SOLID PRINCIPLE

Jai Ganesha's Namah

(SOLID)

S - Single Responsibility Principle
O - Open closed Principle
L - Liskov's Subsitution Principle
I - Interface Segmented Principle
D - Dependency Inversion Principle

Adv of SOLID principle

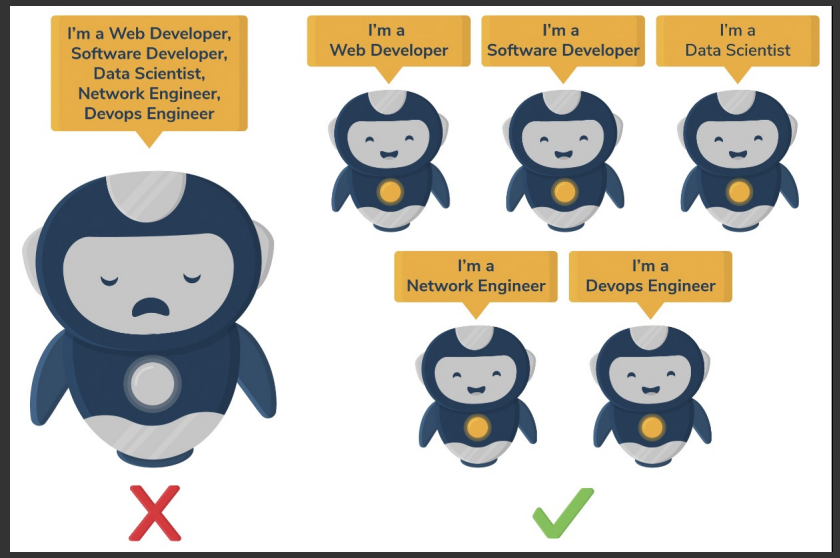
+ Avoid Duplicate code
+ Easy to maintain
+ Easy to Understand
+ Flexible Software
+ Reduce complexity

1. (Single Resposibility Principle)

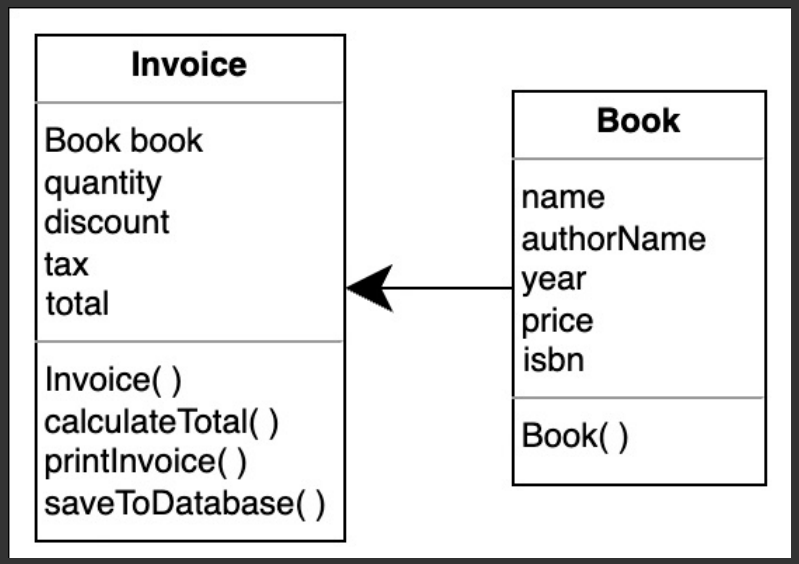
A class should have only one reason to change.

