Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

Maven

Testing (Foundation)

Java Intermediate

○ Optionals
○ JDBC CRUD
○ Exceptions
○ SOLID Principles
○ Single Responsibility
○ Open/Closed
○ Liskov Substituiton
○ Interface Segregation
○ Dependency Inversion
○ Best Practice
○ Design Patterns
○ Creational Design Patterns
○ Structural Design Patterns
○ Behavioural Design Patterns
○ Collection & Map
○ HashSets
○ HashMaps
○ Enums
○ Logging
○ Generics
○ Lambda Expressions
○ Streams
○ Complexity
○ Input and Output
○ Local Type Inference

HTML

CSS

Single Responsibility

Contents

Overview

In object-oriented programming, the first of the **SOLID Principles** is **S** - which stands for **Single Responsibility**.

When talking about *responsibilities* in programming, we essentially talk about *reasons for something to change*.

If a class has a single responsibility, then it only has one reason to change.

This is the essence of *Single Responsibility* - a class should only ever have one purpose, and therefore only one reason for it to ever need to change.

If we have two reasons to change a class, we should split the functionality into two classes, with each class handling only one responsibility.

When breaking classes down, it should be done in a way that *decreases coupling* (how much two classes interdepend on each other) and *increases cohesion* (how much the elements inside that class belong together in that class).

That way, in the future, if we need to make a change, then we would make it in the class which handles it.

But if we needed to make a change in a class that had more than one responsibility, that change might then affect functionality which relates to another responsibility of that class instead.

Single Responsibility In Action

The `Car.java` class

Let's take a look at how a class might come to have multiple responsibilities, and how we can fix it.

Take a look at this **Car.java** class:

▶ Car.java

The above `Car.java` class looks fine, logically speaking - it's got all the usual information we'd expect from a car.

However, this flies in the face of the *Single Responsibility Principle*.

The way to deal with this issue conceptually is to ask yourself questions like:

Should a `Car` be responsible for driving itself?

What we can say in response to this question is that the *functionality which deals with driving the `Car`* should not be the responsibility of the `Car` object itself.

- Javascript
- Spring Boot
- Selenium
- Sonarqube
- Advanced Testing (Theory)
- Cucumber
- MongoDB
- Express
- NodeJS
- React
- Express-Testing
- Networking
- Security
- Cloud Fundamentals
- AWS Foundations
- AWS Intermediate
- Linux
- DevOps
- Jenkins Introduction
- Jenkins Pipeline
- Markdown
- IDE Cheatsheet

So then we can ask another question:

What should be responsible for driving the `Car`?

Perhaps, in this case, this should be the responsibility of something like a driver, or a remote control.

Ultimately, the `Car.java` class should **only** contain the core attributes and functions of a `Car` object.

Splitting up the `Car.java` class

We can fix this by splitting the functionality of the `Car.java` class into two:

- `Car.java` will deal with **only** storing information about a `Car` object.
- `Driver.java` will deal with changing the information about a `Car` (that goes beyond the usual getters and setters).

▶ Driver.java

Now we'll update the content of the `Car.java` class accordingly:

▶ Car.java

The `Car.java` class now only has the single responsibility of maintaining the core attributes of a `Car`, while the `Driver.java` class takes care of other things, like updating the mileage of the current car.

If we wanted to extend out how a `Car` is maintained - changing the tyres, adding a spoiler, lowering the ride height, or whatever else - we no longer need to do that inside the `Car` class itself.

Instead, we could:

- *update* the `Driver.java` class (with things a `Driver` would do)
- *create* a new class for any other functionality (like a `Mechanic.java` class for things a `Mechanic` would do)

Tutorial

There is no tutorial for this module.

Exercises

Mechanic

Try adding a `Mechanic.java` class to this setup.