Should I stay or should I Go?

Ralf Wirdemann, kommitment GmbH & Co. KG

Go ist eine einfache, statisch getypte und kompilierte Programmiersprache. Einfaches Deployment, hohe Performance und sehr guter HTTP- und JSON-Support machen sie zur ersten Wahl für die Entwicklung von REST-APIs. Durch hohe Lesbarkeit ist Go zudem eine Sprache für produktive Softwareteams.

Go wurde im Jahr 2007 als Antwort auf die zunehmende Komplexität der Software-Entwicklung von Google ins Leben gerufen. Inzwischen setzt neben Google eine Reihe weiterer bekannterer Firmen Go ein. Das bekannteste Beispiel ist sicherlich Docker, aber auch Firmen wie Dropbox sind dabei, ihre Software auf Go umzustellen. Bei Dropbox sind es mittlerweile fünfzehn Go-Teams, deren Repositories mehr als 1.3 Millionen Zeilen Go-Code enthalten. Ein anderes Beispiel sind die Wunderkinder aus Berlin, die ihre sehr erfolgreiche Aufgabenverwaltung Wunderlist von einem Rails-Monolithen auf eine Microservice-Architektur umgestellt haben, bei der die meisten Services in Go programmiert sind.

Dieser Artikel führt in die Grundlagen der Programmiersprache Go ein, erklärt die zugrunde liegenden Paradigmen und macht sich auf die Suche nach Gründen für die rasche Verbreitung von Go. Darüber hinaus beleuchtet der Artikel die Aspekte von Go, die nicht so gut gelungen sind. Aufbauend auf diesen Pros und Kontras schließt der Artikel mit einem Fazit und einer Antwort auf die Frage: „Should I stay or should I Go?".

Der Go-Workspace

Der Go-Workspace ist ein zentrales Verzeichnis, das über die Umgebungsvariable „GOPATH" referenziert wird. Darunter finden sich die drei Unterverzeichnisse „bin", „pkg" und „src", die Binaries, Packages und Sourcen der im Workspace verwalteten Projekte enthalten *(siehe Listing 1)*.

Die zugrunde liegende Idee ist, dass alle Go-Projekte inklusive der benötigten Abhängigkeiten an zentraler Stelle abgelegt sind. Pro-

jekte werden so schnell gefunden und können sich benötigte Bibliotheken teilen.

Die Sprache Go

Der Einstieg in ein Go-Programm erfolgt über die Funktion „main", die bei Programmstart ausgeführt wird *(siehe Listing 2)*. Das Programm lässt sich durch Eingabe des Kommandozeilen-Befehls „go run main.go" direkt ausführen.

Packages

Go-Files werden in Packages organisiert; alle Dateien eines Verzeichnisses gehören zum selben Package. Der letzte Teil des Pfadnamens bestimmt den Package-Namen, der als erstes Statement in den Dateien des Packages deklariert wird *(siehe Listing 3)*.

Packages definieren Namensräume: Alles Kleingeschriebene bleibt innerhalb des Packages, alles Großgeschriebene ist auch außerhalb des Packages sichtbar. Packages werden per „import"-Statement importiert. Öffentliche Typen, Variablen und Funktionen werden genutzt, indem ihnen der Package-Name vorangestellt wird. So wird die öffentliche Funktion „Greet" des Packages „service" wie in *Listing 4* importiert und aufgerufen.

Das „main"-Beispiel zeigt eine Ausnahme in Bezug auf die Namenskonvention für Packages: Die Go-Datei mit der Funktion „main" gehört zum Package „main", unabhängig vom Verzeichnis, in dem sie liegt.

Funktionen und Variablen

Funktionen in Go werden mit dem Schlüsselwort „func", gefolgt vom Namen einer optionalen Parameterliste sowie keinem, einem oder mehreren Rückgabewerten deklariert *(siehe Listing 5)*. Variablen sind mit dem Schlüsselwort „var" oder per Direktzuweisung deklariert *(siehe Listing 6)*. Bei der Direktzuweisung erkennt der Compiler den Typ der Variablen, sodass die explizite Angabe des Typs „string" entfällt und die Variable direkt initialisiert wird.