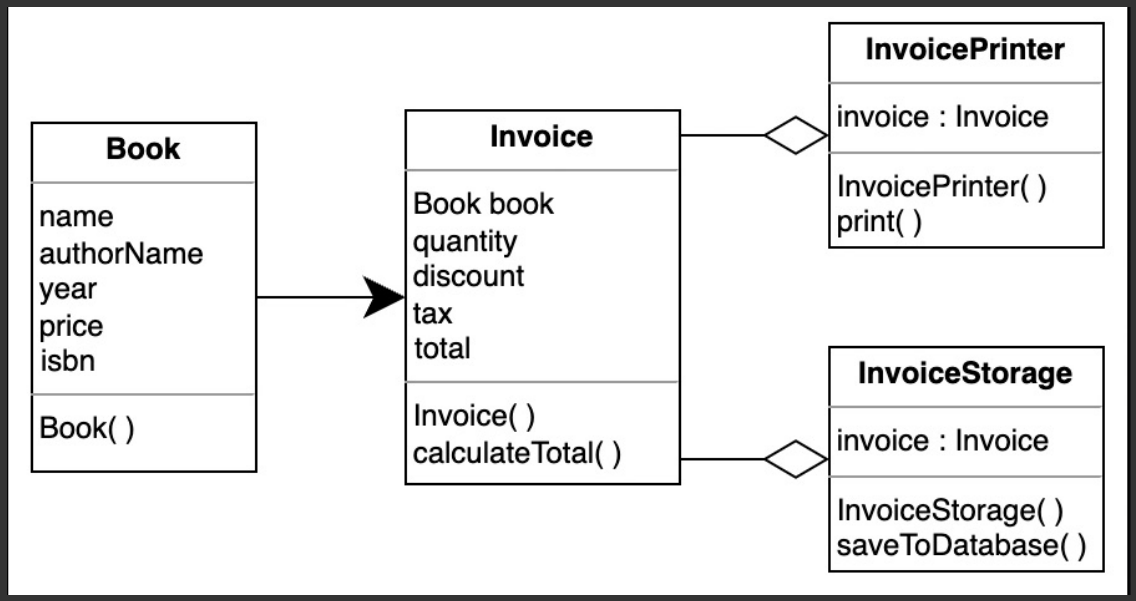
pom.xml (SRP) | Main.java | Marker.java | Invoice.java

```
2usages
public class Marker {
    1 usage
    String name;
    1 usage
    String color;
    1 usage
    int year;
    2 usages
    int price;

    no usages
    public Marker(String name, String color, int year, int price) {
        this.name = name;
        this.color = color;
        this.year = year;
        this.price = price;
    }
}
```



Real life example of SRP.





here it is violating law of single responsibility to change

pom.xml (SRP) | Main.java | Marker.java | Invoice.java

```java
no usages
public class Invoice {
    2 usages
    private Marker marker;
    2 usages
    private int quantity;

    no usages
    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    //invoice class is getting changed due to the change in the calculation logic (reason number one)
    no usages
    public int getTotalPrice() {
        return marker.price * quantity;
    }


    no usages
    public void printInvoice() {
      //we can print invoice here
    }

    //invoice class is getting changed due to the change in the Saving value on DB  (reason number two)
    no usages
    public void saveInvoiceDB() {
      //we can save invoice here
    }
}
```

We can separate classes into different smaller classes with a single responsibility.

pom.xml (SRP) | Main.java | Marker.java | Invoice.java | InvoiceDao.java | InvoicePrinter.java

```java
package org.example;

no usages
public class InvoiceDao {
    1 usage
    Invoice invoice;

    no usages
    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }
    no usages
    public  void savetoDB(){
        //save to db
    }
}
```

pom.xml (SRP) | Main.java | Marker.java | Invoice.java | InvoiceDao.java | InvoicePrinter.java

```java
package org.example;

no usages
public class InvoicePrinter {

    1 usage
    private  Invoice invoice;

    no usages
    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }
    no usages
    public void printInvoice(Invoice invoice) {
      //print invoice
    }
}
```

pom.xml (SRP) | Main.java | Marker.java | Invoice.java | InvoiceDao.java | InvoicePrinter.java

```java
package org.example;

5 usages
public class Invoice {
    2 usages
    private Marker marker;
    2 usages
    private int quantity;

    no usages
    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    //invoice class is getting changed due to the change in the calculation logic (reason number one)
    no usages
    public int getTotalPrice() {
        return marker.price * quantity;
    }

}
```

2. Open Closed Principle

"Open for extension & closed for modification"

pom.xml (SRP) | Main.java | Marker.java | Invoice.java | InvoiceDao.java | InvoicePrinter.java

```java
package org.example;

public class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }
    public  void savetoDB(){
        //save to db
    }
}
```

This is Normal case

pom.xml (SRP) | Main.java | Marker.java | Invoice.java | InvoiceDao.java | InvoicePrinter.java

```java
package org.example;

public class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }
    public  void savetoDB(){
        //save to db
    }

    public  void savetoFile(){
        //save to file
    }
}
```

New Requirement Came
Save to file.

Is it following Open/closed principle?

No

To Resolve this → we

Main.java | Marker.java | Invoice.java | InvoicePrinter.java | InvoiceDao.java | DatabaseInvoi

