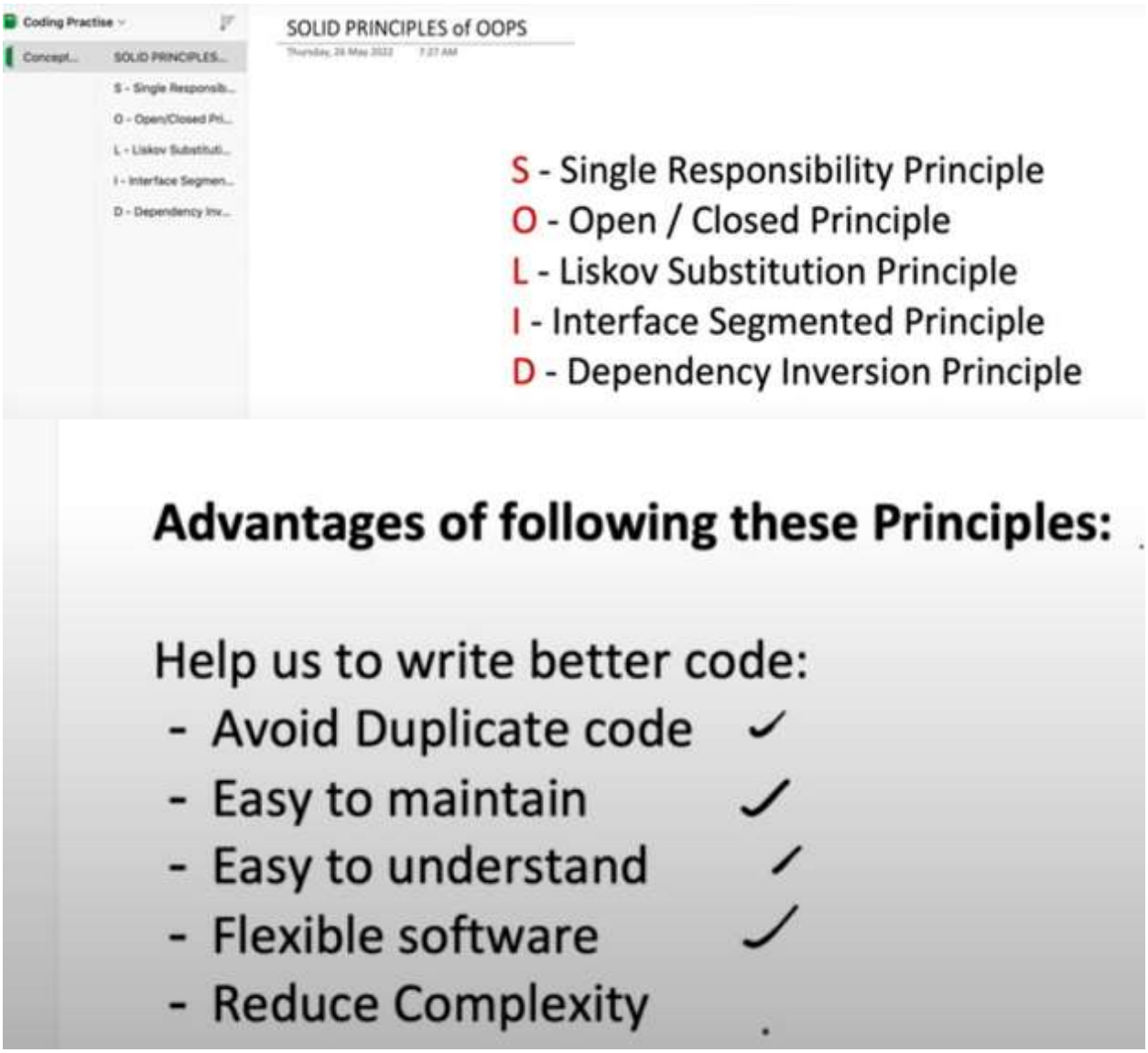

Coding Practise
Concept...
SOLID PRINCIPLES...
S - Single Responsib...
O - Open/Closed Pri...
L - Liskov Substituti...
I - Interface Segmen...
D - Dependency Inv...
SOLID PRINCIPLES of OOPS
S - Single Responsibility Principle
O - Open / Closed Principle
L - Liskov Substitution Principle
I - Interface Segmented Principle
D - Dependency Inversion Principle
Advantages of following these Principles:
Help us to write better code:
- Avoid Duplicate code
- Easy to maintain
- Easy to understand
- Flexible software
- Reduce Complexity

