KD Knowledge Diet Follow
Mar 27 · 6 min read · Listen

Single Responsibility Principle: What? working as “a software engineer” for three years, you still don’t know this?

Have you ever experienced that code refactoring doesn’t make your code better? Adding spaghetti on top of spaghetti won’t make much of a difference.

Python, Java, Swift whatever language you are using, Single Responsibility Principle can be applied.

If you don’t know Single Responsibility Principle, your code cannot be improved. But it’s okay if you didn’t know. you just have found this article!

Software has an old methodology called the **SOLID** principle.

- **S** stands for **Single Responsibility Principle**
- **O** stands for **Open Closed Principle**
- **L** stands for **Liskov Subsitution Princple**
- **I** stands for **Interface Segregation Principle**
- **D** stands for **Dependency Inversion Principle**

Here, in this article, I will introduce you **Single Responsibility Principle.**

So then, what does it mean "Single Responsibility Principle?"

> *Every Software Component should have only one responsibility.*

"Software component" could refer to a class or a function or a method or even a module.

We can say then,

- "a class should have one responsibility.".
- "a function should tackle only one task" .

[1] Always Aim for High Cohesion

> *Cohesion is the degree to which the various parts of a software component are related*

Before starting, take a look at the code below. And guess for a second what is wrong with the code

```
// - Measurements of squares
// - Rendering images of squares
// Before Being Refactored
class Square {

    int side = 5;

    public int calculateArea() {
        return side * side;
    }

    public int calculatePerimeter() {
        return side * 4;
    }

    public void draw() {
        if (highResolutionMonitor) {
            // Render a high resolution image of a square
        } else {
            // Render a normal image of a square
        }
    }

    public void rotate(int degree) {
        // Rotate the image of the square clockwise to
        // the required degree and re-render
    }

}
```

Our `Square` class has now two responsibilities.

Not only does square take care of **Measurements of squares,** but it also **renders images of squares.**

For Solid Principle, then how should we write the code?

This is the refactored version

```
/** After Refactoring **/
// Responsibility: Measurements of squares
// After Refactoring
class Square {
    public int calculatePerimeter() {
        return side * 4;
    }

}

// Responsibility: Rendering images of squares
class SquareUI {

    public void draw() {
        if (highResolutionMonitor) {
            // Render a high resolution image of a square
        } else {
            // Render a normal image of a square
        }
    }

    public void rotate(int degree) {
        // Rotate the iamge of the square clockwise to
        // the required degree and re-render
    }

}
```

As you can see **Square** now only tackles measurements, and **SquareUI** handles only rendering.

If you find it ambiguous to divide by responsibility, think about code cohesion.

Here,

- `calculateArea()` and `calculatePerimeter()` are certainly related with the measurement of square
- `draw()` and `rotate()` are related with rendering.

[2] Avoid Tight Coupling

> *Coupling is defined as the level of inter dependency between various software*

```
/** Before Refactoring **/
// Tightly Coupled
// Responsibility: Handle core student profile data
// Responsibility: Handle Database Operations
class Student {

    private String studentId;
    private Date studentDOB;
    private String address;

    public void save() {
        // Serialize object into a string representation
        String objectStr = MyUtils.serialzieIntoAString(this);
        Connection connection = null;
        Statement stmt = null;

        // We are using MYSQL
        // What if I want to use another database?
        // This is why Tight Coupling is bad practices for
prgramming
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDb",
"root", "password");
            stmt = connection.createStatement();
            stmt.execute("INSERT INTO STUDENTS VALUES (" + objectStr
+ ")");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    public String getStudentId() {
        return studentId;
    }

    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }

}
```

This code, especially `save()` method is a big problem. This only works with ‘MySQL’. What if you have to use a different database?

How do I refactor?

```
/** After Refactoring **/
// Responsibility: Handle Database Operations
public class StudentRepository {
    public void save() {
        // Serialize object into a string representation
        String objectStr = MyUtils.serialzieIntoAString(this);
        Connection connect = null;
        Statement stmt = null;

        // We are using MYSQL
        // What if I want to use another database?
        // This is why Tight Coupling is bad practices for
prgramming
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDb",
"root", "password");
            stmt = connection.createStatement();
            stmt.execute("INSERT INTO STUDENTS VALUES (" + objectStr
+ ")");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}

// Responsibility: Handle core student profile data
class Student {

    private String studentId;
    private Date studentDOB;
    private String address;

    public void save() {
        new StudentRepository().save(this);
    }

    public String getStudentId() {
        return studentId;
    }

    public void setStudentId(String studentId) {
        this.studentId = studentId;
```

Now, we have two classes instead of one. `StudentRepository` is for handling database operations and `Student` is for handling student profile data.

In this way, you can change your database on `StudentRepository` anytime you need to change databases. And `Student` 's `save()` method won’t change its functionality when you implement another database in `StudentRepository` .

[3] Software component needs only one reason to change

> ***Software is never dormant. It always keeps changing.***

Software changes are not unusual. Sometimes it’s so stressful because when you change code, there is a high probability that a bug will occur. The more changes you make, the more bugs you get. Still, is there any way to minimize the bugs as much as possible?

Let’s see our refactored code. Consider the possibilities of what might be changed in the future.

```
// Responsibility: Handle Database Operations
// Changes might occur
// 1. A change in the database backend, as advised by the technical
team.
public class StudentRepository {
    public void save() {
        // Serialize object into a string representation
        String objectStr = MyUtils.serialzieIntoAString(this);
        Connection connect = null;
        Statement stmt = null;

        // We are using MYSQL
        // What if I want to use another database?
        // This is why Tight Coupling is bad practices for
prgramming
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection =
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}

// Responsibility: Handle core student profile data
// Changes might occur
// 1. A change in Student Data Format
class Student {

    private String studentId;
    private Date studentDOB;
    private String address;

    public void save() {
        new StudentRepository().save(this);
    }

    public String getStudentId() {
        return studentId;
    }

    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }

}
```

- `StudentRepository` is likely to change its database.
- `Student` is likely to change its student format.

Likewise, all software components should have only one reason to change.

Conclusion

Note that ***SRP is not the solution that fits all***. It is more of an abstract concept. Because **“a single responsibility”** you define can contain many responsibilities. Therefore, how to define responsibility is the developer’s ability.

Other Articles for Solid Principles

Single Responsibility Principle: What? working as “a software engineer” you don’t know it?!?!

Have you ever experienced that code refactoring doesn’t make your code better? Adding spaghetti on top of spaghetti…

paigeshin1991.medium.com

Open Closed Principle: Make your code cost-free and flexible

Software is never dormant. It’s constantly changing. The annoying fact about the software development is that every…

paigeshin1991.medium.com

Liskov Substitution Principle: Top Developer’s technique which improves 2.5x

As you write code, doesn’t anyone ever question whether your code is logically correct or consistent enough? To solve…

paigeshin1991.medium.com

Interface Segregation Principle: Your only way to be a super competent developer

During development, the code tends to be long. At some point, the codebase becomes so bloated that you can’t handle it…

paigeshin1991.medium.com

Dependency Inversion Principle: How Google Developers write code

To become a high-paid developer, you need to learn TDD. Basically