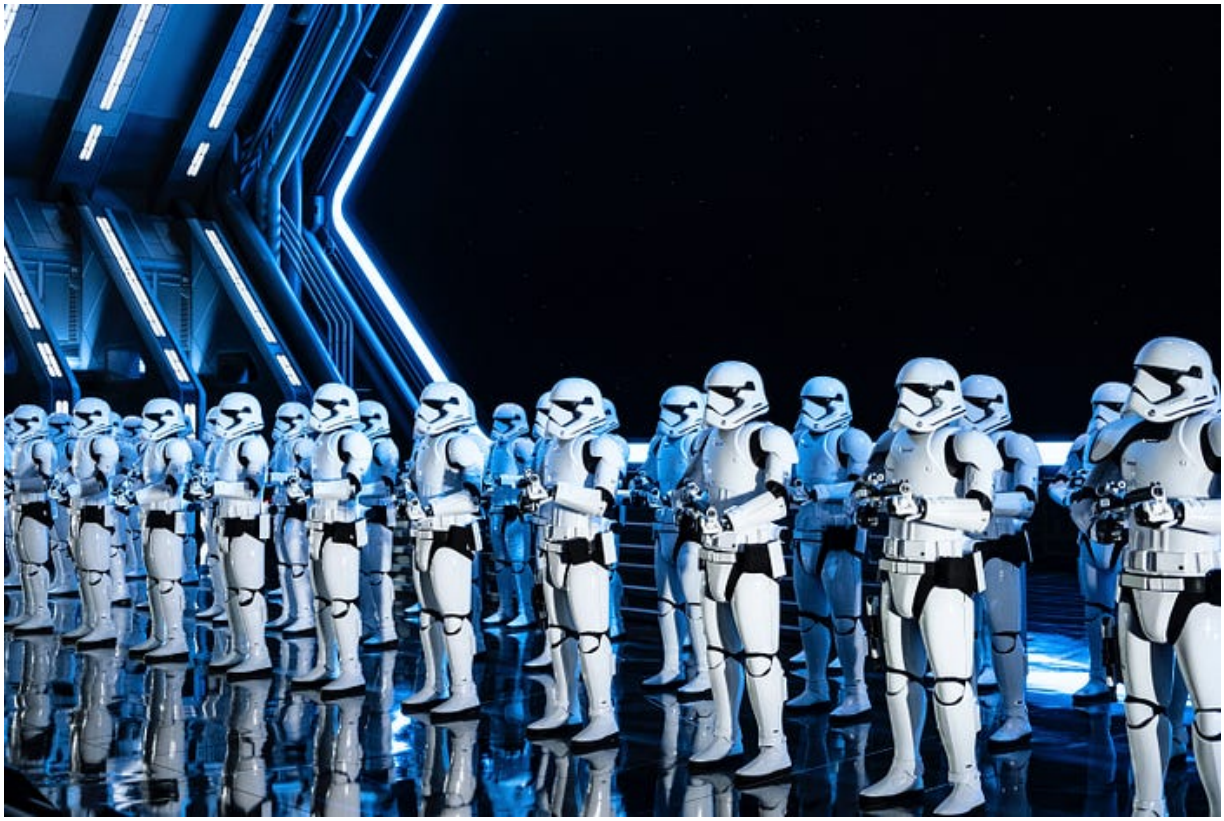

The Evil in Code Generation in Swift

Disadvantages of Automation and Generated Code

Sven Korset

Follow

Better Programming · ~6 min read ·
February 19, 2020 (Updated: December 13, 2021) · Free: No

PIECES OF A SCALABLE IOS APP ARCHITECTURE

As a clever developer, you already know the new-fashioned creed:

> Automate everything!

eally? Everything? 😉

manual way over the lifespan of the system.

What Does That Mean?

Every automatism needs an initial effort to put it on. For example, you first have to read the tool's API documentation and then build a script based on it.

Also every automation system usually has a need for maintenance because something always changes over time. The next API-breaking major update is already lurking.

At some point, the maintenance effort may be so high that it is simply no longer worth it (if only we had made it cleaner). Or the knowledge may have been lost over time (if only we hadn't fired the admin). Or the tool is simply no longer needed (if only we had seen into the future). 😄

Everything is finite, so is an automatism. You have to add up all the time you have invested in the automation and ask yourself: "Did I really save time? Or was it at least worth it?"