S - Single Responsibility Principle

Thursday, 26 May 2022 7:28 AM

A Class should have only 1 reason to change

Marker Entity:

```
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;
    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
    }
}
```

Solution: make separate class for each functionality

```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
    }
}
```

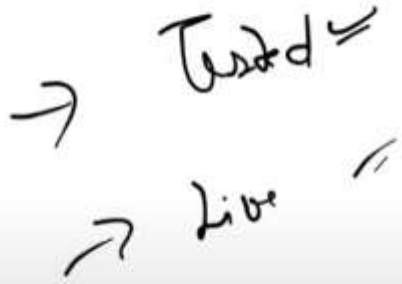
Second Principal :

O - Open/Closed Principle

Thursday, 26 May 2022 7:28 AM

Open for Extension but Closed for Modification

```java
class InvoiceDao {
    Invoice invoice;

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

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



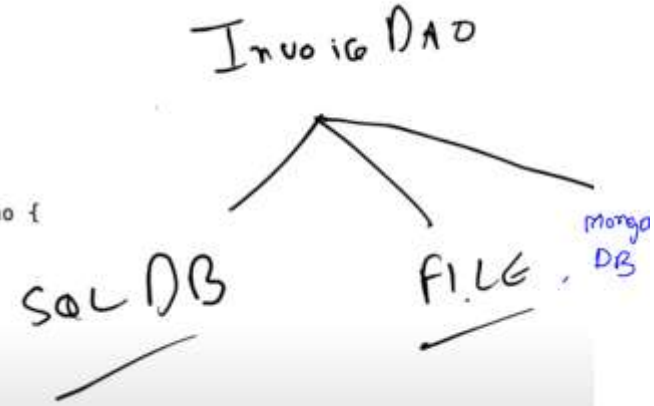
It's in production and LIVE

```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
    }
}
```

New Requirement came & we changed the live class

So, it's not following the principal

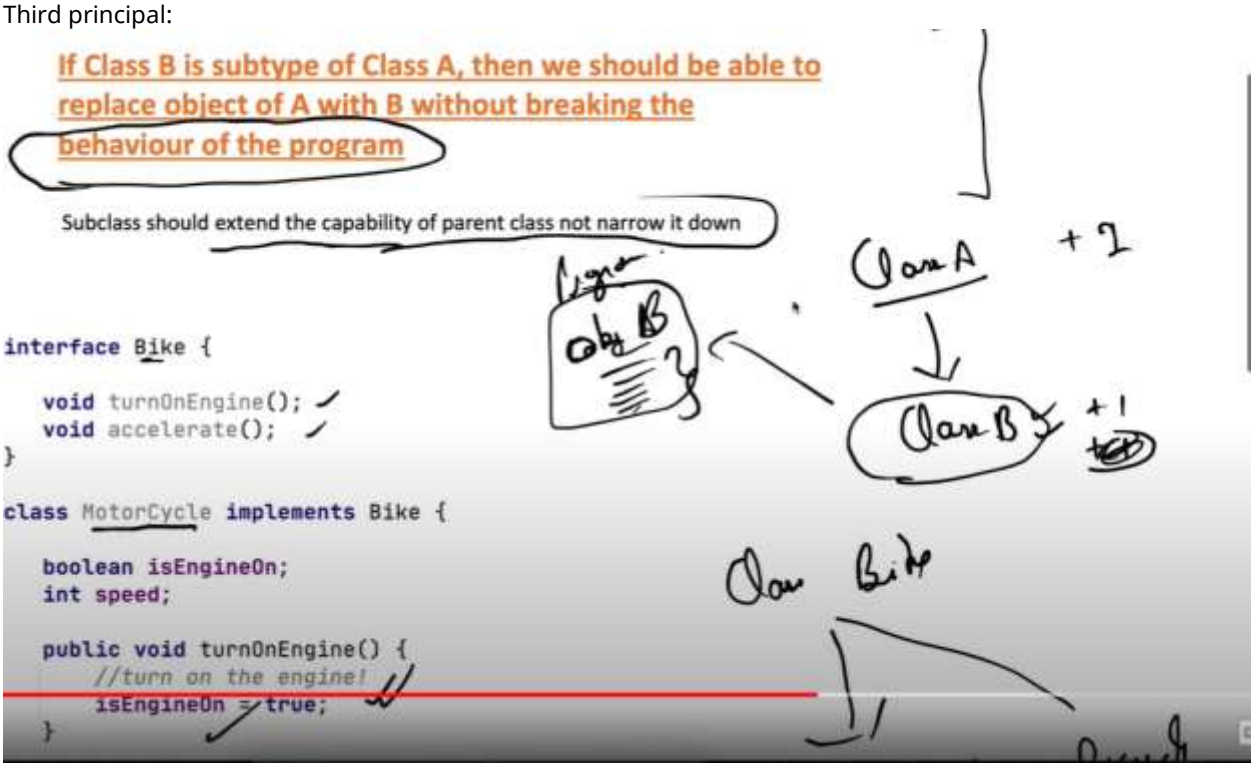
```java
interface InvoiceDao {

