1. Introduction to SOLID Principles of OOPS

08 January 2024 20:32

S - Single Responsibility Principle
O - Open/Closed Principle
L - Liskov Substitution Principle
I - Interface Segregation Principle
D - Dependency Inversion Principle

Advantages:

- Avoid code duplication
- Maintainability
- Easy to Understand
- Flexible Software
- Reduces Complexity

2. Single Responsibility Principle (SRP)

08 January 2024 20:40

A class should have only 1 responsibility to change.

Issue:

Before Application of Single Responsibility Principle

```
8   class Marker {
9       String name;
10      String color;
11      int year;
12      int price;
13
14      public Marker(String name, String color, int year, int price) {
15          this.name = name;
16          this.color = color;
17          this.year = year;
18          this.price = price;
19      }
20  }
21
22  class Invoice {
23      private Marker marker;
24      private int quantity;
25
26      public Invoice(Marker marker, int quantity) {
27          this.marker = marker;
28          this.quantity = quantity;
29      }
30
31      public int calculateTotal() {
32          return marker.price * this.quantity;
33      }
34
35      public void printInvoice() {
36          // print the invoice
37      }
38
39      public void saveToDB() {
40          // save data into DB
41      }
42  }
```

The above Class do not follow SRP as there are 3 reasons to change here. The above Class can change if logic is changed in :

1. calculateTotal()

2. printInvoice()
3. saveToDB()

Solution:

After Application of Single Responsibility Principle

```
4     class Marker {
5         String name;
6         String color;
7         int year;
8         int price;
9
10        public Marker(String name, String color, int year, int price) {
11            this.name = name;
12            this.color = color;
13            this.year = year;
14            this.price = price;
15        }
16    }
17
18    class Invoice {
19        private Marker marker;
20        private int quantity;
21
22        public Invoice(Marker marker, int quantity) {
23            this.marker = marker;
24            this.quantity = quantity;
25        }
26
27        public int calculateTotal() {
28            return marker.price * this.quantity;
29        }
30    }
31
32    class InvoicePrinter {
33        private Invoice invoice;
34
35        public InvoicePrinter(Invoice invoice) {
36            this.invoice = invoice;
37        }
38
39        public void printInvoice() {
40            // print the invoice
41            System.out.println(invoice);
42        }
43    }
```

```java

class InvoiceDAO {
    private Invoice invoice;

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

    public void saveToDB() {
        // save data into DB
        System.out.println(invoice);
    }
}
```

Advantages of Single Responsibility Principle:

1. Easy to Maintain
2. Easy to Understand
3. No impact to other classes if any of printInvoice() or savetoDB() method logic is changed

3. Open/Closed Principle (OCP)

08 January 2024 21:34

Open for extension but Closed for modification.

Issue:

In the code snippet of SRP, refer to the below snippet where a requirement was to add a functionality to write invoice to a file and for which saveToFile method was added to class InvoiceDAO.

Is it following the Open/Closed Principle?

Note: Before adding saveToFile method, the class is in Production (Live) and is stable and tested.

```
class InvoiceDAO {
    private Invoice invoice;

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

    public void saveToDB() {
        // save data into DB
        System.out.println(invoice);
    }

    public void saveToFile() {
        // write data into File
        System.out.println(invoice);
    }
}
```

Solution:

As per OCP, we should not modify the tested class InvoiceDAO, we should have created interfaces such that we can extend the same as below:

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

class InvoiceDatabaseDAO implements InvoiceDAO {
    private Invoice invoice;

    @Override
    public void save(Invoice invoice) {
        this.invoice = invoice;
        // save data into DB
        System.out.println(this.invoice);
    }
}

class InvoiceFileDAO implements InvoiceDAO {
    private Invoice invoice;

    @Override
    public void save(Invoice invoice) {
        this.invoice = invoice;
        // write data into File
        System.out.println(this.invoice);
    }
}
```

4. Liskov Substitution Principle (LSP)

08 January 2024 21:56

If Class B is a sub-type of Class A, then we should be able to replace object of A with B without breaking the behavior of program.

Sub-Class should extend (or add) to the capabilities of its parent Class and not narrow (or remove) it down.

Issue:

The below example shows where the Sub-Class Bicycle narrows down the parent Vehicle capability.

Whenever anytime in the code Bicycle object is used and turnOnEngine method is invoked, then it is going to throw Exception.

This is against Liskov Substitution Principle.

```
static class Vehicle {
    void turnOnEngine() {

    }

    void accelerate() {

    }

    void applyBrake() {

    }
}

static class MotorCycle extends Vehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void turnOnEngine() {
        // turn on engine
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 20;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 20;
    }

}
```

```
static class Car extends Vehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void turnOnEngine() {
        // turn on engine
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 20;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 20;
    }

}

static class BiCycle extends Vehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void turnOnEngine() {
        throw new AssertionError(detailMessage:"There is no engine!");
    }

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 5;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 5;
    }

}
```

Code where problem would happen is below:

```
public static void main(String[] args) {
    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    vehicles.add(new MotorCycle());
    vehicles.add(new Car());
    vehicles.add(new BiCycle());

    for (Vehicle vehicle : vehicles) {
        vehicle.turnOnEngine();
    }
}
```

Solution:

To resolve this, we created Vehicle class which do not have turnOnEngine method and another class EngineVehicle class which extends Vehicle class which has turnOnEngine method.

