Caleb Mantey

Posted on 10 févr.

Solid Design Principles In Javascript (Part 1) — Single Responsibility Principle

#javascript #soliddesignpatterns #designpatterns #typescript

Hi i am Mantey Caleb a software engineer based in Accra, Ghana. One of the things i have been doing recently is reading about design patterns and practicing a lot. I tried applying some of the principles in some of my personal work and i am impressed by how much my code looks cleaner and maintainable.
But don't get me wrong, some of these principles should not just be applied to any problem you come across. They are a good skill to have as a developer but you should know when to use them and not just applying them to every problem you encounter.

In this article i will talk about solid design principles with examples in javascript.

SOLID Design Principles

SOLID PRINCIPLE is usually referred to as the first five principles of object oriented design. This principle was formulated by Robert C. Martin (also known as Uncle Bob). In this article i will be using javascript to explain certain concepts. Javascript doesn't support features like interfaces and abstract classes but with the addition of typescript we can write javascript like we do in other languages like c# and java. So in this article we will be using typescript too. Solid principles helps in reducing tight coupling between classes in our code. Tight coupling is when a group of classes highly depend on one another. Loose coupling is the opposite of tight coupling and this approach makes our code more reusable, readable, flexible, scalable and maintainable. It is advisable to avoid tight coupling as much as possible and always make your code loosely coupled.

SOLID stands fo

S — Single Responsibility Principle
O — Open-closed Principle
L — Liskov Substitution Principle
I — Interface Segregation Principle
D — Dependency Inversion Principle

In this article we will talk about only the single responsibility principle and reserve the rest for another article.
If you want to see a full explanation of all these principles with code examples, you can check it on my GitHub.

View Github Code

Single Responsibility Principle

Single responsibility principle states that

A class should have one and only one responsibility. Which means your class should have only one job or task.

Consider this example, where we have a mailer class that connects to an smtp service, takes an email, processes it and sends the email as either text or html. Now Let's see what this class has to do to get the job done.

```
class Mailer{
        constructor(mail){
            this.mail = mail
            this.smtpService = this.smtp_service_connection()
        }
        smtp_service_connection(){
            // Connects to smtp service
        }
        send(){
            this.smtpService.send(this.format_text_mail())
             this.smtpService.send(this.format_html_mail())
        }
        format_text_mail(){
            // formats to text version of mail
            this.mail = "Email For You \n" + this.mail;
            return this.mail;
        }
        format_html_mail(){
            // formats to html version of mail
             this.mail = `<html>
            <head><title>Email For You</title></head>
            <body>${this.mail}</body>
            </html>`;
            return this.mail;
        }
    }
   const mailer = new Mailer("hello kwame");
   mailer.send();
```

This code does not follow the single responsibility principle.

The mailer class is responsible for doing all the following

- Connects to an smtp service
- Format the mail in text format
- Format the mail in html format
- Sending the mail

This will make the `Mailer` class very difficult to maintain. Let's say for example we want to change the smtp provider we are using, we will have to come into this class and do some changes to the `smtp_service_connection` method and this can get tricky and messy if the new provider does not implement a `send` method but a `deliver` method, we then will have to

also come and change this line `this.smtpService.send(this.format_html_mail())` in our `send` method to `this.smtpService.deliver(this.format_html_mail())`. All these is a result of the fact that our class is not performing only one functionality.

Better Approach

Mailer

```
class Mailer{
        constructor(mail, mailerFormats){
            this.mail = mail
            this.mailerFormats = mailerFormats
            this.smtpService = new MailerSmtpService()
        }

         send(){
            // Loops through mail formats and calls the send method
            this.mailerFormats.forEach((formatter) =>
           this.smtpService.send(formatter.format(this.mail)))
        }
}
```

MailerSmtpService

```
class MailerSmtpService{
        constructor(){
           this.smtp_con = this.smtp_service_connection()
        }

        send (mail){
            this.smtp_con.send(mail)
            // can easily change to be this (smtp_con.deliver(mail))
           // if a service requires this implementation
        }

        smtp_service_connection(){
            // Connects to smtp service
        }
}
```

HtmlFormatter

```
class HtmlFormatter{
        constructor(){
        }

        format(mail){
             // formats to html version of mail
              mail = `<html>
            <head><title>Email For You</title></head>
            <body>${mail}</body>
            </html>`;
            return mail;
        }
}
```

TextFormatter

```
class TextFormatter{
        constructor(){
        }

        format(mail){
             // formats to text version of mail
```

```
            mail = "Email For You \n" + mail;
            return mail;
        }
}
```

A more better approach is seen above where we divide all the tasks into separate classes.

We will now have the following.

- A class that connects to the smtp service (MailerSmtpService)
- A class that formats our mail in text (TextFormatter)
- A class that formats our mail in html (HtmlFormatter)
- A class responsible for sending the mail (Mailer)

You can see now the code looks better and our smtp service can be changed easily in only one class which does not affect the other parts of the mailing systems behaviour. If we use a new smtp service and it implements a `deliver` method instead of a `send` method then we only have to change one method (we change `this.smtp_con.send(mail)` to `this.smtp_con.deliver(mail)`) in the `MailerSmtpService` class. This will not affect other parts of our application and our app will still function properly. The `Mailer` class takes an instance of a `MailerSmtpService` class and only sends a mail (NOTE: It is performing one and only one job to send mail)

Also our `HtmlFormatter` and `TextFormatter` are doing just one thing formatting the mail in the right format.

Now we can send an email by simply doing this

```
const mailer = new Mailer("hello kwame", [new HtmlFormatter(), new
TextFormatter()])
mailer.send();
```

Thanks for your time. Give me a follow or a like if you loved this article.
In the next article we expand on this example by focusing on the second and third principle (**Open-Closed Principle** and **Liskov Substitution Principle**) to make our code even better.