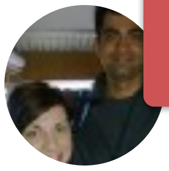

Technical Writer @fixate.io Tweet me at @nerdokto.
Nov 12, 2017 · 4 min read

Understanding SOLID Principles: Interface Segregation Principle

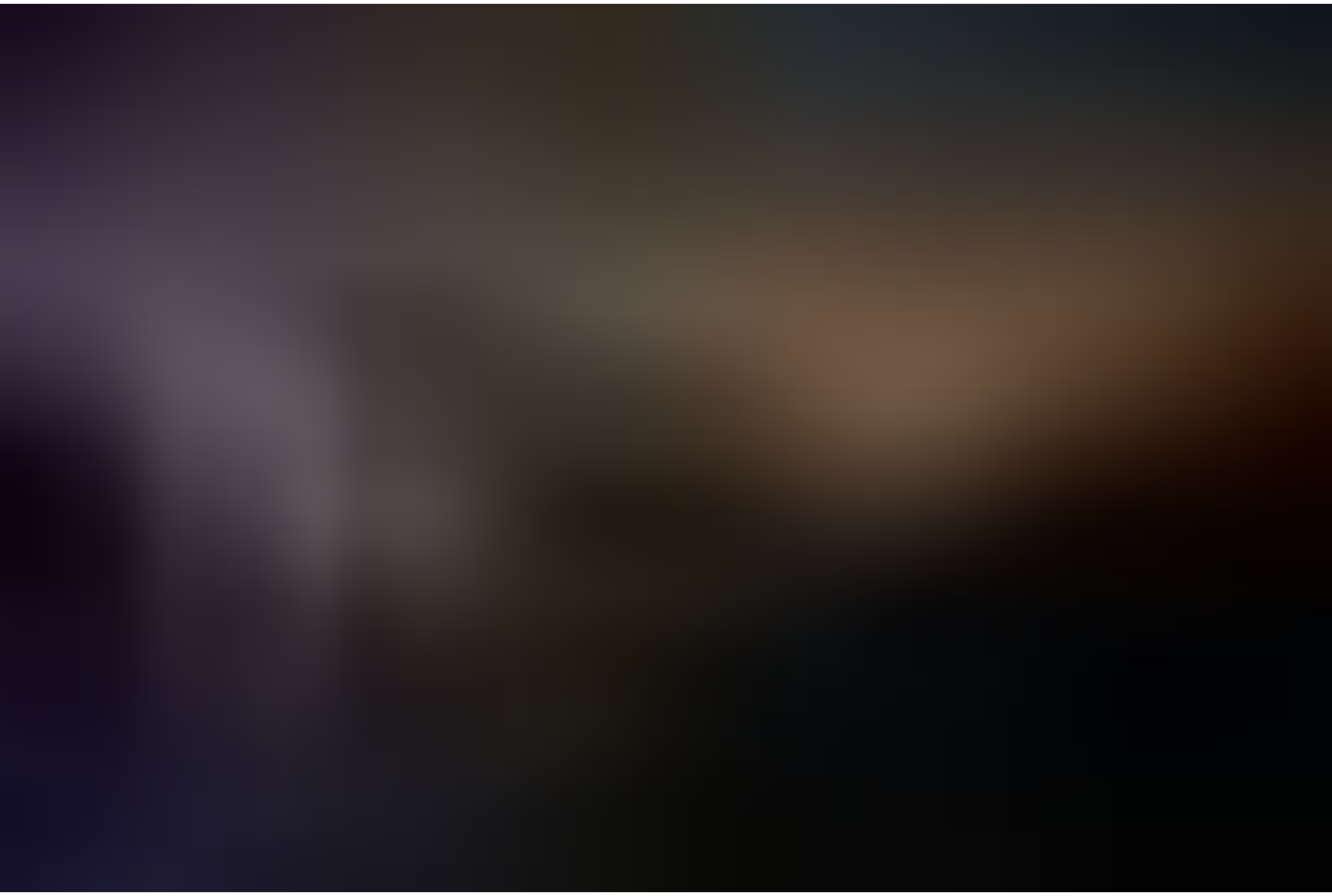

Image from Pexels.com

This is the 4th part of the series of understanding ***SOLID*** *Principles where we explore what is* ***Interface Segregation Principle*** *and why it helps with creating thin abstraction interfaces that make it easy for clients to have fewer dependant factors between them.*

