Go: Examining Google's hottest new open source project

TOBIAS ERIKSSON

Doktor Abrahams Väg 55
168 59 Bromma
070-7426812

tobier@kth.se

DD143X Degree Project in Computer Science, First Level
KTH School of Computer Science and Communication
Supervisor: Henrik Eriksson

Examiner: Mads Dam

June 12, 2011

Abstract

The Go programming language is a new open source project from Google, that aims to make programmers more productive. It claims to be a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language. It is being hyped to the extent that it is difficult to find an unbiased opinion about the language. The purpose of this essay is to evaluate Go to see if it makes a newcomer a productive programmer, mainly focusing on ease-of-use, development time and restrictions or disadvantages.
The origins and design decisions of the language are presented, and is followed by an introduction to the syntax and features of Go. The language is evaluated by programming a small web application, and analysing the work done. The closing discussion concludes that Go is a powerful language that can improve programmers productivity.

Referat

Go: Undersökning av Googles hetaste nya open source-project

Programmeringsspråket Go är ett nytt open source projekt från Google, som har målet att göra programmerare mer produktiva. Go påstås vara ett snabbt, statiskt typat, kompilerat språk som känns som ett dynamiskt typat, interpreterat språk. Det har blivit hypat till sådan utsträckning att det har blivit svårt att hitta objektiva åsikter om språket. Målet med denna uppsats är att utvärdera Go, för att se om en nykomling kan bli en mer produktiv programmerare, med fokus på lättanvändhet, nerlagd utvecklingstid och restriktioner eller nackdelar.
Bakgrunden till språket blir presenterat, och följs av en introduktion till syntax och funktionalitet. Språket blir utvärderat genom programmering av en enkel webbapplikation, och analys av det utförda arbetet. Den avslutande diskussionen drar slutsatsen att Go är ett kraftfullt språk, som kan utöka en programmerares produktivitet.

Contents

Chapter 1

Introduction

The purpose of this study is to evaluate Go, Google's hot new programming language. Go is designed to be easy to use with good support for concurrency and communication, distributed large scale programming, high speed compilaton and efficient latency-free garbage collecting. Go aims to have the efficiency of a statically-typed compiled language with the ease of programming of a dynamic language[1].

1.1 Problem statement

Ever since its revelation in late 2009, Go has been hyped to the extent that the only way to get an unbiased opinion is to evaluate it oneself. This will be done by documenting the language, programming a small one-man project, and analyzing the work and results. There are three (3) main points that is to be taken into consideration:

- How easy is Go to use, for a newcomer to the language?
- Does the claimed ease of use and language features cut down on development time?
- What are the limitations and disadvantages of Go?

The goal of the programming project is to gain experience with the language, so that the above questions can be answered. In other words, the product of this study is not the software, but the analysis of the experience gained.

1.2 Programming project

The purpose of the Wiki is to gain experience in the language, and to support the analysis. The experiences will be documented, development time for each component will be noted. The product of this study is not the software, but the analysis of the experience gained. The project consist of programming a simple Wiki, using the various features and packages that Go provides. Owning to the time allocated to writing this essay, the Wiki will be limited in scope when compared to other projects such as MediaWiki. The Wiki is supposed to be a tool in the evaluation of the language, not a full-fledged product.

1.3 Content summary

Chapter 1 summarizes the goals of this essay; it presents the main questions the author wants to answer and touches briefly on the implementation.

Chapter 2 gives an introduction to Go; the most important and interesting features are discussed in a brief and relaxed way, to get the reader familiar enough with Go to follow along in the implementation.

Chapter 3 focuses on the implementation of the programming project; it shows the most important aspects of the implementation and how Go can be used in developing a *real* application, in contrast to chapter 2.

Chapter 4 analyses the questions stated in chapter 1 while using the experience gained in chapter 3 to back up the arguments.

Chapter 5 concludes the essay by summarizing the results and answering the questions stated in chapter 1.

1.4 Literature summary

The literature that is used as reference in this essay is material produced by the Go authors; tutorials, articles and the language specification. These are extremely useful when learning how the language works. They make good references when presenting the language objectively, especially the language specification.

Google Tech Talk: The Go Programming Language, by Rob Pike, was the first time Go was presented publically, and it gives an overview of the history and features of the language. This is good introductory literature to Go.

A Tutorial for the Go Programming Language, and *Effective Go* (both credited to the Go authors), are articles useful for learning to program the language, and have been heavily used in chapter 2 as references. These have both been extensively studied.

The Go Programming Language Specification is by far the most unbiased and useful document used a reference: it is a heavily technical document that describes how Go works in detail.

Chapter 2

Background

In order to follow along in later chapters, the reader needs to get familiar with Go. This chapter will present some of the history and design principles of the language, and then proceed to describe the features.