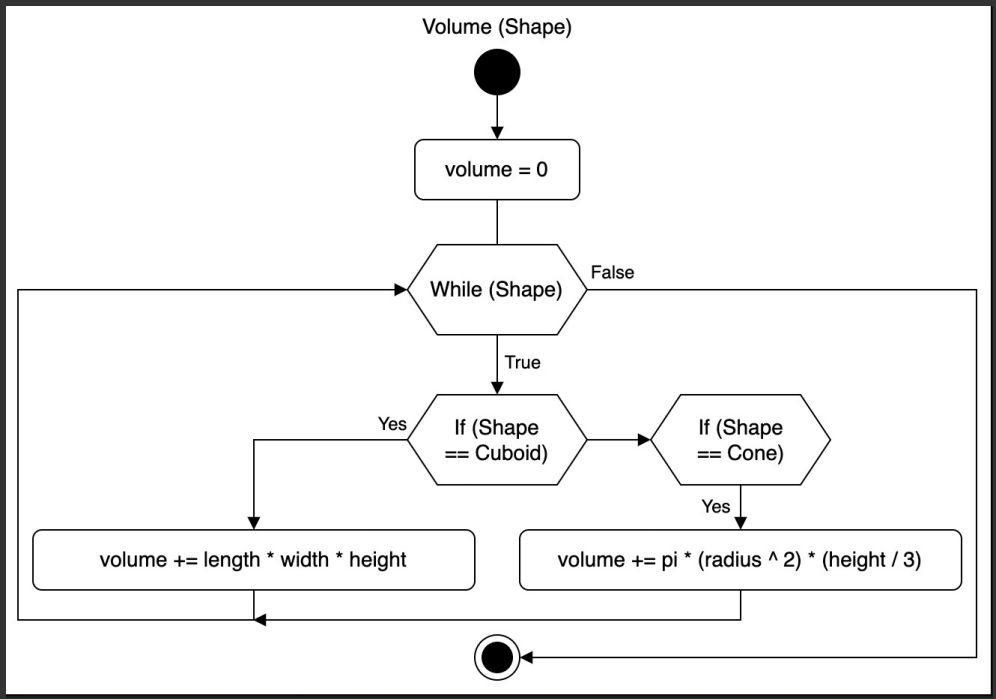
```java
package org.example;

2 usages  2 implementations
public interface InvoiceDao {

    no usages  2 implementations
    public void save(Invoice invoice);
}
```

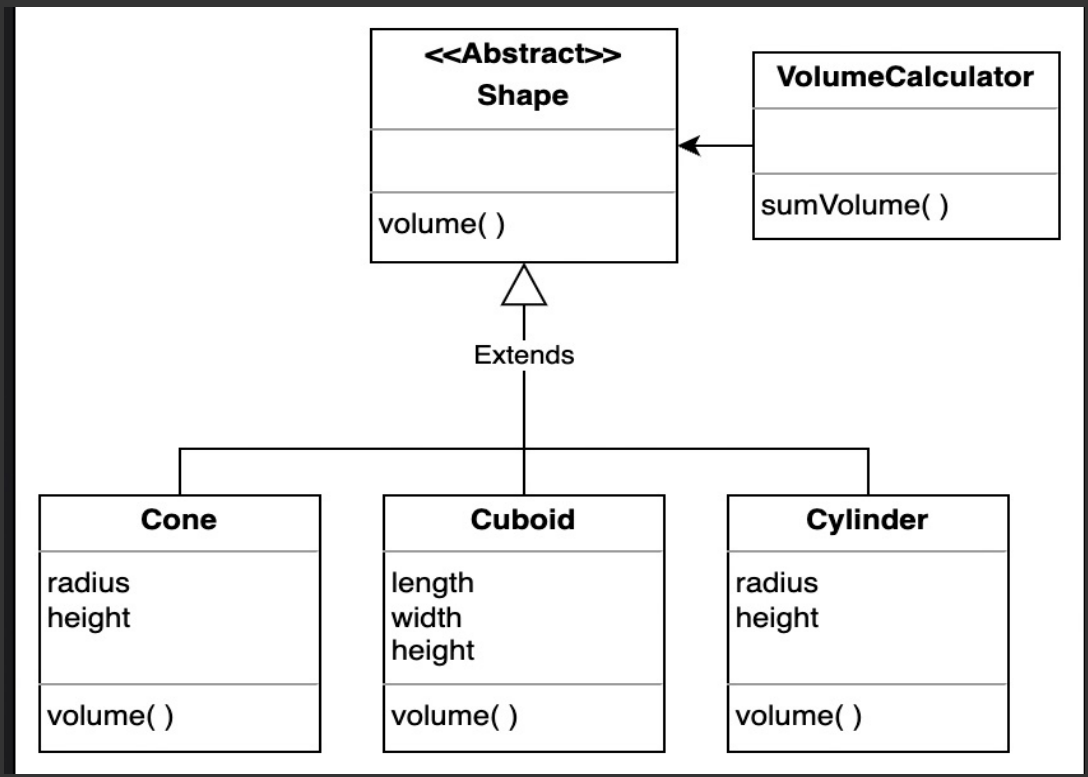
ml (SRP) | Main.java | Marker.java | Invoice.java | InvoicePrinter.java | InvoiceDao.java | DatabaseInvoiceDao.java

```java
package org.example;

no usages
public class FileInvoiceDao implements InvoiceDao{

        no usages
        @Override
        public void save(Invoice invoice) {
            //save invoice to file
        }
}
```

ml (SRP) | Main.java | Marker.java | Invoice.java | InvoicePrinter.java | InvoiceDao.java | DatabaseInvoiceDao.java

```java
package org.example;

no usages
public class DatabaseInvoiceDao implements InvoiceDao{

        no usages
        @Override
        public void save(Invoice invoice) {
            //save invoice to database
        }
}
```

