Go at Google: **Language Design** in the **Service** of **Software Engineering**

Rob Pike

Google, Inc.

@rob_pike

http://golang.org/s/plusrob

http://golang.org

Contents

1. Abstract

(This is a modified version of the keynote talk given by Rob Pike at the SPLASH 2012 conference in Tucson, Arizona, on October 25, 2012.)

The Go programming **language** was conceived in late 2007 as an answer to some of the problems we were seeing developing **software** infrastructure at Google. The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather

than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.

Go was designed and developed to make working in this environment more productive. Besides its better-known aspects such as built-in concurrency and garbage collection, Go's **design** considerations include rigorous dependency management, the adaptability of software architecture as systems grow, and robustness across the boundaries between components.

This article explains how these issues were addressed while building an efficient, compiled programming **language** that feels lightweight and pleasant. Examples and explanations will be taken from the real-world problems faced at Google.

2. Introduction

Go is a compiled, concurrent, garbage-collected, statically typed **language** developed at Google. It is an open source project: Google imports the public repository rather than the other way around.

Go is efficient, scalable, and productive. Some programmers find it fun to work in; others find it unimaginative, even boring. In this article we will explain why those are not contradictory positions. Go was designed to address the problems faced in software development at Google, which led to a **language** that is not a breakthrough research **language** but is nonetheless an excellent tool for engineering large software projects.

3. Go at Google

Go is a programming **language** designed by Google to help solve Google's problems, and Google has big problems.

The hardware is big and the software is big. There are many millions of lines of software, with servers mostly in C++ and lots of Java and Python for the other pieces. Thousands of engineers work on the code, at the "head" of a single tree comprising all the software, so from day to day there are significant changes to all levels of the tree. A large custom-designed distributed build system makes development at this scale feasible, but it's still big.

And of course, all this software runs on zillions of machines, which are treated as a modest number of independent, networked compute clusters.

In short, development at Google is big, can be slow, and is often clumsy. But it *is* effective.

The goals of the Go project were to eliminate the slowness and clumsiness of software development at Google, and thereby to make the process more productive and scalable. The **language** was designed by and for people who write—and read and debug and maintain—large software systems.

Go's purpose is therefore *not* to do research into programming **language design**; it is to improve the working environment for its designers and their coworkers. Go is more about software engineering than programming **language** research. Or to rephrase, it is about **language design** in the **service** of software engineering.

But how can a **language** help **software engineering**? The rest of this article is an answer to that question.

4. Pain points

When **Go** launched, some claimed it was missing particular features or methodologies that were regarded as *de rigueur* for a modern **language**. How could **Go** be worthwhile in the absence of these facilities? Our answer to that is that the properties **Go** *does* have address the issues that make large-scale **software** development difficult. These issues include:

- slow builds
- uncontrolled dependencies
- each programmer using a different subset of the **language**
- poor program understanding (code hard to read, poorly documented, and so on)
- duplication of effort
- cost of updates
- version skew
- difficulty of writing automatic tools
- cross-**language** builds

Individual features of a **language** don't address these issues. A larger view of **software engineering** is required, and in the **design** of **Go** we tried to focus on solutions to *these* problems.

As a simple, self-contained example, consider the representation of program structure. Some observers objected to **Go**'s C-like block structure with braces, preferring the use of spaces for indentation, in the style of Python or Haskell. However, we have had extensive experience tracking down build and test failures caused by cross-**language** builds where a Python snippet embedded in another **language**, for instance through a SWIG invocation, is subtly and *invisibly* broken by a change in the indentation of the surrounding code. Our position is therefore that, although spaces for indentation is nice for small programs, it doesn't scale well, and the bigger and more heterogeneous the code base, the more trouble it can cause. It is better to forgo convenience for safety and dependability, so **Go** has brace-bounded blocks.

5. Dependencies in C and C++

A more substantial illustration of scaling and other issues arises in the handling of package dependencies. We begin the discussion with a review of how they work in C and C++.

ANSI C, first standardized in 1989, promoted the idea of `#ifndef` "guards" in the standard header files. The idea, which is ubiquitous now, is that each header file be bracketed with a conditional compilation clause so that the file may be included multiple times without error. For instance, the Unix header file `<sys/stat.h>` looks schematically like this:

```
/* Large copyright and licensing notice */
#ifndef _SYS_STAT_H_
#define _SYS_STAT_H_
/* Types and other definitions */
#endif
```

The intent is that the C preprocessor reads in the file but disregards the contents on the second and subsequent readings of the file. The symbol `_SYS_STAT_H_`, defined the first time the file is read, "guards" the invocations that follow.

This **design** has some nice properties, most important that each header file can safely `#include` all its dependencies, even if other header files will also include them. If that rule is followed, it permits orderly code that, for instance, sorts the `#include` clauses alphabetically.

But it scales very badly.

In 1984, a compilation of `ps.c`, the source to the Unix `ps` command, was observed to `#include <sys/stat.h>` 37 times by the time all the preprocessing had been done. Even though the contents are discarded 36 times while doing so, most C implementations would open the file, read it, and scan it all 37 times. Without great cleverness, in fact, that behavior is required by the potentially complex macro semantics of the C preprocessor.

