class should be having only very generic methods which are used by all.

now, hasEngine is not present for Bicycle

```java
public class EngineVehicle extends Vehicle{

    public boolean hasEngine(){
        return true;
    }
}
```

```java
public class Bicycle extends Vehicle{
}
```

```java
public class EngineVehicle extends Vehicle{

    public boolean hasEngine(){
        return true;
    }
}
```

```java
public class MotorCycle extends EngineVehicle{
}
```

client code 1

```java
public class Main {
    public static void main(String args[]){

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        vehicleList.add(new Bicycle());

        for(Vehicle vehicle : vehicleList){
            System.out.println(vehicle.getNumberOfWheels().toString());
        }
    }
}
```

will work bez Vehical class have this method

client code 2

```java
public class Main {
    public static void main(String args[]){

        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        vehicleList.add(new Bicycle());

        for(Vehicle vehicle : vehicleList){
            System.out.println(vehicle.hasEngine());
        }
    }
}
```

compilation error:- Vehical class doesn't have hasEngine method

client code 3

```java
public class Main {
    public static void main(String args[]){

        List<EngineVehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        vehicleList.add(new Bicycle());

        for(EngineVehicle vehicle : vehicleList){
            System.out.println(vehicle.hasEngine());
        }
    }
}
```

EngineVehicle class cannot make object of Bicycle

4) Interface Segregation Principle

Interfaces should be such, that client should not implement unnecessary functions they do not need

Ex:-

```
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
    }
}
```

not a waiters job then should he implement it

Solution:-

```java
interface WaiterInterface {
    void serveCustomers();
    void takeOrder();
}
```

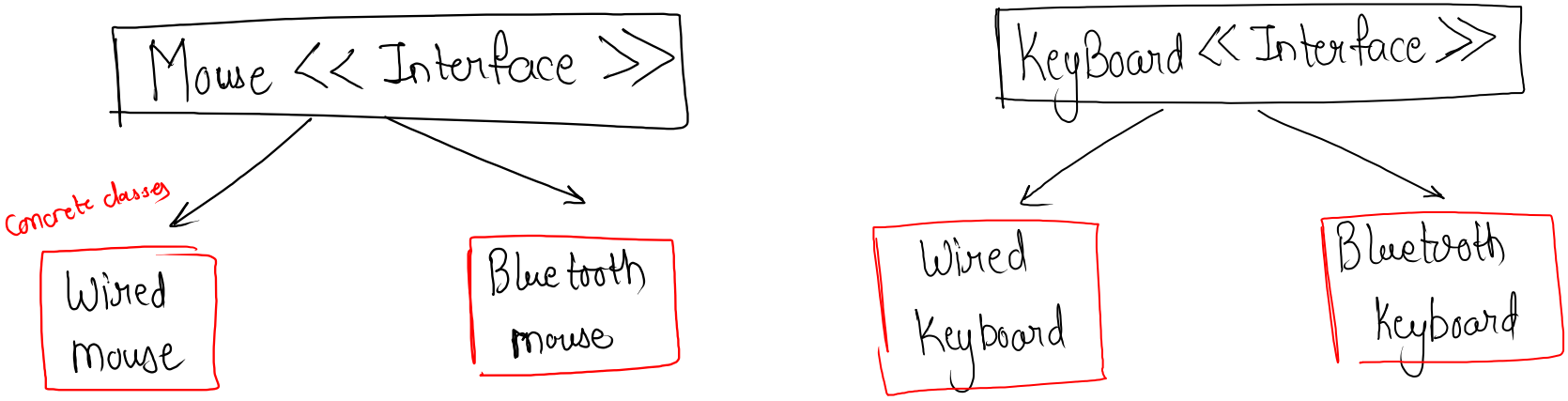
```java
interface ChefInterface {
    void cookFood();
    void decideMenu();
}
```

```java
class waiter implements WaiterInterface {

    public void serveCustomers() {
        System.out.println("serving the customer");
    }

    public void takeOrder(){
        System.out.println("taking orders");
    }
}
```

now, waiter is only implementing what it needs to implement

5) Dependency Inversion Principle

★ **Class should depend on interfaces rather than concrete classes**



This is the scenario

Problem :-

```
class MacBook {

    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;

    public MacBook() {
        keyboard = new WiredKeyboard();
        mouse = new WiredMouse();
    }
}
```

object that we have created are of type concrete class, so now in future if we want to make our macbook have bluetooth mouse then we cannot have its properties.

Solution :-

```
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

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

now objects are of interface type and now can have properties of both bluetooth and wired mouse and keyboard.

Advantages :-

- avoid duplicate code
- Easy to maintain
- Easy to understand
- Flexible software
- Reduce complexity.