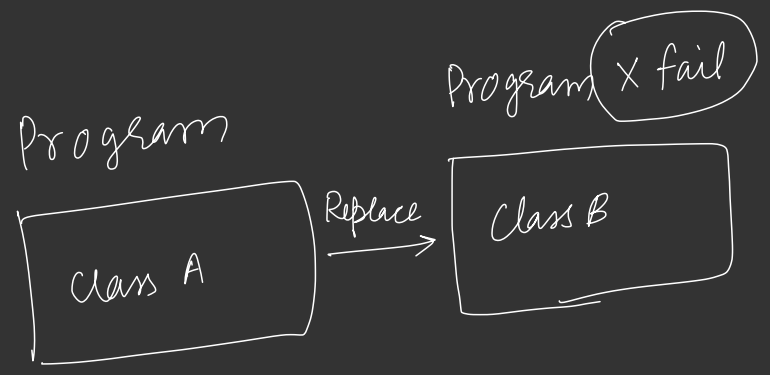

Volume (Shape)
volume = 0
While (Shape)
False
True
Yes
If (Shape == Cuboid)
If (Shape == Cone)
Yes
volume += length * width * height
volume += pi * (radius ^ 2) * (height / 3)

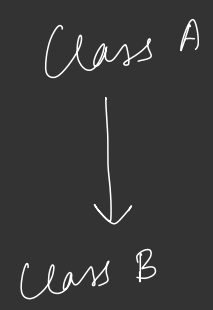

<<Abstract>>
Shape
volume( )
VolumeCalculator
sumVolume( )
Extends
Cone
radius
height
volume( )
Cuboid
length
width
height
volume( )
Cylinder
radius
height
volume( )

3. Liskov's Subsitution Principle

" If class B is subtype of class A, then we should be able to replace Object of A & B without breaking the behaviour of the Program."

★ Subclass should extend the capability of parent class not narrow it down.





Program should not break the behaviour.

Bike.java | Bicycle.java | Motorcycle.java

```java
package org.example;

2 usages   2 implementations
public interface Bike {
    no usages   2 implementations
    void turnOnEngine();
    no usages   2 implementations
    void accelerate();
}
```

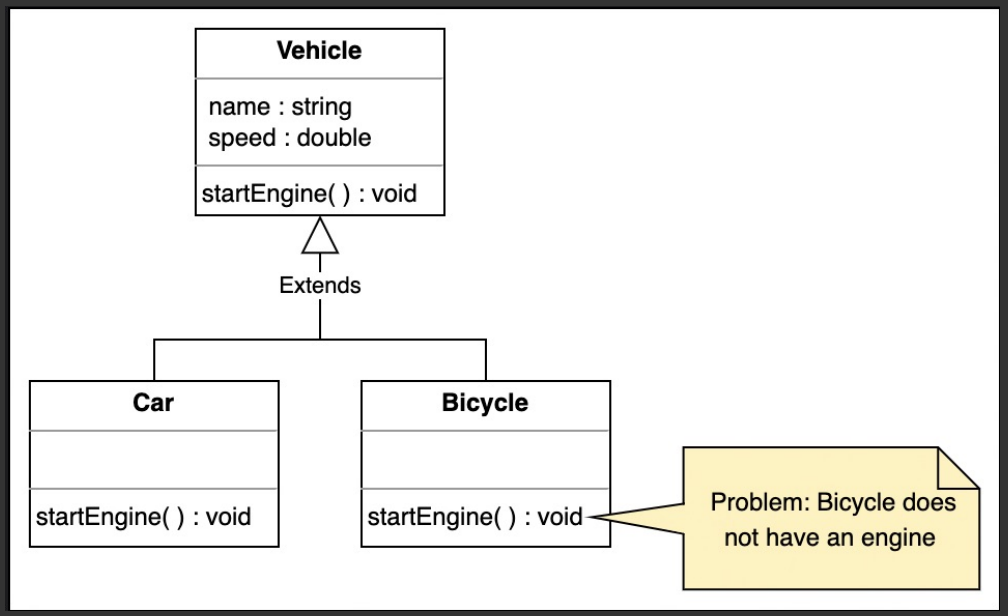
Bike.java | Bicycle.java | Motorcycle.java

```java
package org.example;

no usages
public class Bicycle implements Bike{

    no usages
    @Override
    public void turnOnEngine() { throw new UnsupportedOperationException(); }

    no usages
    @Override
    public void accelerate() {
        // Do Something
    }
}
```