The effect on **software** is the gradual accumulation of `#include` clauses in C programs. It won't break a program to add them, and it's very hard to know when they are no longer needed. Deleting a `#include` and compiling the program again isn't even sufficient to test that, since another `#include` might itself contain a `#include` that pulls it in anyway.

Technically speaking, it does not have to be like that. Realizing the long-term problems with the use of `#ifndef` guards, the designers of the Plan 9 libraries took a different, non-ANSI-standard approach. In Plan 9, header files were forbidden from containing further `#include` clauses; all `#includes` were required to be in the top-level C file. This required some discipline, of course—the programmer was required to list the necessary dependencies exactly once, in the correct order—but documentation helped and in practice it worked very well. The result was that, no matter how many dependencies a C source file had, each `#include` file was read exactly once when compiling that file. And, of course, it was also easy to see if an `#include` was necessary by taking it out: the edited program would compile if and only if the dependency was unnecessary.

The most important result of the Plan 9 approach was much faster compilation: the amount of I/O the compilation requires can be dramatically less than when compiling a program using libraries with `#ifndef` guards.

Outside of Plan 9, though, the "guarded" approach is accepted practice for C and C++. In fact, C++ exacerbates the problem by using the same approach at finer granularity. By convention, C++ programs are usually structured with one header file per class, or perhaps small set of related classes, a grouping much smaller than, say, `<stdio.h>`. The dependency tree is therefore much more intricate, reflecting not library dependencies but the full type hierarchy. Moreover, C++ header files usually contain real code—type, method, and template declarations—not just the simple constants and function signatures typical of a C header file. Thus not only does C++ push more to the compiler, what it pushes is harder to compile, and each invocation of the compiler must reprocess this information. When building a large C++ binary, the compiler might be taught thousands of times how to represent a string by processing the header file `<string>`. (For the record, around 1984 Tom Cargill observed that the use of the C preprocessor for dependency management would be a long-term liability for C++ and should be addressed.)

The construction of a single C++ binary at Google can open and read hundreds of individual header files tens of thousands of times. In 2007, build engineers at Google instrumented the compilation of a major Google binary. The file contained about two thousand files that, if simply concatenated together, totaled 4.2 megabytes. By the time the `#includes` had been expanded, over 8 gigabytes were being delivered to the input of the compiler, a blow-up of 2000 bytes for every C++ source byte.

As another data point, in 2003 Google's build system was moved from a single Makefile to a per-directory **design** with better-managed, more explicit dependencies. A typical binary shrank about 40% in file size, just from having more accurate dependencies recorded. Even so, the properties of C++ (or C for that matter) make it impractical to verify those dependencies automatically, and today we still do not have an accurate understanding of the dependency requirements of large Google C++ binaries.

The consequence of these uncontrolled dependencies and massive scale is that it is impractical to build Google server binaries on a single computer, so a large distributed compilation system was created. With this system, involving many machines, much caching, and much complexity (the build system is a large program in its own right), builds at Google are practical, if still cumbersome.

Even with the distributed build system, a large Google build can still take many minutes. That 2007 binary took 45 minutes using a precursor distributed build system; today's version of the same program takes 27 minutes, but of course the program and its dependencies have grown in the interim. The **engineering** effort required to scale up the build system has barely been able to stay ahead of the growth of the **software** it is constructing.

6. Enter Go

When builds are slow, there is time to think. The origin myth for **Go** states that it was during one of those 45 minute builds that **Go** was conceived. It was believed to be worth trying to **design** a new **language** suitable for writing large Google programs such as web servers, with **software engineering** considerations that would improve the quality of life of Google programmers.

Although the discussion so far has focused on dependencies, there are many other issues that need attention. The primary considerations for any **language** to succeed in this context are:

- It must work at scale, for large programs with large numbers of dependencies, with large teams of programmers working on them.
- It must be familiar, roughly C-like. Programmers working at Google are early in their careers and are most familiar with procedural languages, particularly from the C family. The need to get programmers productive quickly in a new **language** means that the **language** cannot be too radical.
- It must be modern. C, C++, and to some extent Java are quite old, designed before the advent of multicore machines, networking, and web application development. There are features of the modern world that are better met by newer approaches, such as built-in concurrency.

With that background, then, let us look at the **design** of **Go** from a **software engineering** perspective.

7. Dependencies in Go

Since we've taken a detailed look at dependencies in C and C++, a good place to start our tour is to see how **Go** handles them. Dependencies are defined, syntactically and semantically, by the **language**. They are explicit, clear, and "computable", which is to say, easy to write tools to analyze.

The syntax is that, after the `package` clause (the subject of the next section), each source file may have one or more import statements, comprising the `import` keyword and a string constant identifying the package to be imported into this source file (only):

```
import "encoding/json"
```

The first step to making **Go** scale, dependency-wise, is that the ***language*** defines that unused dependencies are a compile-time error (not a warning, an *error*). If the source file imports a package it does not use, the program will not compile. This guarantees by construction that the dependency tree for any **Go** program is precise, that it has no extraneous edges. That, in turn, guarantees that no extra code will be compiled when building the program, which minimizes compilation time.