Go enthält die gängigen Standardtypen, wie „string", „int", „float32/64", „byte" oder „bool". Darauf aufbauend können benutzerdefinierte Typen entweder als einfache oder als zusammengesetzte Typen deklariert sein *(siehe Listing 7)*.

Zusammengesetzte Typen werden per Literal instanziiert, indem dem Typnamen eine in geschweiften Klammern angegebene Parameterliste übergeben wird, beispielsweise „g := Greeting{message: „Hello", greetee: „Java Forum"}".

Methoden

Eine Funktion wird zu einer Methode, indem dem Funktionsnamen der Typ vorangestellt ist, auf dem die Methode implementiert werden soll. Das Beispiel in *Listing 8* implementiert eine Methode „Greet" auf dem Typ „Greeting".

Innerhalb des Methodenrumpfs wird das Objekt, auf dem die Methode arbeitet, durch die dem Typ vorangestellte Variable repräsentiert (im Beispiel „g"). Der Aufruf einer Methode erfolgt über die Punkt-Notation *(siehe Listing 9)*.

Pointer

Go behandelt Funktionsparameter nach einer „Call by Value"-Se-

```
bin/
  treubuilder
pkg/
  linux_amd64/
     bitbucket.org/rwirdemann/javaforum/
        service.a
src/
  bitbucket.org/rwirdemann/javaforum/
     main.go
     service/
        hello.go
```

Listing 1

```
package main

func main() {
  println("Hello, Java")
}
```

Listing 2

```
// /home/ralf/gows/src/hello/service/greet.go
package service

func Greet(greetee string) {
}
```

Listing 3

```
package main

import "service"

func main() {
     service.Greet("Kalle")
}
```

Listing 4

```
func swap(x int, y int) (int, int) {
     return y, x
}
```

Listing 5

```
var vorname string // oder
nachname := "Meyer"
```

Listing 6

```
// Einfacher Typ basierten auf string
type message string

// Zusammengesetzter Typ
type Greeting struct {
     message string
     greetee string
}
```

Listing 7

```
func (g Greeting) Greet() string {
  return g.message + ", " + g.greetee
}
```

Listing 8

mantik. Entsprechend arbeiten Funktionen und Methoden auf Kopien ihrer Parameter beziehungsweise Objekte. Gemäß dieser Semantik ist die Methode „Greet" Seiteneffekt-frei, da sie nicht auf dem in „main" instanziierten „greeting"-Objekt, sondern auf einer Kopie davon arbeitet.

Seiteneffekte werden in Go durch den Einsatz von Pointern erzwungen. Ein Pointer ist eine Variable, die statt eines konkreten Wertes eine Speicheradresse enthält. Diese Adresse zeigt auf die Stelle im Hauptspeicher, an der der eigentliche Wert gespeichert ist. Pointer werden mit einem dem Typ vorangestellten Stern gekennzeichnet: „func (g *greeting) Greet() string {}".

Ein Pointer wird entweder über die Funktion „new" oder den Operator „&" erzeugt: „greeting := &Greeting{message: „Hello", greetee: „Java Forum"}". Methodenaufrufe auf Pointern entsprechen denen auf normalen Variablen: „greeting.Greet()". Der entscheidende Unterschied: Die Methode „Greet" arbeitet jetzt nicht mehr auf einer Kopie von „greeting", sondern auf der in „main" instanziierten Version. Entsprechend können Methoden auf Pointer-Typen die an sie gebundenen Objekte verändern.

Ein Vorteil der standardmäßigen „Call by Value"-Semantik von Go ist, dass Funktions- und Methodenaufrufe immer immutable sind, solange sie keinen globalen Zustand verändern. Erst durch die Verwendung von Pointern wird einer Funktion beziehungsweise Methode explizit erlaubt, Seiteneffekte zu erzeugen.

Interfaces

Go-Interfaces haben einen Namen und definieren eine Reihe von Methoden *(siehe Listing 10)*. Ein Typ erfüllt ein Interface, wenn er alle Methoden des Interface implementiert. Entsprechend erfüllt Greeting das Interface Printable, wenn Greeting die Methode „print" implementiert *(siehe Listing 11)*. Interfaces können überall dort eingesetzt werden, wo zuvor der konkrete Typ verwendet wurde, zum Beispiel als Funktionsparameter *(siehe Listing 12)*.