Bike.java | Bicycle.java | Motorcycle.java

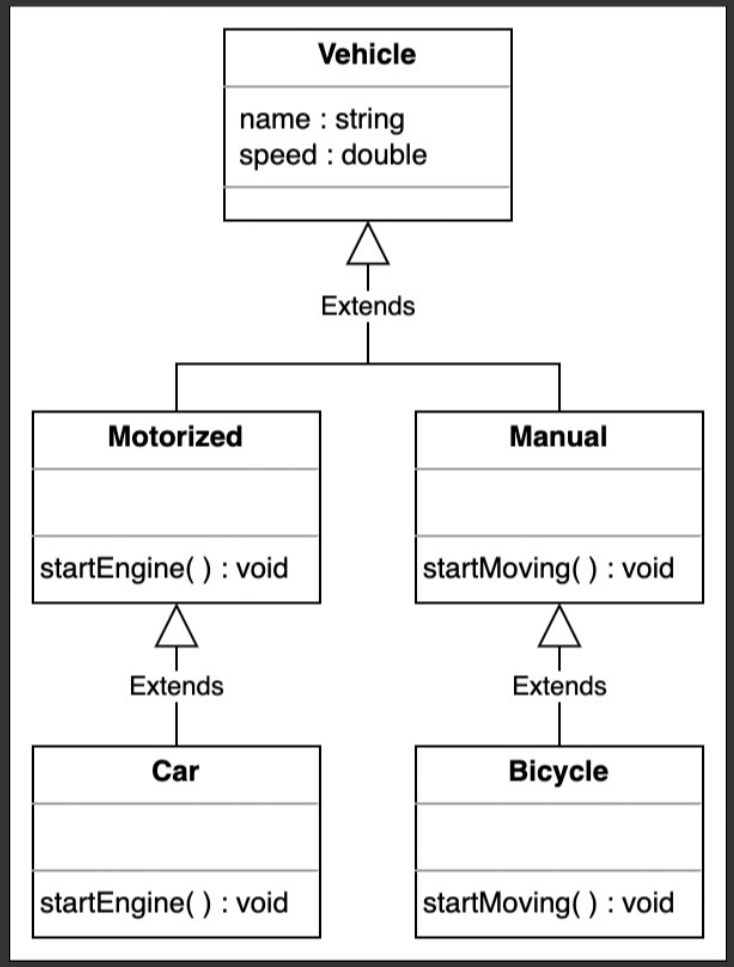
```java
package org.example;

no usages
public class Motorcycle implements Bike{

    1 usage
    boolean engineOn;
    1 usage
    int speed;
    no usages
    @Override
    public void turnOnEngine() {
        //turn on the engine
        engineOn = true;
    }

    no usages
    @Override
    public void accelerate() {
        //accelerate the motorcycle
        speed += 10;
    }
}
```

Here Example bicycle cannot replace bike it does not have turnOnEngine() functionality.



→ This is an example of failure of Liskov's substitution principle.

SOLUTION →



4. Interface Segregation Principle

" Interfaces should be such, that client should not implement unnecessary functions they do not Need".

Bike.java | Bicycle.java | Motorcycle.java | RestaurentEmployee.java | Waiter.java

```java
package org.example;

1 usage  1 implementation
public interface RestaurentEmployee {
    no usages  1 implementation
    void washDishes();
    no usages  1 implementation
    void serveCustomers();
    no usages  1 implementation
    void cookFood();
}
```

Bike.java | Bicycle.java | Motorcycle.java | RestaurentEmployee.java | Waiter.java

```java
package org.example;

no usages
public class Waiter implements RestaurentEmployee{
    no usages
    @Override
    public void washDishes() {
        // Not my job
    }

    no usages
    @Override
    public void serveCustomers() {
        // Do Something
        System.out.println("Serving the Customers");
    }

    no usages
    @Override
    public void cookFood() {
        // Not my job
    }
}
```

here client waiter is failing interface segregation principle because waiter has to implement functions which are not Needed

To Resolve this problem we will be smaller interfaces.