As a small reminder, in **SOLID** there are five basic principles which help to create good (or solid) software architecture. SOLID is an acronym where:-

- **S** stands for **SRP** (Single responsibility principle)
- **O** stands for **OCP** (Open closed principle)
- **L** stands for **LSP** (Liskov substitution principle)
- **I** stand for **ISP** (Interface segregation principle)
- **D** stands for **DIP** (Dependency inversion principle)

We've discussed Dependency Inversion, Single Responsibility and Open-Closed Principle before.

Now we are going to address Interface Segregation.

Interface Segregation Principle

> *Clients should not be forced to depend on methods that they do not use.*

This principle is very much related to the Single Responsibility Principle. What it really means is that you should always design your abstractions in a way that the clients that are using the exposed methods do not get the whole pie instead. **That also include imposing the clients with the burden of implementing methods that they don't actually need.**

Keep your interfaces thin or fine-grained and don't attach to them un-used methods. You will have to be more resourceful with the naming as you will have to name a few more types than you do already. That's fine because it gives you the flexibility to compose thinner interfaces into bigger ones if you need to.

This example is about an interface for the User Interface for an ATM, that handles all requests such as a deposit request, or a withdrawal request, and how this interface needs to be segregated into individual and more specific interfaces.

Let's see an example in Go. Suppose we have a library of utility functions that deal with byte operations that we want to expose. We create the interface:

```
type ByteUtils interface {
    Read(b []byte) (n int, err error) // Read into buffer
    Write(b []byte)(n int, err error) // Write into buffer
    Trim(b []byte, exclusions string)[]byte // Trim buffer
by removing bytes from the exclusion chars
}
```

It works for a while. However notice that the name *ByteUtils* is too generic, it doesn't give you any specific information—it could be called anything. Also, it's very awkward to implement the required methods because if you have a different optimisation to do on the trimming side when you read into a buffer as opposed to when you write to a buffer you will need to provide 2 different implementations of ByteUtils that both read and write that are the same. Not Good!

Instead, we could provide a simple interface list:

```
type Reader interface {
    Read(b []byte) (n int, err error)
}

type Writer interface {
    Write(b []byte)(n int, err error)
}

type Trimmer interface {
    Trim(b []byte, exclusions string)[]byte
}
```

Such thin interfaces are also called ***role interfaces*** as its easier to refactor, change, and redeploy something that it has a very defined role or purpose.

In that case, we could combine a more relevant interface list like that:

```
type ReadWriter interface {
    Reader
    Writer
}
```

```
type TrimReader interface {
    Trimmer
    Reader
}
```

This gives the flexibility for the clients to combine the abstractions as they may see fit and to provide implementations without unnecessary cargo. In the above example a client may provide an implementation for an Optimised TrimReader.

Conclusion

As you can see **fat interfaces lead to inadvertent coupling between classes** and you should avoid them. When designing interfaces you should always ask your self the question "**Do really *need all the methods on this interface I'm using? If not how can I break them into smaller interfaces?***".

As Gandhi said once:

> *"Your actions become your habits,*
> *Your habits become your values,*
> *Your values become your destiny."*

If you do the right thing every and allow only the good patterns to emerge from your architecture, it will become a habit. And that it will become your values or principles. If you keep doing that it will become your destiny, which is to always produce sound solutions or whatever your idea of a software engineer is.

My point is to always strive to be better by challenging yourself. At some point, you will ask, but you will know already the answer…

Happy coding!

References

- Wiki page
- C2 Page

Coming up next is Understanding SOLID Principles: Liskov Substitution Principle

If this post was helpful please clap it and stay tuned for my other articles. You can follow me on GitHub and LinkedIn. If you have any ideas and improvements feel free to share them with me.

If you would like to schedule a mentoring session visit my Codementor Profile.

Happy coding.