Arrays und Slices

Neben den bekannten Basistypen enthält Go mit Arrays und Slices zwei weitere wichtige Typen. Arrays sind Listen gleichen Typs mit fester Länge: „a := [3]int{1, 2, 3}". Arrays sind immutable und können nur verändert werden, indem sie kopiert werden. Beispielsweise erzeugt die Funktion „append" ein neues Array, das aus dem übergebenen Array und dem im zweiten Parameter übergebenen Element besteht: „a = append(a, 4)".

Die eigentliche Arbeit auf Arrays erfolgt mithilfe von Slices. Im Gegensatz zu Arrays besitzen Slices keine feste Länge, sondern definieren eine dynamische Sicht auf Arrays: „var s[]int = a[1:3]". Die Slice „s" enthält die Elemente „2" und „3". Wird einem Element der Slice ein neuer Wert zugewiesen, ändert sich nicht das unterliegende Array, sondern es wird intern eine Kopie des Arrays erzeugt und der Variable „a" zugewiesen: „s[0] = 1 // a => [1, 1, 3]".

Maps

Neben Arrays sind Maps der zweite wichtige Container-Datentyp in Go. Maps enthalten Key-Value-Paare und werden von der Funktion „make" oder per Literal erzeugt: „m := make(map[string] Contact)". Die Zuweisung von Elementen erfolgt durch Angabe des

```
func main() {
  greeting := Greeting{message: "Hello2, greetee:
"Java"}
  greeting.Greet()
}
```

Listing 9

```
type Printable interface {
  print() string
}
```

Listing 10

```
func (g Greeting) print() string {
  return fmt.Sprintf("%s, %s", g.message, g.greetee)
}
```

Listing 11

```
func printOnConsole(printable Printable) {
  fmt.Printf("=> %s", printable.print());
}
```

Listing 12

```
for i := 0; i < 10; i++  // index-basierte Iteration
for i < 10               // while
for                      // for ever
```

Listing 13

```
switch r.Method {
case "GET":
  ...
case "POST":
  ...
default:
  ...
}
```

Listing 14

Keys in geschweiften Klammern: „m[„ralf"] = Contact{name: „Ralf Wirdemann"}". Bereits vorhandene Werte werden überschrieben. Die Überprüfung, ob die Map einen Wert für einen bestimmten Key enthält, erfolgt über das Ok-Idiom: „if v, ok := m[„ralf"]; ok {}". Die Variablen „v" und „ok" sind nur innerhalb des „if"-Blocks gültig.

Schleifen und Bedingungen

Go kennt nur eine Schleife, die „for"-Schleife. Je nach Parameter erfüllt die Schleife unterschiedliche Zwecke *(siehe Listing 13)*.

Für Verzweigungen kennt Go die beiden Schlüsselworte „if" und „switch". „if" haben wir bereits im Zusammenhang mit Maps kennengelernt. Das Besondere am „switch"-Statement ist der Wegfall von „break" in den einzelnen „case"-Zweigen *(siehe Listing 14)*.

Das Schlüsselwort „break" ist optional und kann verwendet werden, um einen „case"-Block vor der kompletten Abarbeitung abzuschließen. Das optionale Schlüsselwort „fallthrough" kann am Ende eines

```
switch {
case height <= 4:
  fmt.Println("Short")
case height <= 5:
  fmt.Println("Normal")
case height > 5:
  fmt.Println("Tall")
}
```

Listing 15

```
go run main.go    // führt main.go direkt aus
go build hello    // übersetzt Projekt "hello",
                  // Binary landet im aktuellen Ver-
zeichnis
go install hello  // übersetzt Projekt "hello",
                  // Binary im bin Verzeichnis
go test           // führt Tests im aktuellen Ver-
zeichnis aus
```

Listing 16

```
type Engine struct {
  hp int
}

type Car struct {
  Engine
}
```

