Revisiting SOLID

One of the cornerstones of great software development, do you remember what it stands for?

Matthew Lucas Follow
Dec 14, 2019 · 8 min read ★

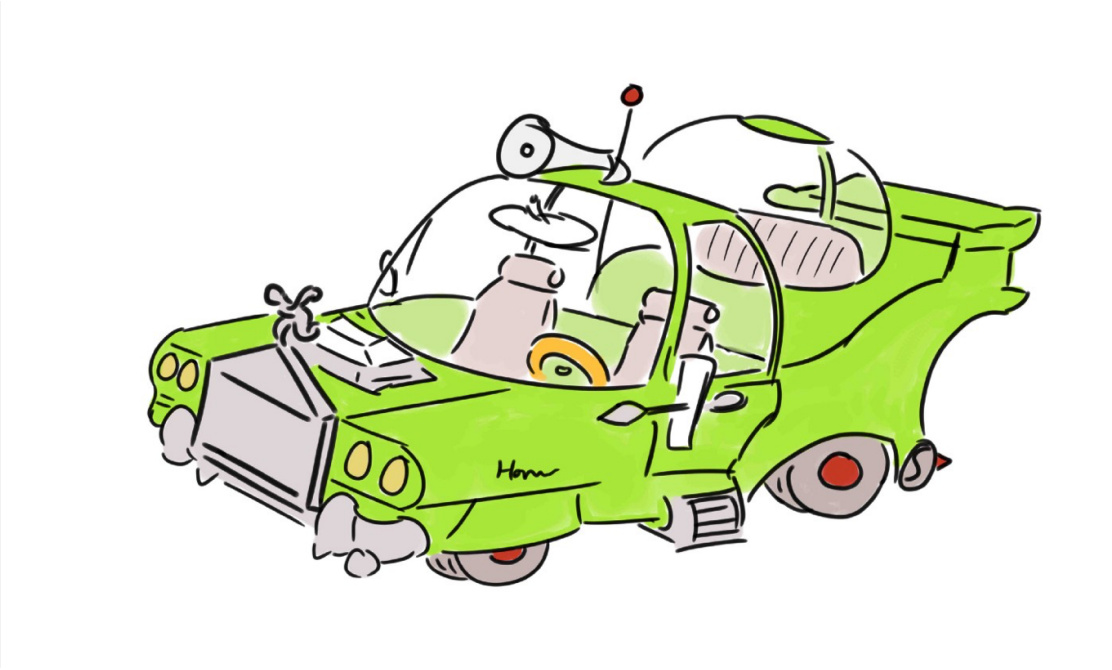
Encountered by most software developers in their early years on the job, the SOLID principles remain a key driver in all well-written software.

I hear it referenced often and frequently find myself using it in conversation, but realized — to my shame — that I couldn't recall what each piece represented.

Single responsibility principle, OK, open-closed principle, sure, and then …? Maybe it's time to revisit SOLID.

. . .

Single Responsibility Principle



> “Gather together the things that change for the same reasons. Separate those things that change for different reasons.”

It’s a common misconception to simplify this as *“every module should do just one thing”*. That’s not quite right because it’s not the “doing” that’s the point.

Boiling your code down to the simplest possible components can lead to horribly disjointed software. It’s the reason to change that’s key rather than what it’s doing, but what defines a reason to change?

The principle is primarily about people. Those users and stakeholders that push for new features are those that drive change in your software, and it’s specifically the scope of that change that we’re looking to limit.

It’s frustrating for both users and developers when a reasonable tweak in one concern leaks into, and breaks, another. You’ll want to separate the responsibilities to avoid exactly this and to reduce the mental burden required by a developer when dealing with just one piece.

The single responsibility principle applies across the whole stack, from minute detail to high-level abstraction. It holds when building granular functions or coarse components and in determining architectural boundaries.