There's another step, this time in the implementation of the compilers, that goes even further to guarantee efficiency. Consider a Go program with three packages and this dependency graph:

- package A imports package B;
- package B imports package C;
- package A does *not* import package C

This means that package A uses C only transitively through its use of B; that is, no identifiers from C are mentioned in the source code to A, even if some of the items A is using from B do mention C. For instance, package A might reference a `struct` type defined in B that has a field with a type defined in C but that A does not reference itself. As a motivating example, imagine that A imports a formatted I/O package B that uses a buffered I/O implementation provided by C, but that A does not itself invoke buffered I/O.

To build this program, first, C is compiled; dependent packages must be built before the packages that depend on them. Then B is compiled; finally A is compiled, and then the program can be linked.

When A is compiled, the compiler reads the object file for B, not its source code. That object file for B contains all the type information necessary for the compiler to execute the

```
import "B"
```

clause in the source code for A. That information includes whatever information about C that clients of B will need at compile time. In other words, when B is compiled, the generated object file includes type information for all dependencies of B that affect the public interface of B.

This **design** has the important effect that when the compiler executes an import clause, *it opens exactly one file*, the object file identified by the string in the import clause. This is, of course, reminiscent of the Plan 9 C (as opposed to ANSI C) approach to dependency management, except that, in effect, the compiler writes the header file when the Go source file is compiled. The process is more automatic and even more efficient than in Plan 9 C, though: the data being read when evaluating the import is just "exported" data, not general program source code. The effect on overall compilation time can be huge, and scales well as the code base grows. The time to execute the dependency graph, and hence to compile, can be exponentially less than in the "include of include file" model of C and C++.

It's worth mentioning that this general approach to dependency management is not original; the ideas go back to the 1970s and flow through languages like Modula-2 and Ada. In the C family Java has elements of this approach.

To make compilation even more efficient, the object file is arranged so the export data is the first thing in the file, so the compiler can stop reading as soon as it reaches the end of that section.

This approach to dependency management is the single biggest reason why Go compilations are faster than C or C++ compilations. Another factor is that Go places the export data in the object file; some languages require the author to write or the compiler to generate a second file with that information. That's twice as many files to open. In Go there is only one file to open to import a package. Also, the single file approach means that the export data (or header file, in C/C++) can never go out of date relative to the object file.

For the record, we measured the compilation of a large Google program written in Go to see how the source code fanout compared to the C++ analysis done earlier. We found it was about 40X, which is fifty times better than C++ (as well as being simpler and hence faster to process), but it's still bigger than we expected. There are two reasons for this. First, we found a bug: the Go compiler was

generating a substantial amount of data in the export section that did not need to be there. Second, the export data uses a verbose encoding that could be improved. We plan to address these issues.

Nonetheless, a factor of fifty less to do turns minutes into seconds, coffee breaks into interactive builds.

Another feature of the Go dependency graph is that it has no cycles. The **language** defines that there can be no circular imports in the graph, and the compiler and linker both check that they do not exist. Although they are occasionally useful, circular imports introduce significant problems at scale. They require the compiler to deal with larger sets of source files all at once, which slows down incremental builds. More important, when allowed, in our experience such imports end up entangling huge swaths of the source tree into large subpieces that are difficult to manage independently, bloating binaries and complicating initialization, testing, refactoring, releasing, and other tasks of software development.

The lack of circular imports causes occasional annoyance but keeps the tree clean, forcing a clear demarcation between packages. As with many of the **design** decisions in Go, it forces the programmer to think earlier about a larger-scale issue (in this case, package boundaries) that if left until later may never be addressed satisfactorily.

Through the **design** of the standard library, great effort was spent on controlling dependencies. It can be better to copy a little code than to pull in a big library for one function. (A test in the system build complains if new core dependencies arise.) Dependency hygiene trumps code reuse. One example of this in practice is that the (low-level) `net` package has its own integer-to-decimal conversion routine to avoid depending on the bigger and dependency-heavy formatted I/O package. Another is that the string conversion package `strconv` has a private implementation of the definition of 'printable' characters rather than pull in the large Unicode character class tables; that `strconv` honors the Unicode standard is verified by the package's tests.

8. Packages

The **design** of Go's package system combines some of the properties of libraries, name spaces, and modules into a single construct.

Every Go source file, for instance `"encoding/json/json.go"`, starts with a package clause, like this:

```
package json
```

where `json` is the "package name", a simple identifier. Package names are usually concise.

To use a package, the importing source file identifies it by its *package path* in the import clause. The meaning of "path" is not specified by the **language**, but in practice and by convention it is the slash-separated directory path of the source package in the repository, here:

```
import "encoding/json"
```

Then the package name (as distinct from path) is used to qualify items from the package in the importing source file:

```
var dec = json.NewDecoder(reader)
```

This **design** provides clarity. One may always tell whether a name is local to package from its syntax: `Name` vs. `pkg.Name`. (More on this later.)

