Software Engineering CS3012 – Report on influential Software Engineer/Engineering project

The go programming language

Why Golang

When trying to think up of a software engineer who influenced me I quickly realised that no such person fit the bill for me. What interested me the most about software engineering wasn't the people or the type of projects they made per say but rather the nature of working on software – akin to puzzle solving. Hence why I chose a software engineering project rather than person since I was more interested in the domain of the problem rather than the people who solved it.

The second reason why is because I felt that software is more than just one person decreeing structure and design and writing all the code themselves. Many of the trade-offs and choices are indeed caused by having to flexibly try to cater to the many stakeholders involved in the creation of software. Creating software is a collaborative effort and while software may take on characteristics of the key designers I think as software evolves to be larger and more complex it made more sense to focus on the project itself rather than any individual involved in its creation.

As for why Golang (the Go programming language), well there are many reasons but primary among them is that unlike many organic languages it felt like Golang was a very top down engineered language. The people working on Golang are pretty smart software engineers who designed Golang to try and solve problems they faced. Golang is a relatively modern/recent language with decent popularity so far. This meant it was a good match since it likely reflected the current trade-offs faced when making software. The uptake means that it likely solved some problem in the current language space. It's considered that tools in many ways shape the solutions created so, as a computer scientist I felt it was interesting to research a language and the engineering decisions made and more importantly why they are made.

There will be many links sprinkled throughout, not because they are necessary to understand the info presented here but because it's just nice to have easy links in case you want greater depth/closer top primary source info.

Golang Background

Go was designed by the Google engineers Robert Griesemer, Rob Pike, and Ken Thompson. Slight appeal to authority there, hence why I left in the hyperlinks and Google bit. I'm just giving their credentials, so that it shows the experience of the engineers. The nice thing about Go is

that there's a lot of documentation on why go as a language exists and if you want a more in depth explanation etc. you can get it almost straight from the horse's mouth here and here.

Go was mainly designed to be a replacement for C++. It was meant to be a more simplified C++, designed to be easier to use as well as developed for modern tools while getting rid of much of the baggage C++ brings. What should be obvious even at this stage is that there are many trade-offs as a result. Mature, commonly used languages like C++ don't carry this baggage just because of poor engineering. The baggage is often caused by prioritizing different factors

Go was designed to be a programming language that's scalable for large projects. Its key design goals are to be easy to maintain, debug and change at the scale of large software systems. Some of the key features they tried to tackle with go are build speed, easy to read code, easy to change code, modernity and streamlined language functionality. The inbuilt decision to favor low latency and concurrency also comes from the types of software domains that go was specialized for, web servers. Here's a nice video to why in-built concurrency.

Golang's Engineering

From here I guess I'll talk a bit more about the in-depth engineering trade-offs that go makes and more importantly why. First however let's give some background to the three key software engineers.

Rob Pike is best known for his work on Go (programming language) and at Bell Labs, where he was a member of the Unix team and was involved in the creation of the Plan 9 from Bell Labs and Inferno operating systems, as well as the Limbo programming language. Source

Ken Thompson is an American pioneer of computer science. Having worked at Bell Labs for most of his career, Thompson designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating systems. Since 2006, Thompson has worked at Google, where he co-invented the Go programming language. Other notable contributions included his work on regular expressions and early computer text editors QED and ed, the definition of the UTF-8 encoding, his work on computer chess that included creation of endgame tablebases and the chess machine Belle. Source

Robert Griesemer is one of the initial designers of the Go programming language. Prior to Go, Robert worked on code generation for Google's V8 JavaScript engine, the design and implementation of the domain-specific language Sawzall, the Java HotSpot virtual machine, and the Strongtalk system. He once wrote a vectorizing compiler for the Cray Y-MP and an interpreter for APL. He is a fan of things that "just work." Robert holds a Ph.D. in computer science from ETH Zurich, Switzerland. Source

Smart people and lots of experience between them. A lot of the info on Go design decisions seem to actually come from Rob Pike so it might be biased in that regard. I don't particularly

want to talk about any of the specific engineers themselves as I said above and Go itself seems to have been designed with a principle of “Design by consensus” after all. This is meant to be a document on the software engineering side rather than the technical side of Go so I’ll try take as high level an approach as possible so we can simply talk about trade-offs and choices rather than low level implementation.

