Editor: **Robert Blumen**
Symphony Commerce
robert@robertblumen.com

The Go Programming Language

Jeff Meyerson

THIS IS THE FIRST of what will become a regular column highlighting recent episodes of the Software Engineering (SE) Radio (www.se-radio.net) podcast. For readers of this magazine who aren't listeners, I'll provide a brief history. The show was founded in 2006 by programmer and computer science researcher Markus Völter. His intention was to fill an empty niche in the podcast ecosystem: a deeply technical show for programmers about software engineering topics. The aim of the show was, and is, to provide relevant and in-depth insights from experts for practitioners; it has been in continuous publication ever since, and has accumulated an archive of more than 200 podcasts. Two years ago, Markus stepped down and passed ownership and management over to *IEEE Software*. The show's audience has now steadily grown to the point where each episode is downloaded more than 50,000 times, sometimes one or even two years after the publication date.

The podcasts consist of one-hour interviews with a host and usually one but sometimes two guests. While the show has covered a huge range of topics under the general umbrella of software engineering, its historical strengths have been programming languages, design patterns, systems, and development processes. SE Radio has featured several thought leaders in modern software engineering, including Amazon CTO Verner Vogels, XP founder Kent Beck, design patterns guru Martin Fowler, and Martin Oderskey, inventor of the Scala programming language.

After listening to nearly all of the archived contents over a period of months, I volunteered to become a show host four years ago; I assumed the role of show editor this April. My goal in this new role is to maintain our historical strengths in systems, computer science, languages, and processes while we stretch into new areas such as career development, hiring, and software's cultural and organizational aspects. Some hosts have stayed on through the transition, and new show hosts have volunteered. This expansion will enable us to increase publication frequency while maintaining the high quality that the show's founders established. By the time that you read this, several of our new hosts will have published their first podcast.

This first installment in *IEEE Software* features excerpts from podcast 202, originally published on 14 March 2014. Host Jeff Meyerson spoke with Google engineer Andrew Gerrand about the Go programming language; I've included their discussion of the history of the language, starting with the frustration among Google programmers at using tools that didn't scale up. You can download the complete show from the archives at se-radio.net. *—Robert Blumen*

Continued on p. 101 …

0740-7459/14/$31.00 © 2014 IEEE

... Continued from p. 104

Could you start us off with some brief history of Go?
Go was actually started at Google, nearly six years ago, maybe even a little earlier, as a side project by Rob Pike, Robert Griesemer, and Ken Thompson. They hacked on it for a while, until it gradually became their full-time thing. Then we added a few more people to the team over the next couple of years and released it as an open source project in November 2009.

About two years ago, we released Go 1.0. That was our sort of big table release. Since then, we've done a couple of other major releases. We've seen a lot of growth, a huge amount of engagement from the open source community.

We have nearly 400 contributors to the project and a very active non-Google development team as well. A bunch of people still work full time on Go at Google, and a lot of people outside contribute to the project, too.

What were some of the objectives of the language's founders when the project initially got started?
There's a joke that Go was conceived while waiting for a C++ program to compile, which is kind of half true. Basically, Go grew from a dissatisfaction with the development environments and languages that we were using at Google. None of them had really been designed well; C++ and Java were more than a decade old by that stage. None of these environments were designed to work at the scale on which we were working at Google.

Everyone knows and thinks about Google in terms of scale of users and scale of servers, but one thing that's not talked about as often is the scale of engineering effort. We have many, many thousands of programmers working on huge code bases. It's just one big shared code base. No language was designed to work at that kind of scale. We had a lot of awful workarounds that we've had to employ to simply function at this kind of organizational scale.

There's a joke that Go was conceived while waiting for a C++ program to compile, which is kind of half true.

Go was an opportunity to think about a lot of the problems that we've encountered working at this scale and to design a language that really works. One good example of this is Gofmt, which stands for Go Format. It's a tool that takes any Go Source file and freely prints it in the conical formatting style, which means that your Go code looks the same as anybody else's Go code. You don't have to talk about brace placement or indentation or any of those other nitpicky things in code reviews. It's all taken care of for you.

I imagine these things help when starting to build editors and debugging tools for working with Go.
The language is designed to be easy to parse and, by extension, easy to mechanically manipulate. We have a growing suite of tools for pausing, processing, and computing on Go source code. One tool just released by Go team members is called Go Imports. You just feed a Go program through it, and it automatically updates the package import statements in the file to add or remove the relevant imports depending on what other packages you're using in the code. It figures out which other libraries you want it to use and then adds those import statements to the file. It takes a lot of the friction out of working with other people's code and just coding in general.

Can you describe how interfaces work in Go?
Go has an interesting object model. It's an object-oriented language, but if you come from, say, a Java or a C++ background, you might find the approach a little different because, for instance, Go doesn't have classes. It doesn't have a notion of inheritance in the same way. The lynchpin of Go's object model is interfaces.

The way interfaces work in Go is that you specify an interface type, which contains some method specifications. A classic one is the I/O reader interface from the Standard Library, which defines a read method. Anything that provides a stream of bytes will typically implement this read method, but when they implement it, they don't need to specify that they implemented it. Simply the fact of implementing that method of that specification means that it implements the I/O reader.

There's also no implement declaration, which has the nice effect of decoupling pieces of code. You don't have this rigid system of declaring an interface in one place and then wherever you implement it, declaring it. Instead, you can be a bit looser about it: you can declare interfaces in a sort of post hoc fashion, like when you discover that there are similarities between certain types and you want to use them in the same way.