For our example, the package path is `"encoding/json"` while the package name is `json`. Outside the standard repository, the convention is to place the project or company name at the root of the name space:

```
import "google/base/go/log"
```

It's important to recognize that package *paths* are unique, but there is no such requirement for package *names*. The path must uniquely identify the package to be imported, while the name is just a convention for how clients of the package can refer to its contents. The package name need not be unique and can be overridden in each importing source file by providing a local identifier in the import clause. These two imports both reference packages that call themselves `package log`, but to import them in a single source file one must be (locally) renamed:

```
import "log"                          // Standard package
import googlelog "google/base/go/log" // Google-specific package
```

Every company might have its own `log` package but there is no need to make the package name unique. Quite the opposite: Go style suggests keeping package names short and clear and obvious in preference to worrying about collisions.

Another example: there are many `server` packages in Google's code base.

9. Remote packages

An important property of Go's package system is that the package path, being in general an arbitrary string, can be co-opted to refer to remote repositories by having it identify the URL of the site serving the repository.

Here is how to use the `doozer` package from `github`. The `go get` command uses the `go` build tool to fetch the repository from the site and install it. Once installed, it can be imported and used like any regular package.

```
$ go get github.com/4ad/doozer // Shell command to fetch package

import "github.com/4ad/doozer" // Doozer client's import statement

var client doozer.Conn         // Client's use of package
```

It's worth noting that the `go get` command downloads dependencies recursively, a property made possible only because the dependencies are explicit. Also, the allocation of the space of import paths is delegated to URLs, which makes the naming of packages decentralized and therefore scalable, in contrast to centralized registries used by other languages.

10. Syntax

Syntax is the user interface of a programming **language**. Although it has limited effect on the semantics of the **language**, which is arguably the more important component, syntax determines the readability and hence clarity of the **language**. Also, syntax is critical to tooling: if the **language** is hard to parse, automated tools are hard to write.

Go was therefore designed with clarity and tooling in mind, and has a clean syntax. Compared to other languages in the C family, its grammar is modest in size, with only 25 keywords (C99 has 37; C++11 has 84; the numbers continue to grow). More important, the grammar is regular and therefore

easy to parse (mostly; there are a couple of quirks we might have fixed but didn't discover early enough). Unlike C and Java and especially C++, Go can be parsed without type information or a symbol table; there is no type-specific context. The grammar is easy to reason about and therefore tools are easy to write.

One of the details of Go's syntax that surprises C programmers is that the declaration syntax is closer to Pascal's than to C's. The declared name appears before the type and there are more keywords:

```
var fn func([]int) int
type T struct { a, b int }
```

as compared to C's

```
int (*fn)(int[]);
struct T { int a, b; }
```

Declarations introduced by keyword are easier to parse both for people and for computers, and having the type syntax not be the expression syntax as it is in C has a significant effect on parsing: it adds grammar but eliminates ambiguity. But there is a nice side effect, too: for initializing declarations, one can drop the `var` keyword and just take the type of the variable from that of the expression. These two declarations are equivalent; the second is shorter and idiomatic:

```
var buf *bytes.Buffer = bytes.NewBuffer(x) // explicit
buf := bytes.NewBuffer(x)                  // derived
```

There is a blog post at golang.org/s/decl-syntax with more detail about the syntax of declarations in Go and why it is so different from C.

Function syntax is straightforward for simple functions. This example declares the function `Abs`, which accepts a single variable `x` of type `T` and returns a single `float64` value:

```
func Abs(x T) float64
```

A method is just a function with a special parameter, its *receiver*, which can be passed to the function using the standard "dot" notation. Method declaration syntax places the receiver in parentheses before the function name. Here is the same function, now as a method of type `T`:

```
func (x T) Abs() float64
```

And here is a variable (closure) with a type `T` argument; Go has first-class functions and closures:

```
negAbs := func(x T) float64 { return -Abs(x) }
```

Finally, in Go functions can return multiple values. A common case is to return the function result and an `error` value as a pair, like this:

```
func ReadByte() (c byte, err error)

c, err := ReadByte()
if err != nil { ... }
```

We'll talk more about errors later.

One feature missing from Go is that it does not support default function arguments. This was a deliberate simplification. Experience tells us that defaulted arguments make it too easy to patch over API **design** flaws by adding more arguments, resulting in too many arguments with interactions that are difficult to disentangle or even understand. The lack of default arguments requires more functions or methods to be defined, as one function cannot hold the entire interface, but that leads to a clearer

API that is easier to understand. Those functions all need separate names, too, which makes it clear which combinations exist, as well as encouraging more thought about naming, a critical aspect of clarity and readability.

One mitigating factor for the lack of default arguments is that Go has easy-to-use, type-safe support for variadic functions.

11. Naming

Go takes an unusual approach to defining the *visibility* of an identifier, the ability for a client of a package to use the item named by the identifier. Unlike, for instance, `private` and `public` keywords, in Go the name itself carries the information: the case of the initial letter of the identifier determines the visibility. If the initial character is an upper case letter, the identifier is *exported* (public); otherwise it is not:

- upper case initial letter: `Name` is visible to clients of package
- otherwise: `name` (or `_Name`) is not visible to clients of package

This rule applies to variables, types, functions, methods, constants, fields... everything. That's all there is to it.

This was not an easy **design** decision. We spent over a year struggling to define the notation to specify an identifier's visibility. Once we settled on using the case of the name, we soon realized it had become one of the most important properties about the **language**. The name is, after all, what clients of the package use; putting the visibility in the name rather than its type means that it's always clear when looking at an identifier whether it is part of the public API. After using Go for a while, it feels burdensome when going back to other languages that require looking up the declaration to discover this information.