    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
    }
}
```

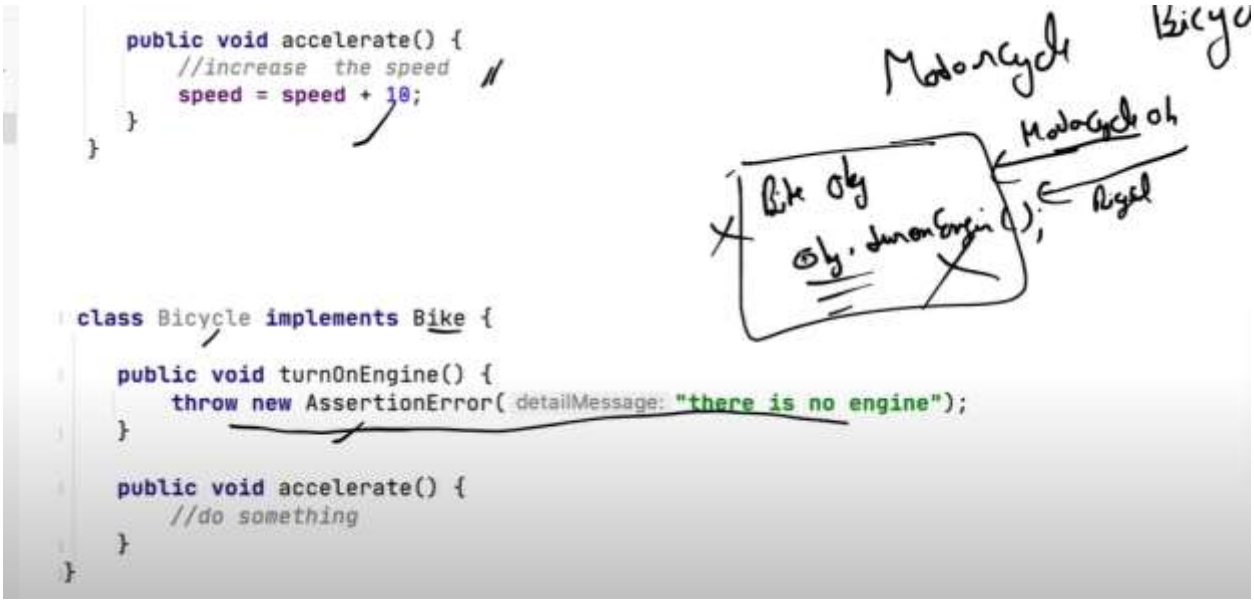


So, Instead of Making changes in live class, we can extend it and do the changes.

Just like above

Third principal:





Above in second class Bicycle , we are removing one capability of parent class Bike, so it's violating the principal.

Fourth ;

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