Listing 17

```
func (e *Engine) start() {...}

func main() {
  var c Car
  c.hp = 100
  c.start()
}
```

Listing 18

```
type Animal interface {}

kalle := Dog{"Kalle"}
felix := Cat{"Felix"}

animals := [2]Animal{kalle, felix}
for _, v := range animals {
  fmt.Printf("Hey, %s\n", v.Talk())
}
```

Listing 19

```
func foo(fn func(x int) int) int {
  return fn(2)
}

func main() {
  f := func(x int) int {
     return x * x
  }
  foo(f)
}
```

Listing 20

„case"-Blocks eingesetzt werden, um die nachfolgenden „case"-Blöcke in die weitere Auswertung mit einzubeziehen. Eine weitere Verwendungsart von „switch" ist die Ersetzung der Variablen im „switch"-Teil durch einzelne Bedingungen in den „case"-Zweigen *(siehe Listing 15)*. Diese Variante wird häufig als lesbare Version hintereinandergereihter „if"-Statements verwendet.

Das Werkzeug Go

Das zentrale Go-Werkzeug heißt „go". Einmal installiert, lassen sich alle wichtigen Aufgaben auf der Kommandozeile ausführen. *Listing 16* zeigt einige Beispiele. Go ist eine Multi-Paradigmen-Programmiersprache, die sowohl Konzepte der objektorientierten als auch der funktionalen Programmierung unterstützt.

Objektorientierte Programmierung in Go

Objektorientierung ist durch drei Eigenschaften gekennzeichnet: Kapselung, Vererbung und Polymorphismus. Kapselung wird in Go durch Packages und das Konzept der Groß- (außerhalb sichtbar) und Kleinschreibung (Package-intern) realisiert. Statt auf Vererbung setzt Go auf Komposition. Im Beispiel in *Listing 17* komponiert der Typ „Car" eine Reihe von Bauteilen, aus denen ein Auto besteht.

Attribute und Methoden enthaltener Typen können direkt auf Variablen des komponierenden Typs aufgerufen werden. So lässt sich eine auf „Engine" implementierte Methode „start" direkt auf einer „Car"-Instanz aufrufen *(siehe Listing 18)*. Die Modellierung basiert auf Komposition, während die Benutzung komponierter Typen der einer vererbten Typ-Hierarchie entspricht.

Polymorphismus

Polymorphismus bezeichnet die Eigenschaften, zur Laufzeit andere Formen annehmen zu können. Erbt beispielsweise in Java die Klasse B von A, dann dürfen Objekte der Klasse A zur Laufzeit auf Objekte der Klasse B zeigen. Die Variable vom Typ A nimmt zur Laufzeit die Form B an. In Go existiert kein Vererbungs-basierter, dafür aber ein Interface-basierter Polymorphismus *(siehe Listing 19)*.

Im Beispiel wird das Interface „Animal" von den beiden Typen „Dog" und „Cat" implementiert. Entsprechend ist es möglich, ein Array vom Typ „Animal" mit Instanzen unterschiedlicher Typen zu initialisieren, sofern sie dieses Interface implementieren.

Funktionale Programmierung

Die wesentlichen Eigenschaften funktionaler Programmiersprachen sind Immutability, Seiteneffekt-Freiheit sowie die Behandlung von Funktionen als „First Class"-Objekte. Die Eigenschaften Immutability und Seiteneffekt-Freiheit haben wir bereits im Abschnitt über Pointer kurz erörtert, sodass wir in diesem Abschnitt nur noch auf den Umgang mit Funktionen eingehen.

Go-Funktionen sind „First Class"-Objekte, sie können also überall dort eingesetzt werden, wo sonst normale Variable genutzt werden. Das Beispiel in *Listing 20* weist einer Funktion der Variablen „f" zu und übergibt diese Variable und damit die Funktion einer anderen Funktion „foo". Zusätzlich lassen sich eigene Funktionstypen deklarieren *(siehe Listing 21)*.

Die Beispiele dieses Abschnitts zeigen, dass Go weder rein objektorientiert noch rein funktional ist. Dennoch lassen sich Konzepte aus beiden Welten in lesbarer Form umsetzen.