The result is, again, clarity: the program source text expresses the programmer's meaning simply.

Another simplification is that Go has a very compact scope hierarchy:

- universe (predeclared identifiers such as `int` and `string`)
- package (all the source files of a package live at the same scope)
- file (for package import renames only; not very important in practice)
- function (the usual)
- block (the usual)

There is no scope for name space or class or other wrapping construct. Names come from very few places in Go, and all names follow the same scope hierarchy: at any given location in the source, an identifier denotes exactly one **language** object, independent of how it is used. (The only exception is statement labels, the targets of `break` statements and the like; they always have function scope.)

This has consequences for clarity. Notice for instance that methods declare an explicit receiver and that it must be used to access fields and methods of the type. There is no implicit `this`. That is, one always writes

```
rcvr.Field
```

(where rcvr is whatever name is chosen for the receiver variable) so all the elements of the type always appear lexically bound to a value of the receiver type. Similarly, a package qualifier is always present for imported names; one writes `io.Reader` not `Reader`. Not only is this clear, it frees up the identifier `Reader` as a useful name to be used in any package. There are in fact multiple exported

identifiers in the standard library with name `Reader`, or `Printf` for that matter, yet which one is being referred to is always unambiguous.

Finally, these rules combine to guarantee that, other than the top-level predefined names such as `int`, (the first component of) every name is always declared in the current package.

In short, names are local. In C, C++, or Java the name `y` could refer to anything. In **Go**, `y` (or even `Y`) is always defined within the package, while the interpretation of `x.Y` is clear: find `x` locally, `Y` belongs to it.

These rules provide an important property for scaling because they guarantee that adding an exported name to a package can never break a client of that package. The naming rules decouple packages, providing scaling, clarity, and robustness.

There is one more aspect of naming to be mentioned: method lookup is always by name only, not by signature (type) of the method. In other words, a single type can never have two methods with the same name. Given a method `x.M`, there's only ever one `M` associated with `x`. Again, this makes it easy to identify which method is referred to given only the name. It also makes the implementation of method invocation simple.

12. Semantics

The semantics of **Go** statements is generally C-like. It is a compiled, statically typed, procedural **language** with pointers and so on. By **design**, it should feel familiar to programmers accustomed to languages in the C family. When launching a new **language** it is important that the target audience be able to learn it quickly; rooting **Go** in the C family helps make sure that young programmers, most of whom know Java, JavaScript, and maybe C, should find **Go** easy to learn.

That said, **Go** makes many small changes to C semantics, mostly in the **service** of robustness. These include:

- there is no pointer arithmetic
- there are no implicit numeric conversions
- array bounds are always checked
- there are no type aliases (after `type X int`, `X` and `int` are distinct types not aliases)
- `++` and `--` are statements not expressions
- assignment is not an expression
- it is legal (encouraged even) to take the address of a stack variable
- and many more

There are some much bigger changes too, stepping far from the traditional C, C++, and even Java models. These include linguistic support for:

- concurrency
- garbage collection
- interface types
- reflection
- type switches

The following sections provide brief discussions of two of these topics in **Go**, concurrency and garbage collection, mostly from a **software engineering** perspective. For a full discussion of the **language** semantics and uses see the many resources on the golang.org web site.

13. Concurrency

Concurrency is important to the modern computing environment with its multicore machines running web servers with multiple clients, what might be called the typical Google program. This kind of **software** is not especially well served by C++ or Java, which lack sufficient concurrency support at the **language** level.

Go embodies a variant of CSP with first-class channels. CSP was chosen partly due to familiarity (one of us had worked on predecessor languages that built on CSP's ideas), but also because CSP has the property that it is easy to add to a procedural programming model without profound changes to that model. That is, given a C-like **language**, CSP can be added to the **language** in a mostly orthogonal way, providing extra expressive power without constraining the **language**'s other uses. In short, the rest of the **language** can remain "ordinary".

The approach is thus the composition of independently executing functions of otherwise regular procedural code.

The resulting **language** allows us to couple concurrency with computation smoothly. Consider a web server that must verify security certificates for each incoming client call; in **Go** it is easy to construct the **software** using CSP to manage the clients as independently executing procedures but to have the full power of an efficient compiled **language** available for the expensive cryptographic calculations.

In summary, CSP is practical for **Go** and for Google. When writing a web server, the canonical **Go** program, the model is a great fit.

There is one important caveat: **Go** is not purely memory safe in the presence of concurrency. Sharing is legal and passing a pointer over a channel is idiomatic (and efficient).

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. We compensate by convention, training programmers to think about message passing as a version of ownership control. The motto is, "Don't communicate by sharing memory, share memory by communicating."

Our limited experience with programmers new to both **Go** and concurrent programming shows that this is a practical approach. Programmers enjoy the simplicity that support for concurrency brings to network **software**, and simplicity engenders robustness.

14. Garbage collection

For a systems **language**, garbage collection can be a controversial feature, yet we spent very little time deciding that **Go** would be a garbage-collected **language**. **Go** has no explicit memory-freeing operation: the only way allocated memory returns to the pool is through the garbage collector.

