Mastering the Single Responsibility Principle

Learn how to apply SRP effectively through real-world examples, common anti-patterns, and proven design practices.

Author: Mr. Jin Vincent N. Necesario

Image by Stefan Cosma On Unsplash

Introduction

Most organizations expect software engineers to explain the “Single Responsibility” design principle in an interview and expecting they (the organizations) to apply it in their day-to-day coding.

However, in some cases, we must acknowledge that violating the “Single Responsibility” design principle is necessary; with a good plan and design, we can identify the tradeoffs.

That’s why in this article, we’ll try to answer what the “Single Responsibility” design principle is, its importance, and when to violate and not to violate it.

Author: Mr. Jin Vicent N. Necesario

What’s Single Responsibility?

To define this in our own words, as a software engineer, we can say: “You should make a class that should have only one reason to change”. Thus, it means that every class should have only “one responsibility”.

Let’s break it down:

- Responsibility means a reason to change (*Responsibility* = *Reason to Change*).
- Each responsibility is an axis of change.
- A class or module should focus on one thing and do it well.

Again, a class or module should focus on one thing and do it well; that way, it is stable, focused, and easier to maintain.

Why is single responsibility important?

There are three things that we can say affect our application when creating or following the “Single Responsibility” design principle.

Let’s break them one by one:

Maintainability

Imagine your application has a class with well-defined responsibility, I can guess that they are also easier to understand and modify.

Making changes to your application will be smooth and have minimal impact on the code.

I still remember the days when maintaining WinForms (Desktop client applications) and Web Forms (ASP.NET 2.0) would always result in “side effects” or some form of “coupling impact”.

That’s why it is essential to practice this design principle to lessen these kinds of issues when maintaining an application.

To note, “side effects” refer to consequences or behaviors that result from a change, whereas “coupling impact” occurs when components are tightly coupled, affecting each other and forcing changes in different components.

Testability

Imagine your application following this design principle; just what would your unit test look like? Easy to test, right? Classes with a single focus are easier to write unit tests for.

Again, I’ll share a story.

I still remember that Web Forms (ASP.NET 2.0) weren’t testable (the .aspx pages). This was due to the design philosophy of that time (early 2000s), which was partly because Web Forms were designed to mimic Windows Forms, and there was no focus on testability back then.

However, unlike today, having “testability” in our applications is essential. By following the “Single Responsibility” design principle, we can make our journey easier when writing unit tests.

Flexibility

Tada, flexibility, when changing one area of your application’s code doesn’t affect unrelated components or other areas of your application.

Again, in practice, this looks great; however, the “Single Responsibility Principle” organizes responsibility but doesn’t prevent external dependencies and shared mutable state, which may introduce “side effects”.

Let me know what you think in the comment section below.

Use Cases Not to Follow Single Responsibility

Here are some of the lists we may not follow, specifically the “Single Responsibility Principle” (SRP).

Or I could say when to violate SRP 😊 .

A good example of violating SRP is Microsoft.Extensions.Logging is problematic because it involves message formatting, routing, and output to the console, file, or event viewer, which violates SRP due to its multiple reasons for change.

Small or Simple Applications

For small or simple applications, or demo purposes. SRP could be an overkill; creating multiple classes for “clean separation” may over-engineer the solution.

So, don’t overthink your demo applications.

Framework or Library Constraints

In some cases, opinionated frameworks such as ASP.NET MVC Core, Angular, or EF Core often guide developers by conventions.

A good example might be an ASP.NET Core Controller.

Typically, a Controller would handle incoming HTTP requests, validate input, log actions, and return responses. Now, from an SRP perspective, this is too much responsibility.

Let’s say just that the controller looks “bloated”.

But here are some tradeoffs: it's fast to develop, easy to understand for newcomers in ASP.NET MVC Core, and it has less boilerplate, with fewer files and classes.

What I’m trying to say is that you should respect SRP within framework constraints better. You can refactor responsibilities out of Controllers, such as logging, validation, and exception handling.

Author: Mr. Jin Vicent N. Necesario

Legacy Codebases

When maintaining an old monolithic application, applying SRP can require massive refactoring, potentially leading to more bugs and increased risks, and may even lead you down a rabbit hole.

However, if you are that strong, and what an adventure, I suggest that incremental changes are safer than enforcing SRP from start to finish.

Good Libraries to check that use SRP

Within the world of .NET, numerous libraries are available that utilize SRP to some extent.

If you are familiar with Stream, it is a good example.

Stream is an abstract base class that only defines the contract, such as Read, Write, Seek, etc.

Concrete implementations, such as FileStream, MemoryStream, and NetworkStream, have their focused responsibilities.

The HttpClient, to a certain degree, respects SRP, and its primary responsibility is to send HTTP requests and receive HTTP responses. Although it can be misused, its core focus is on HTTP communications. You can also configure other concerns, such as logging, caching, retry using delegate handlers, or dependency injections.

Summary

You are now here, at last, and at least you have realized how important the Single Responsibility Design Principle is. As we have discussed, we have covered what it is, why it is essential, the use cases that violate it, and good libraries that use SRP to a reasonable extent.

Once again, I hope you have enjoyed this article as I have enjoyed writing it.

Stay tuned for more. Until next time, happy programming!

Please don't forget to bookmark, share, like, subscribe, and comment.

Cheers! And thank you!

Author: Mr. Jin Vicent N. Necesario