Warum Go?

Go ist eine produktive und leicht zu erlernende Programmiersprache. Erfahrene Java-Programmierer sollten Go innerhalb einer Woche lernen und produktiv einsetzen können. Standards und Konvention sorgen dafür, dass Go-Code sehr gut lesbar ist. Die reduzierte Syntax führt zudem dazu, das Programmierer dieselben Sprachkonstrukte nutzen und sich nicht gemäß ihrer persönlichen Präferenz für das ihnen liebste Sprachkonstrukt entscheiden. So gibt es in Go nur eine einzige Schleife, die „for"-Schleife, oder nur eine gültige Form der Klammersetzung in „if"-Blöcken. Go-Code ist immer gleich formatiert, ungenutzte Imports oder Variablen und Funktion sind Fehler.

Go ist eine typsichere Programmiersprache – eine Eigenschaft, die viele Programmierer aufgrund der deutlich besseren Refaktorisierbarkeit typisierter Sprachen schätzen. Das Go-Typ-System ist dabei äußerst leichtgewichtig und steht dem Programmierer nicht im Weg. Typ-Inferenz sorgt dafür, dass Programmierer den Typ nur dann angeben müssen, wenn der Compiler den Typ nicht automatisch erkennen kann. So wird der Typ der beiden Zielvariablen in *Listing 22* automatisch erkannt.

Neben Produktivität und statischer Typisierung ist Performance ein weiterer wichtiger Grund für den Einsatz von Go. Go-Programme kompilieren zu Assembler und sind in ihrer Ausführungsgeschwindigkeit vergleichbar mit C. Die Go-Standard-Bibliothek enthält derzeit an die 160 Pakete, die für viele typische Programmierprobleme Standardlösungen bieten. Ein gutes Beispiel sind die Bibliotheken „net/http" und „encoding/json", die alles für die Programmierung von REST-APIs enthalten.

```
type myfunction func(x int) int

func foo(fn myfunction) int { ... }
```

Listing 21

```
func swap(x int, y int) (int, int) {
  return y, x
}

i, j := swap(1, 2)
```

Listing 22

```
if file, err := os.Open(file); err == nil {
}
```

Listing 23

Go-Programme lassen sich für unterschiedliche Zielplattformen cross-kompilieren. Ergebnis ist ein statisch gelinktes Binary, das per einfaches Secure-Copy auf die Zielmaschine kopiert wird. Dort müs-

sen weder VMs installiert noch Bibliotheken, wie Jars oder RubyGems, in einer bestimmten Version existieren. Das Programm kann direkt ausgeführt werden. Deployments werden so zum Kinderspiel.

Was nervt?

Genau wie in jeder anderen Programmiersprache gibt es in Go eine Reihe von Kritikpunkten, die im Rahmen einer Gesamtbetrachtung nicht unter den Tisch fallen dürfen. So kennt Go keine Exceptions. Stattdessen geben Funktionen, bei deren Ausführung es zu Fehlern kommen kann, als letzten Rückgabewert eine Instanz vom Typ „error" zurück. Entsprechend ist die Zuweisung von Funktionsergebnissen bei gleichzeitiger Überprüfung des Fehlerwerts ein häufig anzutreffendes Go-Idiom *(siehe Listing 23)*.

Der Verwendung von Fehler-Rückgabewerten ist grundsätzlich keine schlechte Idee, zumindest dann nicht, wenn die Programmiersprache mehrere Rückgabewerte erlaubt. Fehler-Rückgaben sind allerdings sinnlos, wenn sie nicht direkt im Anschluss an den Funktionsaufruf geprüft werden. Das bringt uns zum eigentlichen Minuspunkt des Go-Fehlerbehandlungskonzepts: Einbußen bei der Lesbarkeit. Bei mehreren hintereinander folgenden Funktionsaufrufen mit potenziell fehlerhaftem Ausgang wird aus eigentlich gut lesbarem Code eine „if"-Kaskade, die deutlich schlechter zu lesen ist.