```
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");
    }
```

```
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
    }
}
```

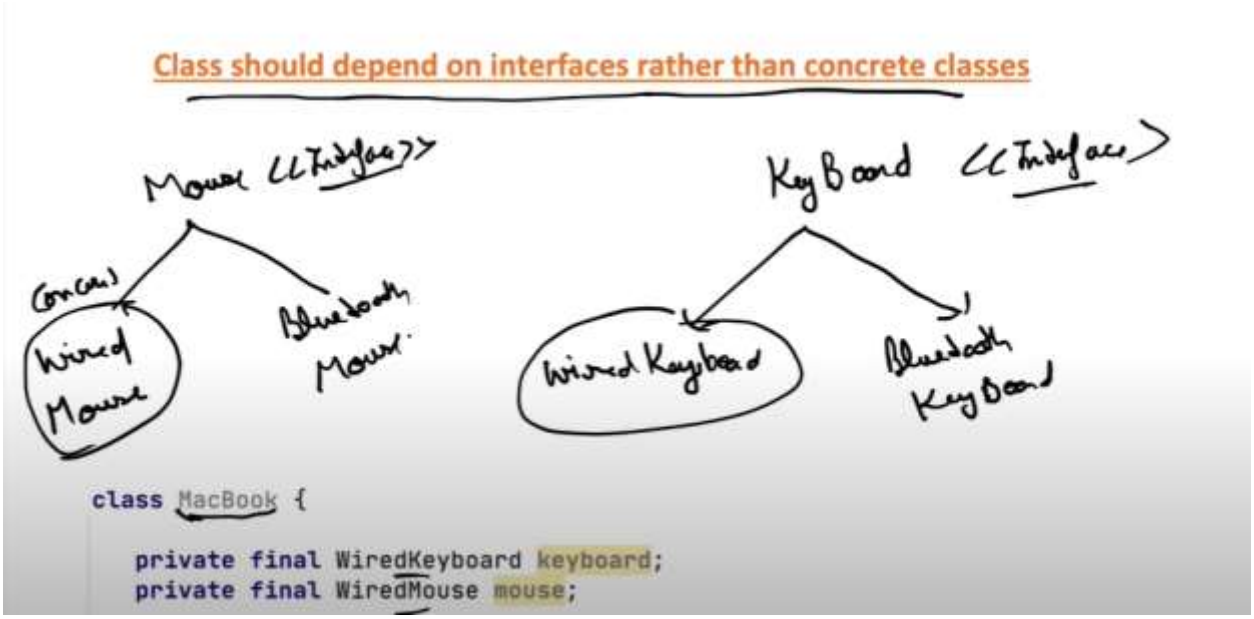
Solution

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

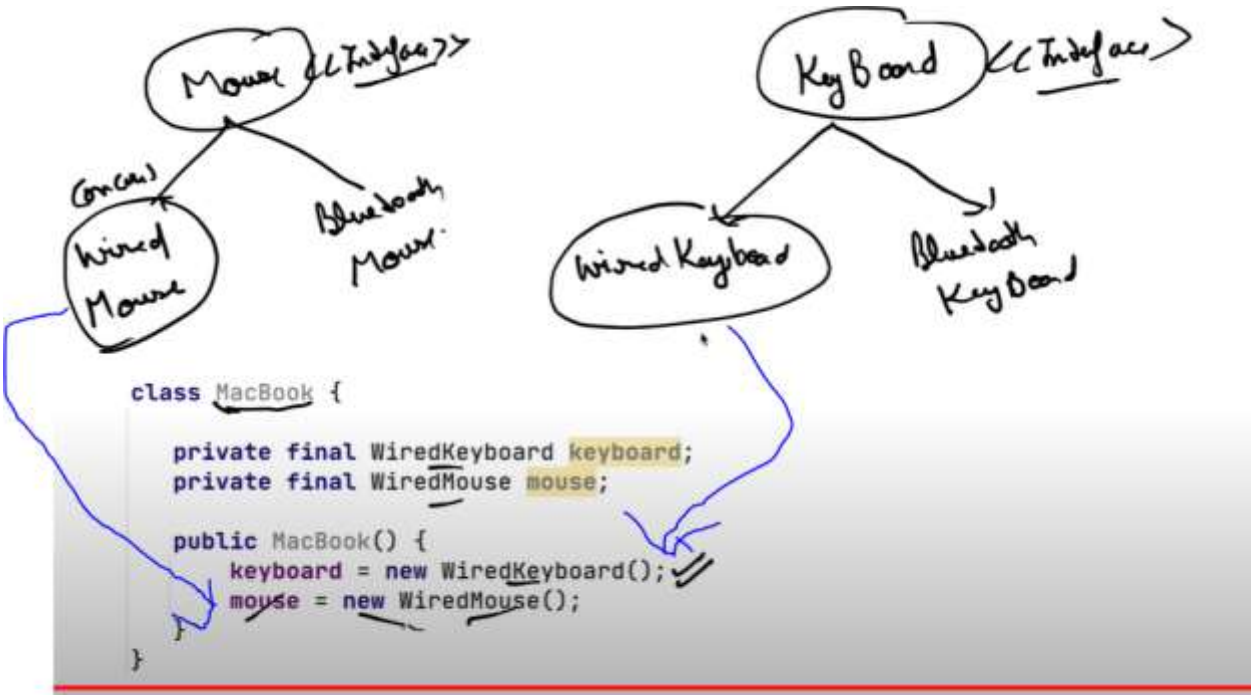
```java
interface ChefInterface {
    void cookFood();
    void decideMenu();
}
```

```java
class waiter implements WaiterInterface {

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

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

Fifth :




```java
class MacBook {

    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;
```



Assigning concrete class Object in above class, so no flexibility ,if requirement changes. i.e .
Object above will always be of WiredKeyboard and Wired Mouse

Solution:
In here , we can store any type of object of Keyboard and Mouse , be it Wired or Wireless, since Here Interface is holding/referencing implemented class object, so we have the flexibility to pass any kind of related object.

```
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

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