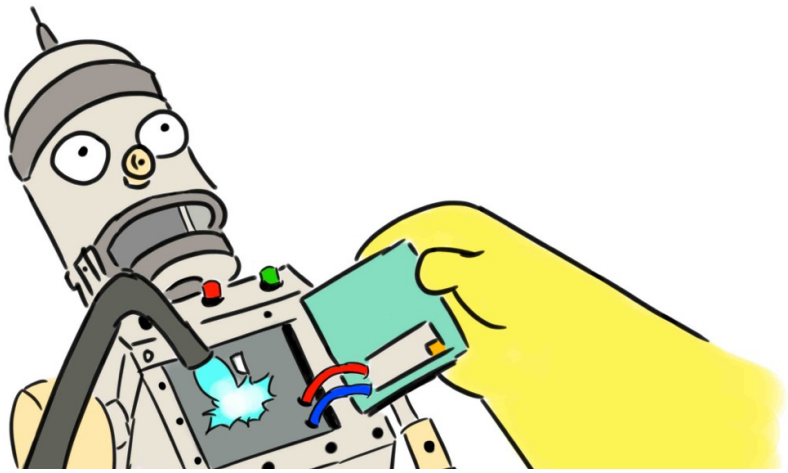
It may be seen under a different guise, for example, the common closure principle which itself advises that components should be deployed according to change and that any change should effect a minimal set of components (ideally one).

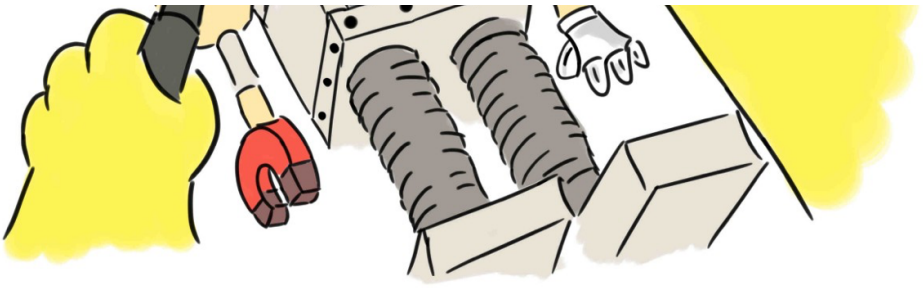
Much of the SRP can be seen as an answer to a development scalability problem.

Separate those areas that can change for different reasons and you can scale out your engineers without them treading on each other’s toes or scale-up teams without blocking dependencies and bottlenecks between them.

. . .

Open/Closed Principle





> “A software artifact should be open for extension but closed for modification.”

Extension, so this must be about inheritance? Sure enough, in Bertrand Meyer’s original incarnation, that was the case but hold on there!

Over the past 30 years, we’ve learned that inheritance is quite a tricky beast to tame, so we might not want to take that route right away.

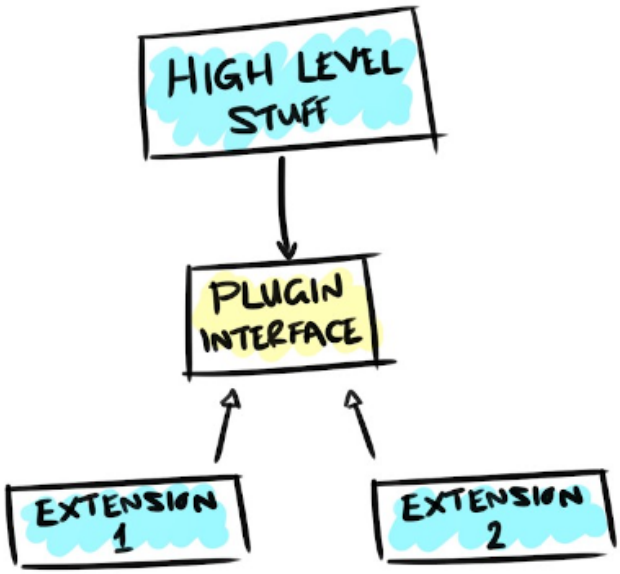
What do we mean by preventing modification and allowing for extension? If you have to change reams of existing code every time you make a small alteration then you know there’s something wrong with your architecture — it isn’t closed for modification.