ChefInterface.java | Chef.java | Waiter.java | WaiterInterface.java

```java
package org.example;

1 usage  1 implementation
public interface ChefInterface {
    no usages  1 implementation
    void cookFood();
    no usages  1 implementation
    void decideMenu();
}
```

ChefInterface.java | Chef.java | Waiter.java | WaiterInterface.java

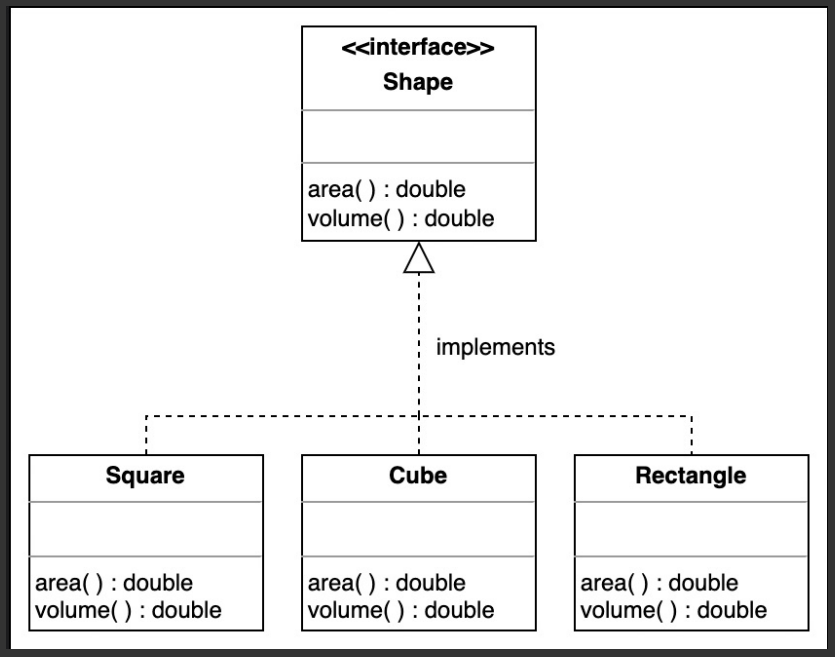
```java
package org.example;

no usages
public class Chef implements ChefInterface{
    no usages
    public void cookFood() {
        System.out.println("Cooking food");
    }

    no usages
    public void decideMenu() {
        System.out.println("Deciding menu");
    }
}
```

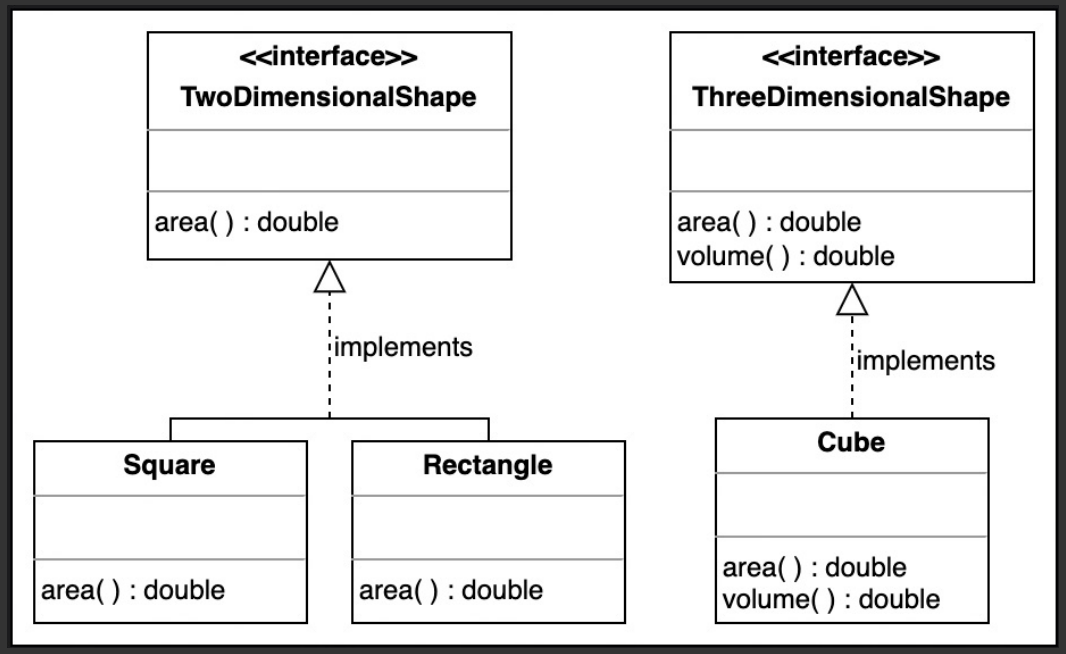
ChefInterface.java | Chef.java | Waiter.java | WaiterInterface.java

```java
package org.example;

1 usage  1 implementation
public interface WaiterInterface {
    no usages  1 implementation
    public void serveFood();
    no usages  1 implementation
    public void takeOrder();
}
```