Code simplicity

One of the key features which Go brags about is the ease of becoming productive in the language. Go tries to maintain a policy of simplicity at the expense of providing expressiveness. Unlike C++ which provides a large amount of functionality, Go is designed to provide a significantly reduced feature set while still maintaining the same power. This link on “less is more” talks about this idea underpinning much of Go’s design. The idea is that you have to keep syntactic sugar in your head as ways to do things are limited. Go attempts to limit expressiveness so that the number of concepts you have to keep in your head while programming are reduced to basically just the problem itself. Readability for others is also improved as mastering the syntax of the language is easier.

The drawback of this approach is quite obvious. By sacrificing expressiveness you sacrifice some degree of power and abstraction of writing code. By cutting down on syntactic sugar you reduce the convenience of writing code in favour of explicitness. Furthermore, go’s decision to not add generics (for users) means that developers have to write boilerplate and increases code reuse.

Garbage Collection

Garbage collection is one of the largest trade-offs that Go makes. Go sacrifices the ability to have full control over memory in return for ease of management. The abstraction of garbage collection trades off performance for convenience and safety. This is one of the most important trade-offs to consider. Languages such as C++ and C offer full control over memory, which is one of their principle benefits. By having full control of memory, garbage collection latency and memory management becomes predictable. Memory management in these languages are explicit which provides a crucial benefit in domains where factors like memory usage and latency are critical.

Go on the other hand abstracts away memory management in favour of a low latency garbage collector. This trade-off doesn’t come cheap however. Due to prioritising latency the Go gc trades away other desirable gc properties in languages such as Java and Haskell. This is one of the big software engineering decisions that Go makes, the idea that in return for losing explicit control of memory Go will provide a gc with negligible latency and scalable performance as memory becomes cheaper over time.

Currently however as a result many of the people who program in Go didn't come from lower level languages like C++ and C but instead from languages such as Python and PHP. It seems reasonable to suggest therefore that currently Go's choice of automatic memory management makes it unsuitable for low level programming where explicit control of memory is still necessary.

Concurrency

One of Go's most flaunted features is built in concurrency, which makes creating concurrent applications easy. However going back to the decision at the start of making Go easy to write for, Go is not purely memory safe in the presence of concurrency. Taken straight from the link https://talks.golang.org/2012/splash.article

"Some concurrency and functional programming experts are disappointed that Go does not take a write-once approach to value semantics in the context of concurrent computation, that Go is not more like Erlang for example. Again, the reason is largely about familiarity and suitability for the problem domain. Go's concurrent features work well in a context familiar to most programmers. Go *enables* simple, safe concurrent programming but does not *forbid* bad programming."

The tradeoff is quite obvious of safety vs ease of use. Race conditions can easily result from Go's system of managing concurrency.

Conclusion

Since this report was only meant to be 4 page long and also because I preferred an approach of studying a small number of important trade-offs in detail rather than a large number of trade-offs in low detail, I think it's about time to wrap it up. The obvious lesson is that making software always comes with a large number of costs and caveats. I didn't cover all the trade-offs that Go makes over other languages since it is an entire programming language with its own way of doing things and this is only a 4 page report.

Before I wrap up however I would like to make a few more, sort of tangential points about Go. One is that there is commercial software that is written in Go such as Docker for example, here's a link about people using Go. However Go seems to not have been having much success displacing much of the very low, systems level C or C++ code but is instead used on a layer slightly higher up (even if the distinction is a bit hazy). I found it just as interesting to find the flaws rather than just a success story because it presents a more nuanced view on what software engineering is. Finally, as I'm running out of space here's an interesting link on Go's drawbacks