Rather than changing what’s already there, you should be able to build on top of a robust foundation through extension. Adding new features should ideally be easy and low risk.

Extension itself has more than one form. Firstly, you have the traditional mode of inheritance. This, as mentioned, comes with its own set of problems (discussed elsewhere).

Alternatively, you can inject additional behavior into high-level components. Interfaces are the perfect tool for extension through injection. They protect against modification by presenting a well-defined contract that controls change.

They then allow us to extend our software by providing and plugging in new implementations of this contract.



The use of interfaces here ensures a robust hierarchy where those important high-level components are protected from changes in lower-level ones. When that next feature request comes along you'll be ready to slot it in without touching those precious existing functions.

. . .

Liskov Substitution Principle

> “If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T”

Ouch! That’s a bit of a mouthful. Let’s forget the computer science-y language for a moment and focus on the core point. The Liskov substitution principle is all about doing inheritance well.

It tells us not to create subtypes that, despite conforming correctly at a contract level, diverge wildly from the true semantics creating some nasty surprises in the behavior of our program in the process.

The usual example used (and I’ll do no different through sheer laziness) is that of the square/rectangle problem. To begin, let’s say we have a type, `Rectangle`, with two methods `#setX` and `#setY` to assign the lengths of the two dimensions appropriately — so far so good.

Now, what if we create a subclass for the special case of a `Square` in which *X* and *Y* must always be equal?

If we have a program that uses `Rectangle`, and then all of a sudden we sneak in the `Square` subtype, things could get a bit strange when we call `#setX` and it changes *Y* too (or vice versa). The subtype has to make sense in the shadow of its parent.

This isn’t limited to just inheritance but also makes sense in ensuring the robustness of any interface, be it an API, REST call, or something else.

Substitution of any of these for another should be seamless, going unnoticed by any caller. If the client has to hack in a workaround to handle different implementations then we’ve botched the point of having a contract.

Respecting this principle, however, can often be quite difficult. Many interfaces aren't quite as abstract as they really should be, leaking some implementation detail.

This is especially true when it comes to the things you may not always really consider, like exceptions.

To avoid setting this trap yourself, try and keep your interface declarations as simple as you possibly can — other developers will thank you.

. . .

Interface Segregation Principle

> "No client should be forced to depend on methods it does not use"

Conceptually the simplest of the principles and one that is closely related to the SRP, the interface segregation principle is about finding the most

appropriate abstractions in your code. The reason for this, is again, to protect against one avenue of change affecting other orthogonal ones.

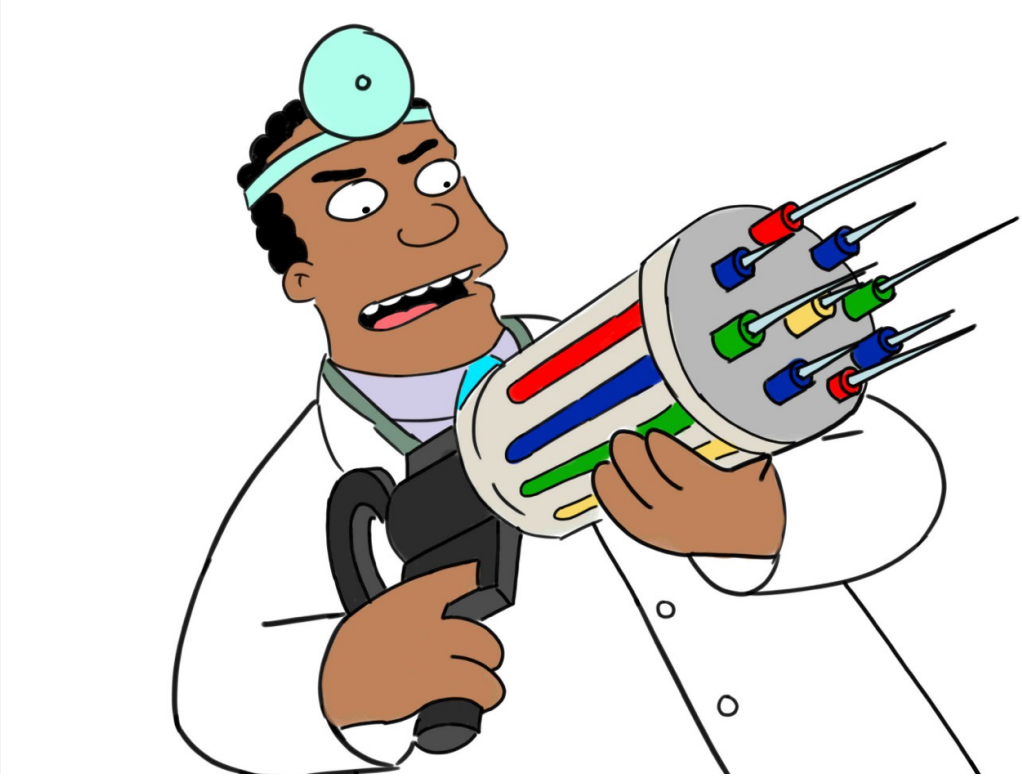
Having to depend on an interface that has muddled concerns can lead to a lot of unwanted baggage. The bloat can leak into your code as you have to cater for cases you don't care about. Upgrades come thick and fast but for features that you don't use and with the risk that those changes will break those you do.