It was an easy decision to make because memory management has a profound effect on the way a **language** works in practice. In C and C++, too much programming effort is spent on memory allocation and freeing. The resulting designs tend to expose details of memory management that could well be hidden; conversely memory considerations limit how they can be used. By contrast, garbage collection makes interfaces easier to specify.

Moreover, in a concurrent object-oriented **language** it's almost essential to have automatic memory management because the ownership of a piece of memory can be tricky to manage as it is passed around among concurrent executions. It's important to separate behavior from resource management.

The **language** is much easier to use because of garbage collection.

Of course, garbage collection brings significant costs: general overhead, latency, and complexity of the implementation. Nonetheless, we believe that the benefits, which are mostly felt by the programmer, outweigh the costs, which are largely borne by the **language** implementer.

Experience with Java in particular as a server **language** has made some people nervous about garbage collection in a user-facing system. The overheads are uncontrollable, latencies can be large, and much parameter tuning is required for good performance. Go, however, is different. Properties of the **language** mitigate some of these concerns. Not all of them of course, but some.

The key point is that Go gives the programmer tools to limit allocation by controlling the layout of data structures. Consider this simple type definition of a data structure containing a buffer (array) of bytes:

```
type X struct {
    a, b, c int
    buf [256]byte
}
```

In Java, the `buf` field would require a second allocation and accesses to it a second level of indirection. In Go, however, the buffer is allocated in a single block of memory along with the containing struct and no indirection is required. For systems programming, this **design** can have a better performance as well as reducing the number of items known to the collector. At scale it can make a significant difference.

As a more direct example, in Go it is easy and efficient to provide second-order allocators, for instance an arena allocator that allocates a large array of structs and links them together with a free list. Libraries that repeatedly use many small structures like this can, with modest prearrangement, generate no garbage yet be efficient and responsive.

Although Go is a garbage collected **language**, therefore, a knowledgeable programmer can limit the pressure placed on the collector and thereby improve performance. (Also, the Go installation comes with good tools for studying the dynamic memory performance of a running program.)

To give the programmer this flexibility, Go must support what we call *interior pointers* to objects allocated in the heap. The `X.buf` field in the example above lives within the struct but it is legal to capture the address of this inner field, for instance to pass it to an I/O routine. In Java, as in many garbage-collected languages, it is not possible to construct an interior pointer like this, but in Go it is idiomatic. This **design** point affects which collection algorithms can be used, and may make them more difficult, but after careful thought we decided that it was necessary to allow interior pointers because of the benefits to the programmer and the ability to reduce pressure on the (perhaps harder to implement) collector. So far, our experience comparing similar Go and Java programs shows that use of interior pointers can have a significant effect on total arena size, latency, and collection times.

In summary, Go is garbage collected but gives the programmer some tools to control collection overhead.

The garbage collector remains an active area of development. The current **design** is a parallel mark-and-sweep collector and there remain opportunities to improve its performance or perhaps even its **design**. (The **language** specification does not mandate any particular implementation of the collector.) Still, if the programmer takes care to use memory wisely, the current implementation works well for production use.

15. Composition not inheritance

Go takes an unusual approach to object-oriented programming, allowing methods on any type, not just classes, but without any form of type-based inheritance like subclassing. This means there is no type hierarchy. This was an intentional **design** choice. Although type hierarchies have been used to build much successful software, it is our opinion that the model has been overused and that it is worth taking a step back.

Instead, Go has *interfaces*, an idea that has been discussed at length elsewhere (see research.swtch.com/interfaces for example), but here is a brief summary.

In Go an interface is *just* a set of methods. For instance, here is the definition of the `Hash` interface from the standard library.

```
type Hash interface {
    Write(p []byte) (n int, err error)
    Sum(b []byte) []byte
    Reset()
    Size() int
    BlockSize() int
}
```

All data types that implement these methods satisfy this interface implicitly; there is no `implements` declaration. That said, interface satisfaction is statically checked at compile time so despite this decoupling interfaces are type-safe.

A type will usually satisfy many interfaces, each corresponding to a subset of its methods. For example, any type that satisfies the `Hash` interface also satisfies the `Writer` interface:

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

This fluidity of interface satisfaction encourages a different approach to software construction. But before explaining that, we should explain why Go does not have subclassing.

Object-oriented programming provides a powerful insight: that the *behavior* of data can be generalized independently of the *representation* of that data. The model works best when the behavior (method set) is fixed, but once you subclass a type and add a method, *the behaviors are no longer identical*. If instead the set of behaviors is fixed, such as in Go's statically defined interfaces, the uniformity of behavior enables data and programs to be composed uniformly, orthogonally, and safely.

One extreme example is the Plan 9 kernel, in which all system data items implemented exactly the same interface, a file system API defined by 14 methods. This uniformity permitted a level of object composition seldom achieved in other systems, even today. Examples abound. Here's one: A system could import (in Plan 9 terminology) a TCP stack to a computer that didn't have TCP or even Ethernet, and over that network connect to a machine with a different CPU architecture, import its `/proc` tree, and run a local debugger to do breakpoint debugging of the remote process. This sort of operation was workaday on Plan 9, nothing special at all. The ability to do such things fell out of the **design**; it required no special arrangement (and was all done in plain C).