2.1 Why A New Language?

On October 30th in 2009, Rob Pike gave the first presentation on the Go Programming Language[1]. The project was started in 2007 by Pike, Ken Thompson and Robert Griesemer. The goals were to develop a new programming language that has the efficiency of a statically-typed compiled language with the ease of programming of a dynamic language. The language had to be type- and memory-safe, with good support for concurrency and communication, with efficient and latency-free garbage collection, and with high-speed compilation.

Pike points out that our world is a changing one. There has not really been a new major systems language in a decade, but much has changed in that time[1]:

- sprawling libraries and dependency chains
- dominance of networking
- client/server focus
- massive clusters
- the rise of multi-core CPUs

He argues that the major systems languages were not designed with all these factors in mind. Not only that, it takes too long to build software; the tools are slow and getting slower, yet the software grows and grows[1].

Robert Griesemer says that "Clumsy type systems drive people to dynamically typed languages.". Pike argues the type systems today are clunky: it taints a good idea with bad implementation. He also argues that types in large programs do not easily fall into hierarchies, and that programmers spend too much time deciding tree structure and rearranging inheritance[1].

Pike says we need to start over; adding anything to existing languages is going in the wrong direction.

2.2 Design principles

In the Tech Talk on October 30th in 2009, Pike showed some design principles of Go:

Keep concepts orthogonal A few orthogonal features work better than a lot of overlapping ones.

Keep the grammar regular and simple Few keywords, parsable without a symbol table.

Reduce typing. Let the language work things out No stuttering; don't want to see **foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT)**. Avoid bookkeeping, but keep things safe.

Reduce typing. Keep the type system clear No type hierarchy. Too clumsy to write code by constructing type hierarchies; it can still be object-oriented.

It is noticeable that the language is designed from the beginning to make life easier for programmers.

2.3 Features

2.3.1 Basic concepts

Let us begin looking at the language with a simple example:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   fmt.Printf("Hello, world!\n")
7 }
```

Code Example 2.1: Hello world!

Go code is similar to C, with some differences. Let us take a brief moment and study the above examle. First, we declare which package we are currently using. All Go source-files are a part of a package, and every executable Go program is required to have a package called main. Next, we import the external fmt package (for formatted I/O). Last, we declare a function called main (also needed by all executable Go programs). In it, we call Printf, a function from the fmt package that prints our message.

You may have noticed the lack of semicolons in our little program. In Go code, the only place you typically see semicolons is separating the clauses of for loops and the like; they are not necessary after every statement. Formally the language uses semicolons, much as in C or Java, but they are inserted automatically by the lexer at the end of every line that looks like the end of a statement[2].

Variables and constants

Variables in Go is much like variables in other languages; they are identifiers with a type and a value.[1] The *static type* of a variable is defined by its declaration.[2] If the type is present in the decleration, the variable is given that type. Otherwise, the types are deduced from the assignment[3].

[1] Uninitialized variables have a *zero value*, decided by its type. See [3]

[2] Variables of interface type also have a distinct *dynamic type*, which is the actual type of the value stored in the variable at run-time.

```
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
  i int
  u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interested in "found"
```

Code Example 2.2: A sample of variable declarations

There is also a shorthand for the regular variable declarations, and this is often used in Go. However, short variable declarations may appear only inside functions[3].

```
i, j := 0, 10 // shorthand for declaring two integers
r, w := os.Pipe(fd)  // os.Pipe() returns two values
_, y, _ := coord(p)  // coord() returns three values;
             // only interested in y coordinate
```

Code Example 2.3: A sample of shorthand variable declarations

Control structures

There are three control structures in Go: the if-else statement, the switch statement, and the for loop. There are no do or while loops in Go. The control structures are related to those in C, but differ in important ways. A simple if looks like this (notice the lack of parentheses):

```
if x > 0 {
    return y
}
```

Code Example 2.4: A simple If statement

Braces are mandatory: it encourages to write simple if statements on multiple lines. Both if and switch accept initialization declarations; it is common to see one used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

Code Example 2.5: If statement with shorthand declarations

The accompaning else statements works in the obvious way; examples are omitted.

The Go for loop unifies for and while. There are three forms, only one of which has semicolons.

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

Code Example 2.6: The three forms of the for loop

Short declarations make it easy to declare the index variable right in the loop, so you often see something like[4]:

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

Code Example 2.7: For loop with shorthand declaration of the loop variable

Go's switch is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the switch has no expression it switches on true. It is therefore possible to write an if-else-if-else chain as a switch.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

Code Example 2.8: A switch statement

Functions

Functions are introduced with the func keyword (we have just seen this in the above code example). A function's visibility outside its package is determined by the first character of its name: an uppercase character makes the function visibile, while lowercase does not (similar to public and private functions in other languages). A visible function can be executed outside of its package like package.MyFunction().

Functions can have *multiple return values*. It is usually used to improve on a couple of clumsy idioms in C-programs[4]: in-band error returns (such as -1 for EOF) and modifying an argument. Say you have a function that writes to a file. With two return values, you can return a count and an error, like: "You have written some bytes, but not all of them because the device is full". It makes for simpler code that is also self-documenting.

The return parameters of a Go function can be given names and used as regular variables, just as the function paramaters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a return statement with no arguments, the current values of the result parameters are used as the returned values.

