SOLID Design Principles Explained: The Single Responsibility Principle

stackify.com/solid-design-principles

April 1, 2020

SOLID is one of the most popular sets of design principles in object-oriented software development. It's a mnemonic acronym for the following five design principles:

All of them are broadly used and worth knowing. But in this first post of my series about the SOLID principles, I will focus on the first one: the Single Responsibility Principle.

Robert C. Martin describes it as:

> A class should have one, and only one, reason to change.

Even if you have never heard of Robert C. Martin or his popular books, you have probably heard about and used this principle. It is one of the basic principles most developers apply to build robust and maintainable software. You can not only apply it to classes, but also to software components and microservices.

Benefits of the Single Responsibility Principle

Let's address the most important questions before we dive any deeper into this design principle: Why should you use it and what happens if you ignore it?

The argument for the single responsibility principle is relatively simple: it makes your software easier to implement and prevents unexpected side-effects of future changes.

Frequency and Effects of Changes

We all know that requirements change over time. Each of them also changes the responsibility of at least one class. The more responsibilities your class has, the more often you need to change it. If your class implements multiple responsibilities, they are no longer independent of each other.

You need to change your class as soon as one of its responsibilities changes. That is obviously more often than you would need to change it if it had only one responsibility.

That might not seem like a big deal, but it also affects all classes or components that depend on the changed class. Depending on your change, you might need to update the dependencies or recompile the dependent classes even though they are not directly affected by your change. They only use one of the other responsibilities implemented by your class, but you need to update them anyway.

In the end, you need to change your class more often, and each change is more complicated, has more side-effects, and requires a lot more work than it should have. So, it's better to avoid these problems by making sure that each class has only one responsibility. Besides, if you want to gain a better understanding of what's happening in your application, you can use Retrace's code profiling solution.

Easier to Understand

The single responsibility principle provides another substantial benefit. Classes, software components and microservices that have only one responsibility are much easier to explain, understand and implement than the ones that provide a solution for everything. This reduces the number of bugs, improves your development speed, and makes your life as a software developer a lot easier.

However, make sure to not oversimplify your code. Some developers take the single responsibility principle to the extreme by creating classes with just one function. Later, when they want to write some actual code, they have to inject many dependencies which makes the code very unreadable and confusing.

Therefore, the single responsibility principle is an important rule to make your code more understandable but don't use it as your programming bible. Use common sense when developing code. There is no point in having multiple classes that just contain one function.

A Simple Question to Validate Your Design

Unfortunately, following the single responsibility principle sounds a lot easier than it often is.

If you build your software over a longer period and if you need to adapt it to changing requirements, it might seem like the easiest and fastest approach is adding a method or functionality to your existing code instead of writing a new class or component. But that often results in classes with more than responsibility and makes it more and more difficult to maintain the software.

You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?

If your answer includes the word "and", you're most likely breaking the single responsibility principle. Then it's better to take a step back and rethink your current approach. There is most likely a better way to implement it.

To give a more concrete example, let's assume we have a class for an employee that holds methods for calculating and reporting their salary. In other words, calculating salary can be classified as reading data and further manipulating it.

While reporting salary is a data persistence operation where the data is stored in some storage medium. If we follow Martin's single responsibility principle, these classes should be split up as the business functions are quite different.

Next, let's look at some real-world Java examples about the single responsibility principle.

Real-World Examples of the Single Responsibility Principle

You can find lots of examples of all SOLID design principles in open source software and most well-designed applications. Such as your Java persistence layer and the popular frameworks and specifications, which you most likely used to implement it.

One of them is the Java Persistence API (JPA) specification. It has one, and only one, responsibility: Defining a standardized way to manage data persisted in a relational database by using the object-relational mapping concept.

That's a pretty huge responsibility. The specification defines lots of different interfaces for it, specifies a set of entity lifecycle states and the transitions between them, and even provides a query language, called JPQL.

But that is the only responsibility of the JPA specification. Other functionalities which you might need to implement your application, like validation, REST APIs or logging, are not the responsibility of JPA. You need to include other specifications or frameworks which provide these features.

