Armakuni

Separation of concerns

Tim Long and Jon Acker
armakuni.com

Contents

Introduction	**3 mins**
Discussion and examples	**15 mins**
Awesome interactive Stuff!!	**~60 mins**
Reflection and discussion	**10 mins**

What is Separation of concerns?

"A class should have only one reason to change"

Robert C Martin

SOLID Principles

Single responsibility principle

Open closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

But what does this mean and how do I apply it?

An example...

Developer A has picked up a ticket that requires some new functionality to be added to an application. He needs to add functionality to take some numbers and return the prime numbers in the list as JSON.

And so Developer A writes the following...

```go
func outputPrimes(numbers []int) []byte {
    var primes []int
    for _, number := range numbers {
        if number <= 1 {
            continue
        }
        for x := 2; x < number; x++ {
            if number%x == 0 {
                primes = append(primes, number)
            }
        }
    }
    type output struct {
        Output []int `json:"output"`
    }
    response, _ := json.Marshal(output{primes})
    return response
}
```

Spot any problems?

What happens when we need to return a list of numbers squared?

What happens when we need to return the list in comma separated values?

It has multiple reasons to change!

In other words, this function is responsible for both calculating the values from the list of numbers AND rendering the response!

What should we do differently?

Here we have two concerns:

- Calculating numbers from a list of numbers
- Rendering a response

```go
func getPrimes(numbers []int) []int {
    var primes []int
    for _, number := range numbers {
        if number <= 1 {
            continue
        }
        for x := 2; x < number; x++ {
            if number%x == 0 {
                primes = append(primes, number)
            }
        }
    }
    return primes
}

func numbersToJson(numbers []int) []byte {
    type output struct {
        Output []int `json:"output"`
    }
    response, _ := json.Marshal(output{numbers})
    return response
}
```

Now each function only has one reason to change!

Bear in mind, you can also go too far with single responsibility

A good rule of thumb is a function should not be more than 10-15 lines long

It is also worth thinking about the testing required against the function

This principle is about people.

When you write a software module, you want to make sure that when changes are requested, those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function.

On to the workshop!

Remember!

- No pressure or expectation
- Enjoy yourself
- Maybe learn something

Exercise goals

The exercise is based around a legacy transformer function that has been written. Its purpose is to read JSON/YAML files (or from STDIN), transform and output to JSON or YAML files/STDOUT. Each one of these bits of functionality has been added over the years to where it is now, a big old pile of spaghetti. Further information can be found in the README.md

The goals of this exercise are:

- Pair/mob if possible
- Identify the different concerns/responsibilities that this function has
- Drive out each concern into its own function using TDD
- Refactor the functions you have made to reduce their complexity
- Reduce complexity of the Transform function and monitor it as you change it
- Behold the magnificent functions you have created