```
func ReadFull(r Reader, buf []byte) (n int, err os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

Code Example 2.9: A function from the I/O library illustrating above concepts

In Go, you can schedule a function call to be run immediately after the calling function returns. This is done with the keyword defer. A canonical example is closing a file[4].

```
1 func Foo(filename string) (string, os.Error) {
2     f, err := os.Open(filename, os.O_RDONLY, 0)
3     if err != nil {
4         return "", err
5     }
6     defer f.Close()  // f.Close will run when we're finished.
7     ... // code omitted
8     return string(result), nil // f will be closed if we return here.
9 }
```

Code Example 2.10: Canonical use of deffered functions

2.3.2 Types and allocation

Go has some familiar types that you would expect from a programming language, such as integers of appropriate size (signed and unsigned), bytes, floating-point types etc. There are arrays, structures, pointers and more.

Most types in Go are values: assignments copies variables of value types. In contrast to C, arrays are values and not pointers. In Go, it is more common to use *slices* insted of arrays; they are reference types that are more flexible: they can vary in length during execution[4].

Pointers are present in Go. With them comes one of the allocation primitives: new() (the other one being make()). You can dynamically allocate storage for a type T with new(T) and save the address in a pointer, like you would in C. However, new() does not apply to all types: reference types such as slices, maps and channels are allocated with make().

There is much that can be said about types and allocation: see [3] for details.

2.3.3 Methods

Methods in Go are not defined within a type structure, like you would define a method in a class in Java. Rather, they are functions with an explicit receiver, placed within parentheses before the function name. The following example illustrates how this is done:

```
1 func (foo *Foo) MyMethod(n int) int {
2   return foo.someMember += n
3 }
```

Code Example 2.11: Example of a method

There is no implicit this and the receiver variable must be used to access members of the structure.

2.3.4 Interfaces

Interfaces in Go provide a way to specify the behavior of an object: if something can do *this*, then it can be used *here*[4].

```
1 type Zoobar interface {
2   Foo(b []byte) (ret int, err os.Error)
3   Bar() string
4 }
```

Code Example 2.12: A simple interface

A type that implements the Foo and Bar methods can be used as a Zoobar type. A type can implement more than one interface by having the required methods.

2.3.5 Concurrency

One of the most interesting features in Go is concurrency. Concurrent programming is a large topic; we will only look into some Go-specific highlights here.

Concurrent programming is made difficult in many environments by the subtleties required to implement correct access to shared memory. Go takes a different approach: shared values are passed around on channels, and is never actively shared by seperate threads of execution. By design, only one thread has access to the value at any given time, and thus data races cannot occur[4].

Goroutines

In Go, threads are known as *goroutines*. Goroutines have a simple model: they are functions executed in parallel in the same adress space along with other goroutines. They are lightweight, costing little more than the allocation of stack space (and the stack starts small, growing by allocating heap storage as needed)[4]. Goroutines are designed to hide the complexity of thread creation and management to the programmer. To run a new goroutine, prefix a function or method call with the go keyword. When the call completes, the goroutine exits silently.

Channels

Channels are a reference type in Go, that provides communcation and synchronization. Channels can be unbuffered (synchronous), or buffered[4]. Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value[4].

```
1 c := make(chan int)  // Allocate a channel.
2 // Sort in a goroutine; when it completes, signal on the channel.
3 go func() {
4     list.Sort()
5     c <- 1  // Send a signal; value does not matter.
6 }()
7 doSomethingForAWhile()
8 <-c   // Wait for sort to finish; discard sent value.
```

Code Example 2.13: Simple use of a goroutine and channel

Chapter 3

Method

The purpose of this essay is to evaluate Go. This not only requires studying the language (as done in chapter 2), but also doing actual programming. This chapter will focus on the implementation of a simple web application, a wiki, written in Go. During development, the time spent working on a specific module will be recorded, and along with the experience gained will be used when evaluating Go in the remaining chapters.

3.1 Scope

A wiki can potentially be a large application; considering the time constraints and to ease the evaluation, our wiki will be limited in scope to the most basic features: adding and editing pages, and very simple markup language support.

3.2 Overview

A *wiki* is a website that allows the creation and editing of any number of interlinked web pages. They are typically powered by *wiki software* that can be written in a number of different programming languages and is often run on a *web server*. They are commonly used to create collaborative works (such as an encyclopedia[1]). Editing content is usually done with a simple markup language or a WYSIWYG[2] editor.