ChefInterface.java | Chef.java | Waiter.java | WaiterInterface.java

```java
package org.example;

no usages
public class Waiter implements WaiterInterface{
    no usages
    public void serveFood() {
        System.out.println("Serving food");
    }

    no usages
    public void takeOrder() { System.out.println("Taking order"); }
}
```



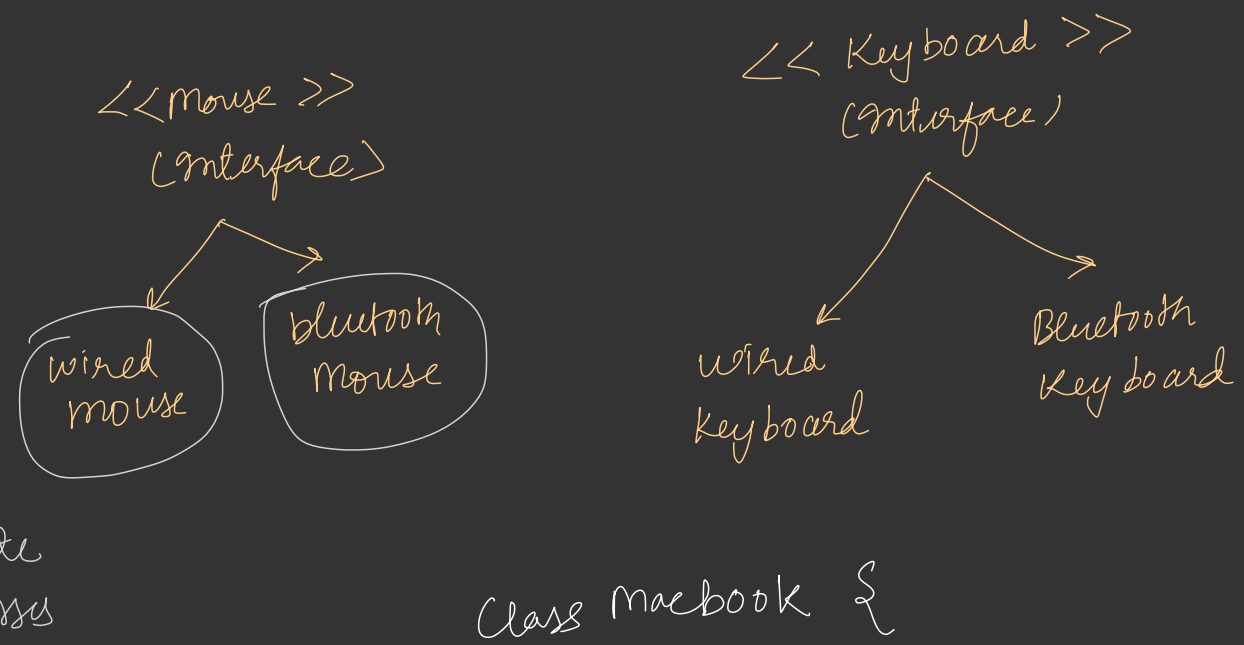


here is failure volume because 2D shape cannot have volume (Failure of ISP)

To Resolve the problem we will break into smaller interfaces.

5. DI (Dependency Inversion Principle)

"Class Should depend upon interfaces rather than concrete class"



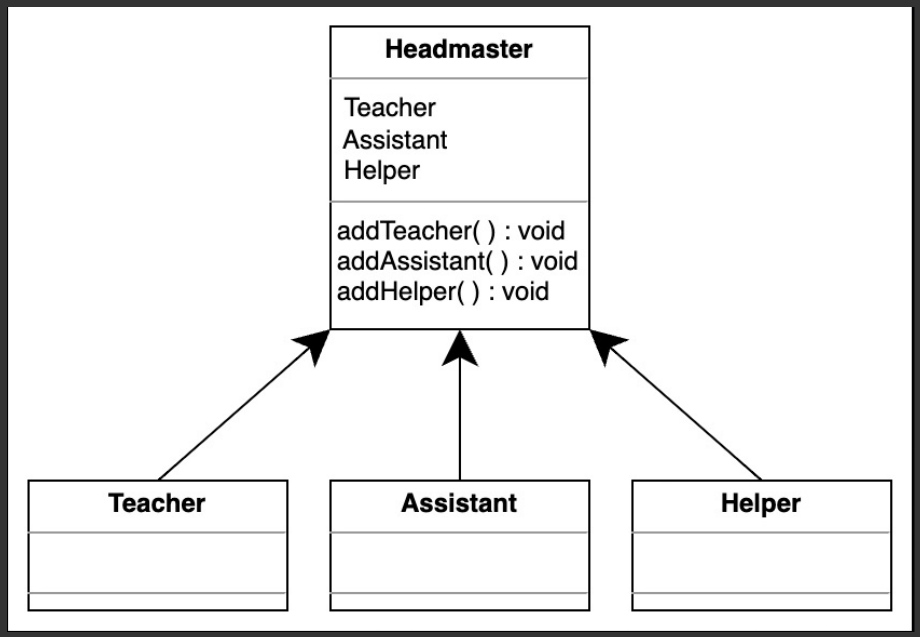
This is failing as dependent on concrete classes.

```
Class Macbook {

    private final WiredKeyboard Keyboard;
    private final Wired Mouse mouse;

    Public Macbook () {
        Keyboard = new WiredKeyboard ();
        mouse = new Wired Mouse ();
    }
}
```