If you dive a little bit deeper into the JPA specification, you can find even more examples of the single responsibility principle.

JPA EntityManager

The *EntityManager* interface provides a set of methods to persist, update, remove and read entities from a relational database. Its responsibility is to manage the entities that are associated with the current persistence context.

That is the only responsibility of the *EntityManager*. It doesn't implement any business logic or validation or user authentication. Not even the application-specific domain model, which uses annotations defined by the JPA specification, belongs to the responsibility of the *EntityManager*. So, it only changes, if the requirements of the general persistence concept change.

JPA AttributeConverter

The responsibility of the *EntityManager* might be too big to serve as an easily understandable example of the single responsibility principle. So, let's take a look at a smaller example: an *AttributeConverter* as the JPA specification defines it.

The responsibility of an *AttributeConverter* is small and easy to understand. It converts

a data type used in your domain model into one that your persistence provider can persist in the database. You can use it to persist unsupported data types, like your favorite value class, or to customize the mapping of a supported data type, like a customized mapping for enum values.

Here is an example of an *AttributeConverter* that maps a *java.time.Duration* object, which is not supported by JPA 2.2, to a *java.lang.Long*: The implementation is quick and easy. You need to implement that AttributeConverter interface and annotate your class with a em>@Converter</em annotation.

```
@Converter(autoApply = true)
public class DurationConverter implements AttributeConverter<Duration, Long> {
   @Override
   public Long convertToDatabaseColumn(Duration attribute) {
     return attribute.toNanos();
   }

   @Override
   public Duration convertToEntityAttribute(Long duration) {
      return Duration.of(duration, ChronoUnit.NANOS);
   }
}
```

As you can see in the code sample, the *DurationConverter* implements only the two required conversion operations. The method *convertToDatabaseColumn* converts the *Duration* object to a *Long*, which will be persisted in the database. And the *convertToEntityAttribute* implements the inverse operation.

The simplicity of this code snippet shows the two main benefits of the single responsibility principle. By limiting the responsibility of the *DurationConverter* to the conversion between the two data types, its implementation becomes easy to understand, and it will only change if the requirements of the mapping algorithm get changed.

Spring Data Repository

The last example to talk about is the Spring Data repository. It implements the repository pattern and provides the common functionality of create, update, remove, and read operations. The repository adds an abstraction on top of the *EntityManager* with the goal to make JPA easier to use and to reduce the required code for these often-used features.

You can define the repository as an interface that extends a Spring Data standard interface, e.g., *Repository*, *CrudRepository*, or *PagingAndSortingRepository*. Each interface provides a different level of abstraction, and Spring Data uses it to generate implementation classes that provide the required functionality.

The following code snippet shows a simple example of such a repository. The *AuthorRepository* extends the Spring *CrudRepository* interface and defines a repository for an *Author* entity that uses an attribute of type *Long* as its primary key.

```
interface AuthorRepository extends CrudRepository<Author, Long> {
   List findByLastname(String lastname);
}
```

Spring's *CrudRepository* provides standard CRUD operations, like a *save* and *delete* method for write operations and the methods *findById* and *findAll* to retrieve one or more *Author* entities from the database.

The *AuthorRepository* also defines the *findByLastName* method, for which Spring Data generates the required JPQL query to select *Author* entities by their *lastname* attribute.

Each repository adds ready-to-use implementations of the most common operations for one specific entity. That is the only responsibility of that repository.

Similar to the previously described *EntityManager*, the repository is not responsible for validation, authentication or the implementation of any business logic. It's also not responsible for any other entities. This reduces the number of required changes and makes each repository easy to understand and implement.

Summary

The single responsibility principle is one of the most commonly used design principles in object-oriented programming. You can apply it to classes, software components, and microservices.

To follow this principle, your class isn't allowed to have more than one responsibility, e.g., the management of entities or the conversion of data types. This avoids any unnecessary, technical coupling between responsibilities and reduces the probability that you need to change your class. It also lowers the complexity of each change because it reduces the number of dependent classes that are affected by it. However, be reasonable.

There is no need to have multiple classes that all hold just one function. Try to find the right balance when defining responsibilities and classes.