As the Go library provides the `http` package, there is no need for a third party web server to host the wiki software. The wiki itself handles requests from clients, and stores its data in a document-oriented database. The database software that is used is MongoDB[3], because structure of such document-oriented databases are simple and they are easy to use[4]. Table 3.1 gives an overview of the *packages* that make up the wiki software.

[1] Wikipedia ((`http://www.wikipedia.org`) is a popular example of a encyclopedia powered by wiki software
[2] Short for 'What-You-See-Is-What-You-Get'
[3] `http://www.mongodb.org`
[4] Another reason is that MongoDB is different; the author enjoys experimenting with unfamiliar software

Name	Description
main	The main responsibility of this packageis to serve clients. It configures the various handlers and then listens for incoming client requests.
handlers	Client requests are given to handlers, functions that exist to handle a specific type of request. Such request could be to view a wiki page.
page	This package contains the structures and functions that deal with wiki pages, such as creating a new instance of a page.
database	This package provides an interface to the MongoDB database with easy-to-use functions for specific purposes, such as querying the databse for a specifc wiki page. It relies on a third party package, gomongo, that gives an API to the database.

Table 3.1: Overview of the packages in the wiki software

3.3 Implementation

3.3.1 The main package

The main package is the smallest and simplest package. Thanks to the extensive packages bundled with Go, main can implement a fully functional web server without much of an effort.

```
1  package main
2
3  import (
4    "http"
5    "fmt"
6    "./database"
7    "./handlers"
8  )
9
10 func main() {
11   err := database.ConnectToDB("127.0.0.1", "mongowiki")
12   if err != nil {
13     fmt.Printf("Error on database connection: %s\n", err)
14     return
15   }
16   fmt.Printf("Connected to database!\n")
17
18   http.HandleFunc("/", handlers.FrontPageHandler)
19   http.HandleFunc("/edit/", handlers.EditHandler)
20   http.HandleFunc("/view/", handlers.ViewHandler)
21   http.HandleFunc("/save/", handlers.SaveHandler)
22   http.HandleFunc("/inc/", handlers.IncludeHandler)
23   http.ListenAndServe(":8080", nil)
24 }
```

Implementation 3.1: The entire main package.

The content of the wiki is stored in a MongoDB database, to which the main function tries to connect at startup. The real logic behind the wiki application are the different *handlers*. The main function associates these handlers with different URLs. It then spends the rest of its life accepting and serving client requests.

3.3.2 The handlers package

A client that wants to execute a functionality in the wiki communicates with the handlers. These are functions that are called from the main function whenever a client goes to a URL in the wiki, for example `/view/some_article`. The five URLs that have handlers associated to them are: `/`, `/view`, `/edit`, `/save` and `/inc`.

The / handler

The simplest handler by far is the handler for the root path. Its sole functionality is to redirect a client to the front page. Any path that is below the root in the URL hierarchy needs a handler, or the root handler will be executed. This is good thing: if the client enters an incorrent path that does not have a handler, he or she is redirected to the front page rather than crashing the wiki or executing something malicious.

The `/view` handler

The `/view` handler loads the page that is requested, or redirects the client to the page editor if the page is not present in the database. It does this by stripping the `/view/` part of the URL, and tries to load the page identified by the resulting *slug*.

```
1 func ViewHandler(w http.ResponseWriter, r *http.Request){
2   slug := r.URL.Path[6:]
3   p, err := page.LoadPage(slug)
4   if err != nil {
5     http.Redirect(w, r, "/edit/"+slug, http.StatusFound)
6     return
7   }
8   p.RenderTemplate(w, "view")
9 }
```

Implementation 3.2: The /view handler.

So, if the client requests a page at the URL `/view/my_article`, the handler will try to load the page identified by `my_article`. If it fails, the client will be sent to `/edit/my_article` so that he or she can create the page. If it did exist, the page would render itself in HTML and presented to the client.

The `/edit` handler

The `/edit` is very similar to the `/view` handler: it tries to load the requested page from the database, and renders the edit page to the user in HTML. If it does not find the page in the database, it will create a new temporary "New Page" so that the user has the chance to create this page from scratch.

The `/save` handler

Saving a page occurs when a client edits a page, and clicks the *save* button. The client is sent to the url `/save/title_of_the_edited_article`, and the `/save` handler tries to save this page. It does so by reading the form values of the edit page (carried over by the request object), and tries to create a Page object. Assuming this went well, the page is saved to the database. Actually, there is more to this. The contents of a page body may contain HTML, potentially opening up a security hole. This is fixed by using a regular expression to replace anything that looks like a HTML-tag, with an empty string.

```
// ...
//remove any scary HTML in the body!
bodyFixer := regexp.MustCompile("<[a-zA-Z/][^>]*>")
fixedBody := bodyFixer.ReplaceAll([]byte(body), []byte(""))
// ...
```

Implementation 3.3: Using the regexp package to remove HTML-tags.