We argue that this compositional style of system construction has been neglected by the languages that push for **design** by type hierarchy. Type hierarchies result in brittle code. The hierarchy must be designed early, often as the first step of designing the program, and early decisions can be difficult to

change once the program is written. As a consequence, the model encourages early overdesign as the programmer tries to predict every possible use the software might require, adding layers of type and abstraction just in case. This is upside down. The way pieces of a system interact should adapt as it grows, not be fixed at the dawn of time.

Go therefore encourages *composition* over inheritance, using simple, often one-method interfaces to define trivial behaviors that serve as clean, comprehensible boundaries between components.

Consider the `Writer` interface shown above, which is defined in package `io`: Any item that has a `Write` method with this signature works well with the complementary `Reader` interface:

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

These two complementary methods allow type-safe chaining with rich behaviors, like generalized Unix pipes. Files, buffers, networks, encryptors, compressors, image encoders, and so on can all be connected together. The `Fprintf` formatted I/O routine takes an `io.Writer` rather than, as in C, a `FILE*`. The formatted printer has no knowledge of what it is writing to; it may be a image encoder that is in turn writing to a compressor that is in turn writing to an encryptor that is in turn writing to a network connection.

Interface composition is a different style of programming, and people accustomed to type hierarchies need to adjust their thinking to do it well, but the result is an adaptability of **design** that is harder to achieve through type hierarchies.

Note too that the elimination of the type hierarchy also eliminates a form of dependency hierarchy. Interface satisfaction allows the program to grow organically without predetermined contracts. And it is a linear form of growth; a change to an interface affects only the immediate clients of that interface; there is no subtree to update. The lack of `implements` declarations disturbs some people but it enables programs to grow naturally, gracefully, and safely.

Go's interfaces have a major effect on program **design**. One place we see this is in the use of functions that take interface arguments. These are *not* methods, they are functions. Some examples should illustrate their power. `ReadAll` returns a byte slice (array) holding all the data that can be read from an `io.Reader`:

```
func ReadAll(r io.Reader) ([]byte, error)
```

Wrappers—functions that take an interface and return an interface—are also widespread. Here are some prototypes. `LoggingReader` logs every `Read` call on the incoming `Reader`. `LimitingReader` stops reading after `n` bytes. `ErrorInjector` aids testing by simulating I/O errors. And there are many more.

```
func LoggingReader(r io.Reader) io.Reader
func LimitingReader(r io.Reader, n int64) io.Reader
func ErrorInjector(r io.Reader) io.Reader
```

The designs are nothing like hierarchical, subtype-inherited methods. They are looser (even *ad hoc*), organic, decoupled, independent, and therefore scalable.

16. Errors

Go does not have an exception facility in the conventional sense, that is, there is no control structure associated with error handling. (Go does provide mechanisms for handling exceptional situations such as division by zero. A pair of built-in functions called `panic` and `recover` allow the programmer to

protect against such things. However, these functions are intentionally clumsy, rarely used, and not integrated into the library the way, say, Java libraries use exceptions.)

The key **language** feature for error handling is a pre-defined interface type called `error` that represents a value that has an `Error` method returning a string:

```
type error interface {
    Error() string
}
```

Libraries use the `error` type to return a description of the error. Combined with the ability for functions to return multiple values, it's easy to return the computed result along with an error value, if any. For instance, the equivalent to C's `getchar` does not return an out-of-band value at EOF, nor does it throw an exception; it just returns an `error` value alongside the character, with a `nil error` value signifying success. Here is the signature of the `ReadByte` method of the buffered I/O package's `bufio.Reader` type:

```
func (b *Reader) ReadByte() (c byte, err error)
```

This is a clear and simple **design**, easily understood. Errors are just values and programs compute with them as they would compute with values of any other type.

It was a deliberate choice not to incorporate exceptions in **Go**. Although a number of critics disagree with this decision, there are several reasons we believe it makes for better **software**.

First, there is nothing truly exceptional about errors in computer programs. For instance, the inability to open a file is a common issue that does not deserve special linguistic constructs; `if` and `return` are fine.

```
f, err := os.Open(fileName)
if err != nil {
    return err
}
```

Also, if errors use special control structures, error handling distorts the control flow for a program that handles errors. The Java-like style of `try-catch-finally` blocks interlaces multiple overlapping flows of control that interact in complex ways. Although in contrast **Go** makes it more verbose to check errors, the explicit **design** keeps the flow of control straightforward—literally.

There is no question the resulting code can be longer, but the clarity and simplicity of such code offsets its verbosity. Explicit error checking forces the programmer to think about errors—and deal with them—when they arise. Exceptions make it too easy to *ignore* them rather than *handle* them, passing the buck up the call stack until it is too late to fix the problem or diagnose it well.

17. Tools

Software engineering requires tools. Every **language** operates in an environment with other languages and myriad tools to compile, edit, debug, profile, test, and run programs.

Go's syntax, package system, naming conventions, and other features were designed to make tools easy to write, and the library includes a lexer, parser, and type checker for the **language**.

Tools to manipulate **Go** programs are so easy to write that many such tools have been created, some with interesting consequences for **software engineering**.