Overly complex interfaces are usually grown rather than designed. The slow drip feed of additional requirements, odd tweaks, and little hacks can snowball over the years to create a monster.

Strong interface and API design is an exercise in restraint. Cohesion is your guide here. Keep your interfaces cohesive, collecting together those concerns that change together, and free your clients from an extra burden.

. . .

Dependency Inversion Principle



> “The most flexible systems are those in which dependencies refer only to abstractions and not concretions”

An application is almost always comprised of some core business logic triggering real-world outcomes through a set of utilities. A report generator exporting to a spreadsheet or railway traffic system controlling stop-lights being two such examples.

The flow of control of a program is often from core logic to more specialized utilities, such as I/O.

If the source code dependencies flow in this same direction, however, it can result in tight coupling of business logic with lower-level concerns. As low-level implementations are likely to change often, this will be a source of volatility within an application.

To protect against volatility, we want any high-level concerns to depend on abstractions rather than concretions — that is, interfaces rather than implementations. Interfaces change much less frequently than their instantiations, and so referencing these will shield a client from uncertainty.

Of course, something has to know about the dirty detail of implementation, and that’s where dependency injection helps.

Rather than core logic knowing what to use directly, you inject it with “an implementation” ensuring the source is as ignorant as possible to those low-level concerns. The grubby wiring can be tucked away in a few key places, such as factories or dependency injection frameworks, e.g. Spring.

To conclude this principle, it’s worth reiterating a few points listed by Uncle

Bob concerning the DIP:

- Don’t refer to volatile concrete classes.
- Don’t derive (inherit) from volatile concrete classes.
- Don’t override concrete functions.
- Never mention the name of anything concrete and volatile.

. . .

Conclusion

A key realization I had from revisiting this topic is that the SOLID principles aren’t just low-level coding tools, but guidelines that ascend all levels of software design, development, and architecture.

Hopefully, this review has helped express the universality of these principles, and will help you recall them next time you’re deep in design — I’m sure, this time, that I won’t forget them!

. . .

References

- Clean Architecture: A Craftsman's Guide to Software Structure and Design by Robert C Martin
- https://hackernoon.com/you-dont-understand-the-single-responsibility-principle-abfdd005b137
- https://medium.com/@severinperez/writing-flexible-code-with-the-single-responsibility-principle-b71c4f3f883f
- https://stackify.com/solid-design-open-closed-principle/
- https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle
- https://hackernoon.com/interface-segregation-principle-bdf3f94f1d11
- https://deviq.com/dependency-inversion-principle/
- https://stackify.com/dependency-inversion-principle/
- All images of 'The Simpsons' are owned by 20th Century Fox, re-drawn and used through the fair-use policy for non-commercial and illustrative (educational) purposes.

Software Development | Programming | Software Engineering | Technology | Coding