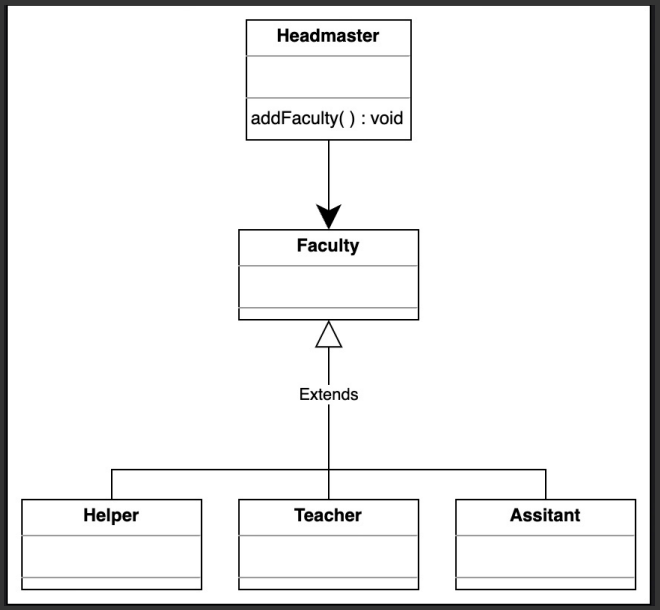
To Resolve this using constructor injection, we can change according to our need

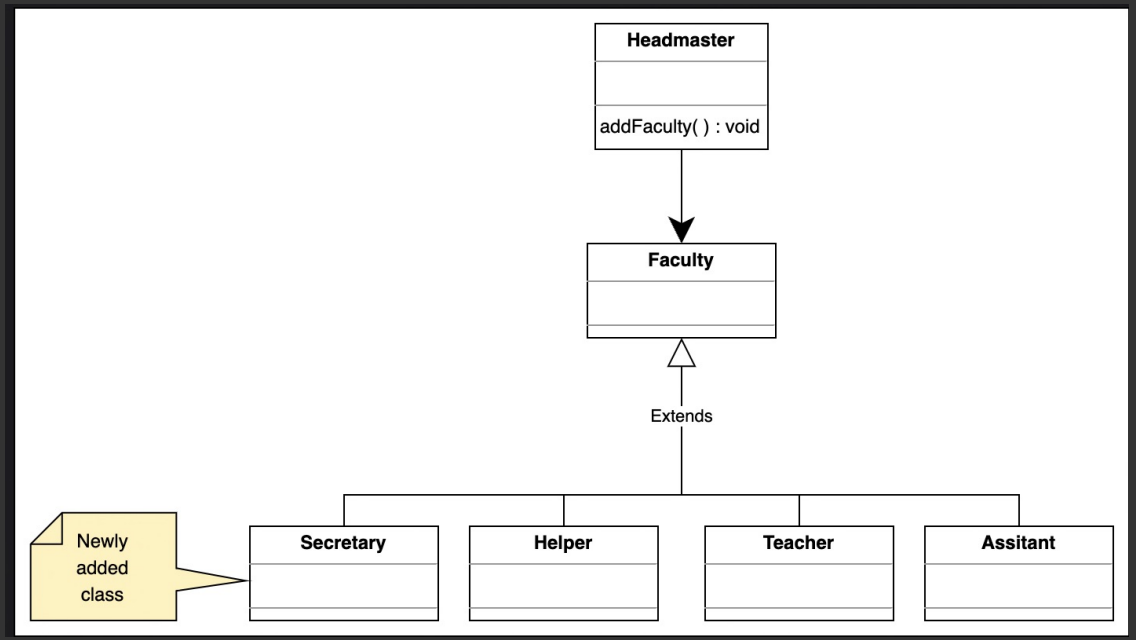
```java
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

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

Real Life Example of Dependency Inversion Principle



- abstraction not implemented, Everything from lower level exposed to upper level.


Headmaster
addFaculty( ) : void
Faculty
Extends
Helper
Teacher
Assitant


Headmaster
addFaculty( ) : void
Faculty
Extends
Newly added class
Secretary
Helper
Teacher
Assitant