```
static class Vehicle {

    void accelerate() {

    }

    void applyBrake() {

    }
}

static class EngineVehicle extends Vehicle {

    public void turnOnEngine() {

    }

    void accelerate() {

    }

    void applyBrake() {

    }
}

static class MotorCycle extends EngineVehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void turnOnEngine() {
        // turn on engine
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 20;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 20;
    }

}
```

```
static class Car extends EngineVehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void turnOnEngine() {
        // turn on engine
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 20;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 20;
    }

}

static class BiCycle extends Vehicle {

    boolean isEngineOn = false;
    int speed;

    @Override
    public void accelerate() {
        // increase the speed
        speed = speed + 5;
    }

    @Override
    public void applyBrake() {
        // decrease the speed
        speed = speed - 5;
    }

}
```

Now, we get a compile-time error in the code as Bicycle class do not have turnOnEngine method (1st error) or Bicycle class is not an instance of EngineVehicle class (2nd error).

```java
public static void main(String[] args) {
    accessVehicles();
    accessEngineVehicles();
}

private static void accessVehicles() {
    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    vehicles.add(new MotorCycle());
    vehicles.add(new Car());
    vehicles.add(new BiCycle());

    for (Vehicle vehicle : vehicles) {
        vehicle.turnOnEngine();
    }
}

private static void accessEngineVehicles() {
    List<EngineVehicle> vehicles = new ArrayList<EngineVehicle>();
    vehicles.add(new MotorCycle());
    vehicles.add(new Car());
    vehicles.add(new BiCycle());

    for (EngineVehicle vehicle : vehicles) {
        vehicle.turnOnEngine();
    }
}
```

5. Interface Segregation Principle (ISP)

08 January 2024 22:20

Interfaces should be written such that we should not have to implement un-necessary functionality that Class which implements it, is not intended to do.

Issue:

The below Waiter class had to implement un-necessary functionality such as washDishes, cookFood and decideMenu.
To overcome this scenario, ISP says to segregate or make small-small segments of interfaces.

```java
interface RestaurantEmployee {
    void washDishes();

    void serveCustomers();

    void cookFood();

    void decideMenu();
}

class Waiter implements RestaurantEmployee {

    @Override
    public void washDishes() {
        // not my job (not intended task)
    }

    @Override
    public void serveCustomers() {
        System.out.println(x:"Serving customers and taking orders");
    }

    @Override
    public void cookFood() {
        // not my job (not intended task)
    }

    @Override
    public void decideMenu() {
        // not my job (not intended task)
    }

}
```

Solution:

Now we have segregated the interfaces into multiple segments such that now, Waiter class is much more cleaner and makes more sense.

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

interface ChefInterface {
    void cookFood();

    void decideMenu();
}

interface HelperInterface {
    void washDishes();
}

class Waiter implements WaiterInterface {
    @Override
    public void serveCustomers() {
        System.out.println(x:"Serving customers and taking orders");
    }
}
```

6. Dependency Inversion Principle (DIP)

08 January 2024 22:37

Class should depend on Interfaces rather than on Concrete Classes.

Issue:

The below Macbook class has objects initialized for WiredKeyboard and WiredMouse. But in future, if the Macbook has the capability to have Bluetooth keyboard and Bluetooth mouse, then we need to modify the below files.

```
class WiredKeyboard {
    public void pressKeys() {
        // press keys
    }
}

class WiredMouse {
    public void clickButtons() {
        // click mouse buttons
    }
}

class Macbook {
    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;

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

    public WiredKeyboard getKeyboard() {
        return this.keyboard;
    }

    public WiredMouse getMouse() {
        return this.mouse;
    }
}
```

Solution:

Now we are dependent on interfaces of Keyboard and Mouse from which we have created concrete classes as Wired and Bluetooth Keyboard and Mouse.

This will help Macbook class constructor to have been passed with any concrete classes as per scenarios and this makes the implementation pretty much dynamic in nature.

```
interface Keyboard {
    public void pressKeys();
}

interface Mouse {
    public void clickButtons();
}

class WiredKeyboard implements Keyboard {
    @Override
    public void pressKeys() {
        System.out.println(x:"Press keys");
    }
}

class BluetoothKeyboard implements Keyboard {
    @Override
    public void pressKeys() {
        System.out.println(x:"Press keys");
    }
}

class WiredMouse implements Mouse {
    @Override
    public void clickButtons() {
        System.out.println(x:"Click buttons");
    }
}

class BluetoothMouse implements Mouse {
    @Override
    public void clickButtons() {
        System.out.println(x:"Click buttons");
    }
}
```

```java
class Macbook {
    private final Keyboard keyboard;
    private final Mouse mouse;

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

    public Keyboard getKeyboard() {
        return this.keyboard;
    }

    public Mouse getMouse() {
        return this.mouse;
    }
}
```