The /inc handler

Due to the fact that the /-handler will handle any requests that do not have a handler implemented, including files such as CSS stylesheets in a HTML file will not work. The wiki stores all "include" files in the folder /inc, so it is neccessary to implement a handler for that URL.

```
func IncludeHandler(w http.ResponseWriter, r *http.Request){
  f, err := os.Open(r.URL.Path[1:], os.O_RDONLY, 0644)
  defer f.Close()
  if err != nil {
    http.NotFound(w, r)
    return
  }
  b := new(bytes.Buffer)
  b.ReadFrom(f)
  fmt.Fprintf(w, b.String())
}
```

Implementation 3.4: The /inc handler.

The handler tries to open the file requested relative from the wiki executable. If this succeeds, the contents of file will copied in a byte buffer, so that it can be written back to the http.ResponeWriter. The textual contents of the file will then be presented to the requester, possibly a client that wants to look at a file in /inc, but more likely the wiki itself wanting to render a template that includes a seperate stylesheet file.

3.3.3 The page package

The page is the fundamental data structure of the wiki: the whole wiki is made up of interlinked pages. Pages exists both in memory and in the database, however only temporarily for the former. It would probably be more efficient to have all the pages in memory for quick access, and only read/write the database for updates. However, the goal of the programming project is not to make a perfect wiki; it is to program in Go.

A page in memory is a simple struct, consisting of 3 fields:

```
type Page struct {
  Slug string
  Title string
  Body string
}
```

Implementation 3.5: The Page struct.

The page is stored in the database as a JSON[5]-like object, with the same structure (see figure 3.1). The database package converts between the page in memory, and the page in the database (see section 3.3.4).

[5]JSON (an acronym for JavaScript Object Notation) is a lightweight text-based open standard designed for human-readable data interchange.

page	
slug	string
title	string
body	string

Figure 3.1: Page schema.

Go does not have constructors; instead it is common to use factories for easy creation of objects.

```
1  // Create a page, validate the title while doing it.
2  func CreatePage(title string, body string) (*Page, os.Error) {
3    if !titleCreationValidator.MatchString(title) { // is it an OK title?
4      return nil, os.NewError("Title contains invalid characters.")
5    }
6    return &Page{generateSlug(title), title, body}, nil
7  }
```

Implementation 3.6: The factory for creating a page.

Loading and saving pages is simple. Loading a page is done by querying the database (through the database package) for the page slug, and creating a page object from the fields returned. Saving a page from memory into the database is done by giving the database the fields of the struct, again by using the database package. The code itself is not that interesting, and is omitted.

Go is bundled with a package called template, that is used to render HTML documents from objects. The wiki uses it to render its pages, with the RenderTemplate method.

```
1  // Render the page as HTML.
2  func (p *Page) RenderTemplate(w http.ResponseWriter, tmpl string) {
3    t, ferr := template.ParseFile("inc/"+tmpl+".html", nil)
4    if ferr != nil {
5      panic(os.NewError("template not found, panic!!"))
6    }
7    pr, err := p.RenderMarkup(tmpl)
8    if err != nil {
9      t.Execute(w, p)
10     return
11   }
12   t.Execute(w, pr)
13   return
14 }
```

Implementation 3.7: Rendering a page as HTML.

The method is generic and is used to render both the `/view` and `/edit` pages. The programmer only needs to create a HTML-file with template specific pseudo-code, and the ParseFile together Execute will output the finished HTML. The HTML itself is not really complicated.

```
1  <html>
2    <head>
3      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
4      <title>MongoWiki - {Title}</title>
5      <link rel="stylesheet" href="/inc/style.css" />
6    </head>
7    <body>
8      <div style="float:left; margin: 5px auto;">
```

```
9       <span class="header">[[ <a class="title" href="/view/{Slug}">{Title}</a> ]]
            </span>
10      <a href="/edit/{Slug}">edit this page</a>
11    </div>
12    <div id="content">
13    {Body}
14    </div>
15    Powered by MongoWiki!
16  </body>
17 </html>
```

Implementation 3.8: The code for view.html.

The parts between the curly braces is what is parsed by the template package. Here they correspond to the fields of the page structure and is replaced with their contents when rendering the HTML.

3.3.4 The database package

The database package provides function to set up a database connection, to save a page, and to load a page. Let us look a bit at how a page is saved.

```
1  // Save a page to the database
2  func SavePage(pageDoc map[string]string) os.Error {
3    coll := db.GetCollection("pages") // get the 'pages' collection from the db
4    defer coll.Drop() // remember to drop the collection before returning
5
6    idBSON, idError := mongo.Marshal(map[string]string{ "slug" : pageDoc["slug"],
         }) // used to search for the page
7    pageBSON, pageDocErr := mongo.Marshal(pageDoc)   // create a BSON document
8
9    // omitted error handling ...
10   _, findErr := coll.FindOne(idBSON) // check if the page is already in the
         database
11   if findErr != nil {
12     if strings.Contains(findErr.String(), "cursor failure") { // this means that
           nothing could be found
13       coll.Insert(pageBSON)                              // insert it
14       return nil
15     }
16     return findErr
17   }
18   updateErr := coll.Update(idBSON, pageBSON)
19   // omitted error handling ...
20 }
```

Implementation 3.9: Implementation of saving a page to the database (some parts omitted).