Andererseits lässt sich auch argumentieren, dass die Fehlerbehandlung in Go eng an die Fehler-verursachende Funktion heranrückt. So wird erstens schnell erkannt, ob Fehler behandelt werden, und zweites wird klar, welche Funktion welche Art von Fehler liefert.

Fehlende Unterstützung von Generics ist einer der am häufigsten geäußerten Kritikpunkte. Go verfügt zwar mit Arrays und Maps über zwei generische Container-Typen, erlaubt jedoch nicht die Implementierung eigener generischer Typen.

Go ist eine Sprache für die Entwicklung von Tools und Services, die schnell und robust sein müssen, dabei aber eher im Hintergrund oder auf der Kommandozeile arbeiten. Beispiele sind Webservices, DevOps-Tools oder auch die Dropbox-Bandbreitenregulierung. Go enthält gute Bibliotheken für die Kommandozeilen-Programmierung, aber keine Bibliothek für die Entwicklung grafischer Benutzeroberflächen. Neben dem Wunsch nach Generics sind GUI-Bibliotheken eine weitere häufig geäußerte Forderung an zukünftige Go-Versionen.

Einfaches, aber unzureichendes Dependency-Management

Das Go-Dependency-Management-Konzept scheint auf den ersten Blick charmant einfach. Statt der Verwendung eines zusätzlichen Tools mit komplizierter XML-Beschreibungssprache wird das benötigte Package einfach importiert und die zugehörige Bibliothek via „go get" in den Workspace geladen.

Spätestens in größeren Projekten und bei mehreren Projekten im selben Workspace zeigt sich jedoch schnell ein Problem. Angenommen, der Workspace enthält zwei Projekte, die beide dieselbe Bibliothek referenzieren, diese aber in unterschiedlichen Versionen benötigen. Oder man hat eine Bibliothek entwickelt, die eine andere Bibliothek in einer ganz bestimmten Version benötigt. Wie stellt man sicher, dass die Benutzer ihrer Bibliothek die benötigte Bibliothek genau in der erwarteten Version in ihrem Workspace haben?

Die hauseigenen Go-Werkzeuge bieten bisher keine Lösungen für die genannten Probleme. Allerdings hat die Community hier schon einen Schritt vorgelegt und mit „godep" ein Werkzeug entwickelt, um das Management von Abhängigkeit in bestimmten Versionen zu ermöglichen.

Fazit

Go besticht durch Einfachheit. Sowohl Sprache als auch Tools sind einfach erlern-und benutzbar. Standards und Konventionen erzwingen gut lesbaren Code, was die Sprache zu einer produktiven Sprache für Teams macht.

Der Go-Sprachumfang ist stark reduziert und besteht aus nur fünfundzwanzig Schlüsselwörtern. Im Vergleich dazu: Java hat fünfzig und Swift sogar achtundfünfzig Schlüsselwörter. Dennoch ist die Sprache sehr ausdrucksstark und unterstützt sowohl objektorientierte als auch funktionale Sprachkonzepte.

Eine im März 2017 veröffentlichte Umfrage hat ergeben, dass die Mehrheit der Entwickler Go für die Entwicklung von Webservices einsetzt. Die Standard-Library liefert bereits so gute HTTP-/JSON-Unterstützung, dass der Einsatz eines zusätzlichen Web-Frameworks entfallen kann. Die entwickelten Services sind schnell und robust und lassen sich durch einfaches Kopieren auf die Zielmaschine deployen.

Go-Entwickler behaupten: „Simplicity is addictive". Einfachheit und Produktivität machen Spaß. Und genau das ist der Punkt: Go-Programmierer sind zufriedene Entwickler, die ihre Werkzeuge lieben, Ergebnisse liefern und Unternehmen erfolgreich machen.

Referenzen und Links

- *https://golang.org*
- *https://tour.golang.org*
- *https://gobyexample.com*

Ralf Wirdemann
ralf.wirdemann@kommitment.biz

Ralf Wirdemann ist Agile Developer Coach und hilft Teams, sowohl technisch als auch methodisch auf Spur zu kommen. Guter Sourcecode ist ihm genauso wichtig wie gute User Stories, ein fürs Team passender Entwicklungsprozess oder eine funktionierende Deployment-Pipeline.