The best known of these is `gofmt`, the Go source code formatter. From the beginning of the project, we intended Go programs to be formatted by machine, eliminating an entire class of argument between programmers: how do I lay out my code? `Gofmt` is run on all Go programs we write, and most of the open source community uses it too. It is run as a "presubmit" check for the code repositories to make sure that all checked-in Go programs are formatted the same.

`Gofmt` is often cited by users as one of Go's best features even though it is not part of the **language**. The existence and use of `gofmt` means that from the beginning, the community has always seen Go code as `gofmt` formats it, so Go programs have a single style that is now familiar to everyone. Uniform presentation makes code easier to read and therefore faster to work on. Time not spent on formatting is time saved. `Gofmt` also affects scalability: since all code looks the same, teams find it easier to work together or with others' code.

`Gofmt` enabled another class of tools that we did not foresee as clearly. The program works by parsing the source code and reformatting it from the parse tree itself. This makes it possible to *edit* the parse tree before formatting it, so a suite of automatic refactoring tools sprang up. These are easy to write, can be semantically rich because they work directly on the parse tree, and automatically produce canonically formatted code.

The first example was a `-r` (rewrite) flag on `gofmt` itself, which uses a simple pattern-matching **language** to enable expression-level rewrites. For instance, one day we introduced a default value for the right-hand side of a slice expression: the length itself. The entire Go source tree was updated to use this default with the single command:

```
gofmt -r 'a[b:len(a)] -> a[b:]'
```

A key point about this transformation is that, because the input and output are both in the canonical format, the only changes made to the source code are semantic ones.

A similar but more intricate process allowed `gofmt` to be used to update the tree when the **language** no longer required semicolons as statement terminators if the statement ended at a newline.

Another important tool is `gofix`, which runs tree-rewriting modules written in Go itself that are therefore are capable of more advanced refactorings. The `gofix` tool allowed us to make sweeping changes to APIs and **language** features leading up to the release of Go 1, including a change to the syntax for deleting entries from a map, a radically different API for manipulating time values, and many more. As these changes rolled out, users could update all their code by running the simple command

```
gofix
```

Note that these tools allow us to *update* code even if the old code still works. As a result, Go repositories are easy to keep up to date as libraries evolve. Old APIs can be deprecated quickly and automatically so only one version of the API needs to be maintained. For example, we recently changed Go's protocol buffer implementation to use "getter" functions, which were not in the interface before. We ran `gofix` on *all* of Google's Go code to update all programs that use protocol buffers, and now there is only one version of the API in use. Similar sweeping changes to the C++ or Java libraries are almost infeasible at the scale of Google's code base.

The existence of a parsing package in the standard Go library has enabled a number of other tools as well. Examples include the `go` tool, which manages program construction including acquiring packages from remote repositories; the `godoc` document extractor, a program to verify that the API compatibility contract is maintained as the library is updated, and many more.

Although tools like these are rarely mentioned in the context of **language design**, they are an integral part of a **language**'s ecosystem and the fact that Go was designed with tooling in mind has a huge

effect on the development of the **language**, its libraries, and its community.

18. Conclusion

Go's use is growing inside Google.

Several big user-facing services use it, including `youtube.com` and `dl.google.com` (the download server that delivers Chrome, Android and other downloads), as well as our own golang.org. And of course many small ones do, mostly built using Google App Engine's native support for Go.

Many other companies use Go as well; the list is very long, but a few of the better known are:

- BBC Worldwide
- Canonical
- Heroku
- Nokia
- SoundCloud

It looks like Go is meeting its goals. Still, it's too early to declare it a success. We don't have enough experience yet, especially with big programs (millions of lines of code) to know whether the attempts to build a scalable **language** have paid off. All the indicators are positive though.

On a smaller scale, some minor things aren't quite right and might get tweaked in a later (Go 2?) version of the **language**. For instance, there are too many forms of variable declaration syntax, programmers are easily confused by the behavior of nil values inside non-nil interfaces, and there are many library and interface details that could use another round of **design**.

It's worth noting, though, that `gofix` and `gofmt` gave us the opportunity to fix many other problems during the leadup to Go version 1. Go as it is today is therefore much closer to what the designers wanted than it would have been without these tools, which were themselves enabled by the **language**'s **design**.

Not everything was fixed, though. We're still learning (but the **language** is frozen for now).

A significant weakness of the **language** is that the implementation still needs work. The compilers' generated code and the performance of the runtime in particular should be better, and work continues on them. There is progress already; in fact some benchmarks show a doubling of performance with the development version today compared to the first release of Go version 1 early in 2012.

19. Summary

Software engineering guided the **design** of Go. More than most general-purpose programming languages, Go was designed to address a set of software engineering issues that we had been exposed to in the construction of large server software. Offhand, that might make Go sound rather dull and industrial, but in fact the focus on clarity, simplicity and composability throughout the **design** instead resulted in a productive, fun **language** that many programmers find expressive and powerful.

The properties that led to that include:

- Clear dependencies
- Clear syntax
- Clear semantics

- Composition over inheritance
- Simplicity provided by the programming model (garbage collection, concurrency)
- Easy tooling (the `go` tool, `gofmt`, `godoc`, `gofix`)

If you haven't tried **Go** already, we suggest you do.

http://golang.org