# SOLID Design Principles Explained – The Single Responsibility Principle

THORBEN JANSSEN | MARCH 8, 2018 |
DEVELOPER TIPS, TRICKS & RESOURCES (HTTPS://STACKIFY.COM/DEVELOPERS/)

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:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

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 (http://blog.cleancoder.com) 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 (https://stackify.com/what-are-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.
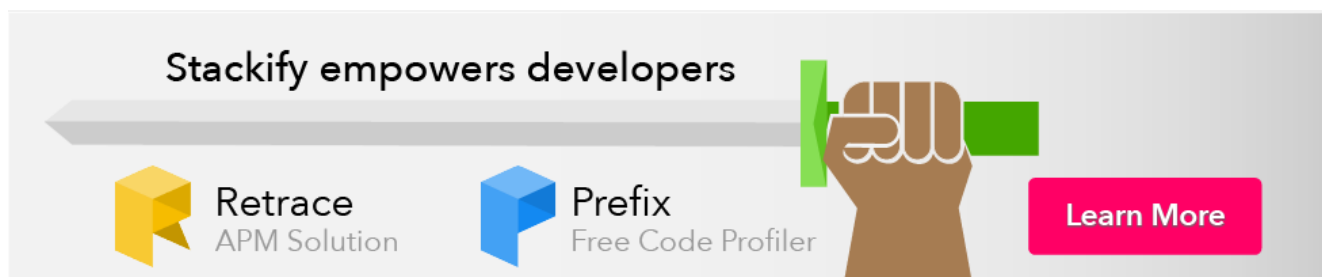


(https://cta-service-cms2.hubspot.com/ctas/v2/public/cs/c/?cta_guid=eb8ba9dc-2898-4c16-a893-432ae7a086a9&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Fsolid-design-principles%2F&redirect_url=APefjpG9Y9D5t1aH0g7TkWFC4fNqLqRlq1QzhAZRJoGujTQdUJm2FNAI_(79ff-45ec-aedb-ab341c31ed90&hsutk=c7a000001e2a1b7a18e801660f1dffec&utm_referrer=https%3A%2F%2Fdzon

# 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.

## 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 take a step back and rethink your current approach. There is most likely a better way to implement it.

## 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 (https://stackify.com/content/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 (https://www.jcp.org/en/jsr/detail?id=338). 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 (http://thoughts-on-java.org/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 (https://stackify.com/multiple-media-types-java-microservices-resteasy/) or logging (https://stackify.com/application-logging-apm-strategy/), 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_ (https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html) 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* (http://www.thoughts-on-java.org/jpa-21-how-to-implement-type-converter/) 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 (https://www.thoughts-on-java.org/jpa-21-type-converter-better-way-to).

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 *@Converter* annotation.

```
@Converter(autoApply = true)
public class DurationConverter implements AttributeConverter<Dura
tion, 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
(https://docs.spring.io/spring-
data/jpa/docs/current/reference/html/#repositories.core-concepts). 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
(https://stackify.com/what-are-crud-operations/) 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.

Read more: Get a primer on OOP Concepts (https://stackify.com/oops-concepts-in-java/) in Java and learn about the 4 main concepts: a (https://stackify.com/oop-concept-abstraction/)bstraction (https://stackify.com/oop-concept-abstraction/), encapsulation (https://stackify.com/oop-concept-for-beginners-what-is-

encapsulation/), inheritance (https://stackify.com/oop-concept-inheritance/), and polymorphism (https://stackify.com/oop-concept-polymorphism/).

---

**👤 About the Author**    **≣ Latest Posts**

### About Thorben Janssen

Thorben is an independent trainer and author of the Amazon bestselling book *Hibernate Tips - More than 70 solutions to common Hibernate problems.*He writes about Java EE related topics on his blog Thoughts on Java (https://www.thoughts-on-java.org).

**f**

(https://plus.google.com/u/0/b/109323639830862412239/1093236;

**🐦** (http://thjanssen123)   **in**

(https://www.linkedin.com/in/thorbenjanssen/)   **g+**

(https://plus.google.com/u/0/b/109323639830862412239/1093236; rel=author)

---

# Improve Your Code with Retrace APM

Stackify's APM tools are used by thousands of .NET, Java, and PHP developers all over the world.
Explore Retrace's product features to learn more.

(/retrace-application-performance-management/)

Stackify (



(/retrace-code-profiling/)

Code Profiling (https://stackify.com/retrace-code-profiling/)



(/retrace-error-monitoring/)

Error Tracking (https://stackify.com/retrace-error-monitoring/)



(/retrace-log-management/)

Centralized Logging (https://stackify.com/retrace-log-management/)

(/retrace-app-metrics/)

App & Server Metrics (https://stackify.com/retrace-app-metrics/)

Learn More (/retrace/)

# Search Stackify

Search

## Topics/Keywords

ASP.NET

.NET Core

Java

Azure

AWS

Cloud

Product Updates

App Monitoring

App Performance Tips

Error Handling

Logging Tips

DevOps

# Popular Posts

APM Pricing Is Now Affordable for All Developers, and Why They Should Care

What is Application Performance Management? Overview, Common Terms, and 10 Critical APM Features

Understanding # of Exceptions Thrown / Sec and How to Find Exceptions

6 Types of ROI for APM (Application Performance Management Tools)

How to Monitor Web Services Performance with Retrace

## Get In Touch

Contact Us

Request a Demo

8900 State Line
Rd #100
Leawood, KS
66206

816-888-5055

## Product

Retrace APM

Prefix

.NET Monitoring

Java Monitoring

PHP Monitoring

Retrace vs New
Relic

Retrace vs
Application

## Solutions

Log Management

Application
Performance
Management

Application
Metrics

Azure Monitoring

Error Tracking

Code Profiling

## Resources

Pricing

Case Studies

Blog

Podcast

Documentation

Free eBooks

Videos

Ideas Portal

APM ROI

## Company

About Us

GDPR

Careers

Security
Information

Terms &
Conditions

Privacy Policy

Stackify

Insights

What is APM?

© 2018 Stackify