It's a difficult worldview to assimilate just by thinking about it. You need to get your hands dirty and use it a bit. The end result is that Go objects tend to have fewer responsibilities and tend to be a lot smaller than objects in other languages. We build systems not by building type hierarchies, but instead by building small pieces that satisfy small interfaces; then we compose them together. It's a subtle but ultimately profound shift in focus to the way a lot of people have been writing software for a long time.

For me personally, it's been hugely beneficial. I've found that the code that I write in Go is a lot simpler and more reliable than the code I was writing in other languages.

The code that I write in Go is a lot simpler and more reliable than the code I was writing in other languages.

Were there initial frustrations when you were trying to design your code around such a different interface system?

It certainly helps that I had the language designers sitting next to me, but by the time I'd gotten to Go, I'd become dissatisfied with the kind of hoops that a lot of these environments were making me jump through. I was using Python and JavaScript pretty heavily, so everything I was doing was fairly unconventional by that stage. It wasn't a stretch to move into Go and adopt a different approach.

The main thing that I think people have trouble with when coming into Go is in letting go of previous kinds of techniques that they had used to structure things. In Go, you're left more to just write the code that works and not worry so much about the structure of it, at least initially. The structure emerges over time, and you're able to codify things a bit more, define some interfaces. It becomes a more formal way to write instead of starting off with this formal design.

How does Go manage concurrency?

Concurrency is one of the big features of Go as a language. The concurrency model is based on two fundamental elements. One of them is Goroutines, which are lightweight threads. Typically, you'll have tens, hundreds, or even thousands of Goroutines multiplexed across fewer operating system threads: the idea is that you would have as many operating system threads as there are CPUs in your machine.

As you know, threads are quite expensive, so you want to only have as many as you need for that parallelism. The Go code actually runs in these Goroutines, which are quite small. You can generally create and destroy Goroutines as often as you like without really thinking about it, which lets you write concurrent code in a top-down, straightforward style, with the behavior of an event-driven framework. Your code can block, but the runtime immediately moves any blocking code away from being executed, and then it executes some code that *can* run. You get that nice, high-performance concurrency that you get from, say, Node.js, but you can run across multiple threads. You don't need to get into callbacks upon callbacks. This leads to a comprehensible programming model but with efficient runtime characteristics.

To communicate between Goroutines, we have another set of primitive code channels, which are basically just like a typed pipe for sending Go values. If a Goroutine does a send on a channel, and another Goroutine does a receive from that channel, then in a fairly safe way, they can pass data back and forth, which means they can also synchronize because the send and receive must happen simultaneously.

Channels are derived from "Communicating Sequential Processes" (C.A.R. Hoare, *Comm. ACM*, vol. 21, no. 8, 1978, pp. 666–677), a seminal paper on concurrent programming from the 1970s. If you haven't used them before, it's an eye-opening experience to write a concurrent code with channels because it really lets you model concurrent processes, which are typically real-world processes but in a way that's very straightforward and easy to understand.

How does the garbage collection system in Go compare to that of other languages?
Go is a garbage collection language, just like a lot of scripting languages such as Java. Go's garbage collector [GC] is pretty simple. It behaves predictably. It's a stop-the-world garbage collector, so no user code runs while it's collecting garbage, but it's one that tends to perform pretty well.

As a Go programmer, it's very easy to see when garbage is being created; the language gives you the tools you need to reuse memory where appropriate and to avoid allocations. So, even though Go's garbage collector isn't as advanced as that of some other environments, you also don't need to put as much pressure on it in Go. You simply avoid creating a lot of garbage in the first place.

We have some nice profiling tools for memory allocations and for seeing where the garbage is coming from. Once you know where it comes from, the language gives you the tools that you need to reuse memory where you can and to avoid putting the pressure on the GC. It's a very different model than, say, in Java, where you tune the GC to get good performance. It's harder in Java to tune the application because a lot of allocations are just impossible to avoid: they're baked into some of Java's core APIs. SW

JEFF MEYERSON is a software engineer at OwnerIQ. Contact him through www.jeffmeyerson.com.

See www.computer.org/software-multimedia for multimedia content related to this article.

Take the CS Library wherever you go!

IEEE Computer Society magazines and Transactions are now available to subscribers in the portable ePub format.

Just download the articles from the IEEE Computer Society Digital Library, and you can read them on any device that supports ePub. For more information, including a list of compatible devices, visit

www.computer.org/epub

IEEE   IEEE computer society

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscription rates: IEEE Computer Society members get the lowest rate of US$59 per year, which includes printed issues plus online access to all issues published since 1984. Go to www.computer.org/subscribe to order and for more information on other subscription prices. Back issues: $20 for members, $216.17 for nonmembers (plus shipping and handling).

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

Reuse Rights and Reprint Permissions: Educational or personal use of this material is permitted without fee, provided such use: 1) is not made for profit; 2) includes this notice and a full citation to the original work on the first page of the copy; and 3) does not imply IEEE endorsement of any third-party products or services. Authors and their companies are permitted to post the accepted version of IEEE-copyrighted material on their own webservers without permission, provided that the IEEE copyright notice and a full citation to the original work appear on the first screen of the posted copy. An accepted manuscript is a version which has been revised by the author to incorporate review suggestions, but not the published version with copyediting, proofreading, and formatting added by IEEE. For more information, please go to: http://www.ieee.org/publications_standards/publications/rights/paperversionpolicy.html. Permission to reprint/republish this material for commercial, advertising, or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org. Copyright © 2014 IEEE. All rights reserved.

Abstracting and Library Use: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy for private use of patrons, provided the per-copy fee indicated in the code at the bottom of the first page is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.