The smart developer estimates this in advance. And the experienced developer is even halfway right with the estimate. 😜

Automation Tools

If you develop iOS apps, sooner or later, you'll come into contact with automation tools like the following:

Fastlane

automatically. Fastlane is a tool suite that lets you automatically build the code, run tests, upload beta versions, create screenshots, and more. Not only for iOS but also for Android.

Sounds cool, and it is. At least until the pipeline is broken again because something is stuck on the CI server. Now you have to spend another day, two, or three fiddling around. At some point this mutates into a full-time job. 😉

However, the effort is often worthwhile, because if Fastlane is set up correctly on the CI server, it helps to keep the quality of the project high in larger teams. It can test things that you might not do manually, either because you forget it or because it is simply too cumbersome or tedious.

But let's talk about source code generators. 🧐

SwiftGen

With SwiftGen, you can have Swift code generated automatically. For example, the tool goes through a resource folder and creates type-safe constants for strings, colors, fonts, etc. You can even use your own stencil templates to individualize the code. 🤓

The limitation lies in the generator or the parser. What is not accepted as input or not processed appropriately for the template can of course not come out as a transformed output. The tool is open-source. Therefore, in an emergency, it could also be extended. But who really wants to maintain SwiftGen? 😖

R.swift

downside, however, is that it doesn't use templates that you can customize. This means either you take the tool as it is because it fits or it just does not fit and you take another tool like SwiftGen. 😕

Personally, I prefer R.swift over SwiftGen because it's easier and the result is good enough. Configuring SwiftGen so that it gives me more than R.swift out of the box would be disproportionately more effort, especially in maintenance. Therefore, I will stay with R.swift for now. ☺

Sourcery

Sourcery can also be used to generate Swift code, but it is used more to generate repetitive boilerplate code. So, you script a template and then automatically create Swift code from it according to a certain scheme.

The effort, therefore, consists of creating and maintaining the templates. But if you find suitable use cases, you can certainly save time with Sourcery. Of course, only after having worked through tutorials and docs. 😅

Weaver

If you want to use dependency injection but want to resolve the injected dependency at compile time rather than runtime, then you need a corresponding code generator such as Weaver.

However, it is questionable whether this tool can really save you time and if it fits well enough into your project. I personally prefer the manual way, as described in my previous piece "Dependency Management Done Manually in Swift". 😇

When working against a REST API, chances are good that the back-end developer will also provide documentation in the OpenAPI/Swagger format. With Swagger Codegen, you can then automatically convert this standardized specification format into Swift code.

It's a great joy when the models and requests are finally automatically converted into code — at least until the realization comes that the result may not suit you as you would like and there is no way to adjust any template. 😞

Problems With Automatically Generated Code

Code is generated automatically. It saves time. What's so bad about that? 🤨

Training time

Every new tool means a certain training period. And familiarization is rarely just a five-minute readme. You experiment for days until you understand it and it fits into your project.

In the worst case, after a year you have to run the tool again or reconfigure it but have forgotten everything. Of course, then you don't have the time to familiarize yourself with it again. 😅

Depending on the tool

What if the tool does not exist after a year? Or maybe it still exists but it is no longer maintained. As long as you can still work with the old version from your own Git repo, everything is fine, but this is a

Then you have to look for alternatives and notice that each tool creates code somewhat differently. One tool creates one class, the next two classes, then the third works with protocols and factories, and the fourth tool names everything completely differently. You guessed well: You now have to adapt the interface. 😕

Maintenance of generator resources

Everything the generator needs as input may need to be maintained. These can be configuration files and templates but also the source code itself. New parameters or variables during a tool update might make it necessary, for example, to extend your own template. Otherwise, the project can no longer be built. A simple `pod update` may change into a day-long task. 😩

Result is not always optimal

You are rarely 100% satisfied with the generated code. It's good if you can tweak the template, but that too has limits. Usually, however, the code is not generated as if you had written it cleanly manually. At least the interface would have looked different. This leads to compromises. However, that means that the time savings come at the price of code quality. 😟

Conclusion

You could have a different impression now, but automating is essentially a good thing. Automation means having to do less boring work and thus having more time for the interesting work. 😄

quality. And sometimes, you can increase the quality but also have to invest much more time.

It's often a trade-off. You just have to be aware of it to be able to weigh the costs and benefits. 😜

Incidentally, this is an article from the "Pieces of a scalable iOS app architecture" series.

#automation #ios #programming #swift #mobile