The saving of a page in the database is perhaps the must difficult code to read. Luckily, it makes use of the third party gomongo package to communicate with the database (seen in the implementation as the variable db). The input to the save function is an associative map, the keys being the public fields of the Page struct. Implementation 3.9 is a slighty modified version of the actual code, to make it a bit more readable with some parts omitted. What is basically done is that two BSON[6] objects are created: one for searching and one for inserting. If the page is already in the database, we update it instead of inserting it directly in the database.

[6]BSON is a computer data interchange format like JSON, but for binary data. MongoDB uses BSON because it's designed to be lightweight, traversable, and efficient[5].

If nothing could be found, we insert the latter BSON document into the database. Thanks to gomongo, this is done without much effort.

Chapter 4

Analysis

This chapter will focus on the analysis of the experience gained during the implementation of the wiki (as discussed in the previous chapter). More specifically, we will discuss the experiences in terms of the questions stated in the introductory chapter.

4.1 Ease-of-use

A major factor (in the author's opinion) in how easy a programming language is to use, is how much a programmer needs to type. Usually, less is better as long as the code is still humanly readable. Consider Figure 4.1, where we make a comparison of hypothetical Go and C++ code, for the sake of argument. The two lines do essentially the same thing: construct an object of

```
(C++)   foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT);
(Go)    myFoo := foo.CreateFoo(foo.FOO_INIT)
```

Figure 4.1: The same line of code written in C++ and Go respectively.

type foo.Foo, passing the value FOO_INIT as an argument. Which line is the most readable? The author is experienced in both C++ and Go, so neither line appears *unreadable*. But does the programmer really need the redundancy on the left-hand side in the C++ statement to understand the code? It is quite evident from looking at the right-hand side of the assignment (in both lines) what the resulting type will be (something a compiler should be able to do). Here, the author prefers the Go line. The two lines does not differ that much in length, but in a large applications (written in Go and C++ respectively) the overall difference will most likely be significant.

One interesting feature is multiple return-values. The author argues that multiple return-values is a huge bonus point to Go's ease-of-use (and also in other languages where this feature is present, such as Python and Ruby). With, say, C++ one has to return containers such as an std::pair or equivalent, which in comparison feels less attractive. Using containers not only makes the code more complicated, but is more to type (consider return std::make_pair(value_one, value_two) vs return value_one, value_two).

4.2 Development time

Part of the programming project was to record the time spent coding each separate package (see figure 4.2). Approximately 17 hours was spent programming the wiki, less than the author expected. This is quite impressive, considering no previous programming experience in Go (or

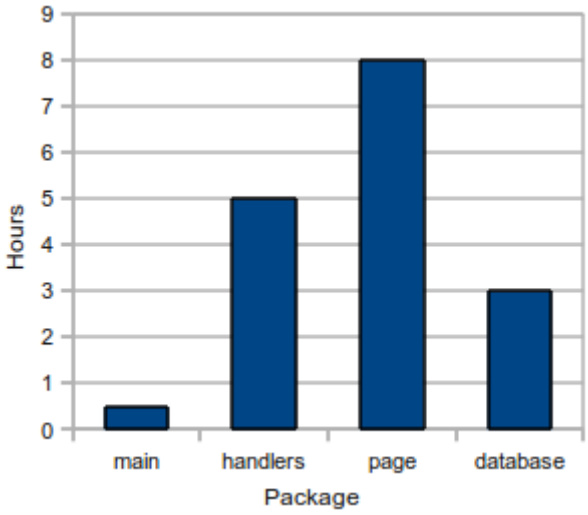

Figure 4.2: Development time (hours) for each package

programming a wiki for that matter). Also, these development times do not represent the time it took to write the final versions of the packages from scratch: they were iteratively developed as the functionality of the wiki grew.

The author expected writing the database package would dominate the development time, however it was dominated by writing the page package. This is most likely due to the fact that development started by writing a preliminary page package, just to get familiar with using Go. The use of the third party package gomongo, to communicate with MongoDB, probably cut down the development time of the database package considerably. The most difficult part of implementing the page package was the template rendering, which due to lack of examples in the documentation took maybe half of the 8 hours spent on the page package.

The handlers package was fairly straightforward to develop, but a couple of hours were "lost" due to the debugging problems related to the /inc handler (due to the fact that the author did not realize that it was needed when importing the CSS stylesheet in the template HTML).

The main package was trivial to implement, the actual to do the final versions could be measured in a couple of minutes. However, approximately half an hour was spent written the different versions that were use for debugging (each taking just minutes to implement).

As mentioned, the author expected more time to be spent implementing the wiki. The wiki includes only the most rudimentary features, but developing an application in a strange, new language could potentially take time. The fact that it did not is in the authors opinion thanks to the tutorial[2] and the *Effective Go*[4] article.

4.3 Restrictions and/or disadvantages

Lack of type hierarchy

The author is divided on whether the lack of a type hierarchy is a good thing or not. Most computer science students are taught to appreciate the hierarchical object-oriented paradigm; to have an object oriented language that lacks a strict type hierarchy is quite alien. The Go authors argue that programmers sometimes over-design their code, rather than getting real work done[1].

Unfortunately, developing the wiki did not present cases where having a class hierarchy would

be benificial, so it is difficult to make an argument backed-up by experience. However, we can reason about the implications.

Consider this example: We have two types of objects A and B. Let us assume we want A to be an abstract class, with B inheriting it. This can be achieved in Go, using interfaces. Define an interface A, and implement its methods for B, thus making B appear as a subclass of the abstract type A.

What if we introduce a third type C, and let us assume we want it to inherit B, how do we implement it in Go? We could take two approaches: extend A so that it declares all the methods that B implements (but then it would be like C inherits A), or define a new interface that declares the methods B implements not from A, and make sure C implement both interfaces.

What can we take from this? Even though we don't have a type hierarchy, it's possible to declare interfaces to that we basically implement the same behaviour.

Package documentation

One major disadvantage is the lack of extensive beginner-friendly documentation of the various packages. While the main tutorial[2] and *Effective Go*[4] gives a solid understanding of the language, the documentation of the packages are usually a bit difficult to understand. They are descriptive of the functional aspects, but lack solid examples[1].

Stability and backwards-compatibility

Another disadvantage is the fact that Go is a young language. Andrew Gerrard (of the Go Programming Language Blog) says:

> The Go project is moving fast. As we learn more about Go we are compelled to change our tools, libraries, and occasionally even the language itself. We permit backward-incompatible changes so that we can learn from, rather than immortalize, our mistakes. We believe flexibility at this stage of Go's development is essential to the project's development and, ultimately, its longevity.

These backward-incompatible changes (that fortunately the author has not run into) can and most likely will be a major problem if Go is used in a production environment. Go has recently moved to a new release schedule, Andrew Gerrard continues:

> Questions I hear often are "Is Go stable? How can I be sure that I won't have to update my Go code every week?" The answer to those questions are now "Yes," and "You won't."

Hopefully, Go will quickly move to a state where the programmer can be sure that his or her code doesn't often break because of backward-incompatible changes to the language[2].

Desktop programming

Go is not bundled with any real graphics- or GUI widget-packages (some might argue that Go is not meant for desktop programming). This is a restriction if the programmers wants to do desktop applications; however there most likely will be stable third-party package that will provide this soon[3].

[1] This may or may not be the case for the packages the author has not used. There are many packages, and the author has not come close to use them all.

[2] For the full blog post by Andrew Gerrard, see `http://blog.golang.org/2011/03/go-becomes-more-stable.html`

[3] A quick Google search found https://github.com/mattn/go-gtk, so there are third-party packages available for developing desktop applications.

Chapter 5

Conclusions

It is difficult to establish objectively if a programming language is easy to use or not. Let us consider what has been done in this essay: a typical programmer has taken on the task to learn a new programming language, with a background of Java and C/C++. This programmer (a.k.a the author) feels that Go is indeed learn and use, and it is likely that a typical programmer with the same background and experiences feel the same.

What does ease-of-use really mean? Consider what a programmer does with a programming language in everyday work: he or she deducts a solution to a problem, and translates that solution into code. It is important that the language does not get in the way of solving the problem: complex syntax and typing (as in writing code) gets in the way of getting work done. If we were to compare two languages with the same feature set, the authors would use the one with simplest syntax and least typing.

Not only need it be easy to use the language, but it has to be powerful enough so that development time does not suffer. Go is bundled with many packages that provide often used functionality such as regular expressions, locks, and more. This makes Go a powerful language: packages eliminate the need to implement commonly used functions, and also eliminates the time it would take to implement and debug these.

It is evident that there are restrictions and disadvantages to Go, as discussed in chapter 4. However, it's important to note one thing: Go is different. The Go authors themselves argue that programming like one would in Java and/or C++ does not produce good Go code[4]:

> Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

Using Go is definitely a refreshing experience for those familiar with C/C++ and Java. While not the language to rule them all, it is certainly a solid, usable systems language that the author envisions to be primarily deployed on server hardware rather than being used for desktop applications. It will be interesting to see if Go can make a breakthrough on the commercial market[1].

[1] According to the Go FAQ (`golang.org/doc/go_faq.html`), Go is in production use internally at Google.

In conclusion, it is evident that Go can make programmers productive, if they adapt to how Go is meant to be programmed, and not bring over idioms from other (different) programming languages.

Bibliography

[1] Rob Pike, 2009. *Tech Talk: The Go Programming Language*
`http://golang.org/doc/talks/go_talk-20091030.pdf`
Fetched 2011-01-26.

[2] The Go Authors, 2011. *A Tutorial for the Go Programming Language*
`http://golang.org/doc/go_tutorial.html`
Fetched 2011-02-24.

[3] The Go Authors, 2011 *The Go Programming Language Specification*
`http://golang.org/doc/go_spec.html`
Fetched 2011-02-22.

[4] The Go Authors, 2011. *Effective Go*
`http://golang.org/doc/effective_go.html`
Fetched 2011-02-24.

[5] The MongoDB Project. *BSON*
`http://www.mongodb.org/display/DOCS/BSON` Fetched 2011-06-03.

Appendix A

Source code

main.go

```
1  // Copyright 2011 Tobias Eriksson. All rights reserved.
2  // Use of this source code is governed by a BSD-style
3  // license that can be found in the COPYING file.
4
5  package main
6
7  import (
8    "http"
9    "fmt"
10   "./database"
11   "./handlers"
12 )
13
14 func main() {
15   err := database.ConnectToDB("127.0.0.1", "mongowiki")
16   if err != nil {
17     fmt.Printf("Error on database connection: %s\n", err)
18     return
19   }
20   fmt.Printf("Connected to database!\n")
21
22   http.HandleFunc("/", handlers.FrontPageHandler)
23   http.HandleFunc("/edit/", handlers.EditHandler)
24   http.HandleFunc("/view/", handlers.ViewHandler)
25   http.HandleFunc("/save/", handlers.SaveHandler)
26   http.HandleFunc("/inc/", handlers.IncludeHandler)
27   http.ListenAndServe(":8080", nil)
28
29 }
```

A.1: Implementation of the main package

handlers.go

```
1  // Copyright 2011 Tobias Eriksson. All rights reserved.
2  // Use of this source code is governed by a BSD-style
3  // license that can be found in the COPYING file.
4
5  package handlers
```

```
6
7  import (
8    "bytes"
9    "fmt"
10   "http"
11   "regexp"
12   "os"
13   "./page"
14 )
15
16 var ShowDebugMessages bool = false
17
18 func FrontPageHandler(w http.ResponseWriter, r *http.Request){
19   http.Redirect(w, r, "/view/front_page", http.StatusFound)
20 }
21
22 func ViewHandler(w http.ResponseWriter, r *http.Request){
23   slug := r.URL.Path[6:]
24   p, err := page.LoadPage(slug)
25   if err != nil {
26     http.Redirect(w, r, "/edit/"+slug, http.StatusFound)
27     return
28   }
29   p.RenderTemplate(w, "view")
30 }
31
32 func EditHandler(w http.ResponseWriter, r *http.Request) {
33   slug := r.URL.Path[6:]
34   p, err := page.LoadPage(slug)
35   if err != nil {
36     p,_ = page.CreatePage("New Page", "")
37     p.RenderTemplate(w, "edit")
38     return
39   }
40   p.RenderTemplate(w, "edit")
41 }
42
43 func SaveHandler(w http.ResponseWriter, r *http.Request) {
44   title := r.FormValue("title")
45   body := r.FormValue("body")
46
47   //remove any scary HTML in the body!
48   bodyFixer := regexp.MustCompile("<[a-zA-Z/][^>]*>")
49   fixedBody := bodyFixer.ReplaceAll([]byte(body), []byte(""))
50
51   p, err := page.CreatePage(title, string(fixedBody))
52   if err != nil {
53     http.Redirect(w, r, "/edit/"+r.URL.Path[6:] , http.StatusFound)
54     return
55   }
56   err = p.Save()
57   if err != nil {
58     http.Redirect(w, r, "/", http.StatusFound)
59     return
60   }
61   http.Redirect(w, r, "/view/"+p.Slug, http.StatusFound)
62 }
63
64 func IncludeHandler(w http.ResponseWriter, r *http.Request){
65   f, err := os.Open(r.URL.Path[1:], os.O_RDONLY, 0644)
66   if err != nil {
```

```
67      http.NotFound(w, r)
68      return
69    }
70    b := new(bytes.Buffer)
71          b.ReadFrom(f)
72    fmt.Fprintf(w, b.String())
73  }
```

A.2: Implementation of the handlers package

page.go

```
1   // Copyright 2011 Tobias Eriksson. All rights reserved.
2   // Use of this source code is governed by a BSD-style
3   // license that can be found in the COPYING file.
4
5   package page
6
7   import (
8     "http"
9     "os"
10    "regexp"
11    "strings"
12    "template"
13    "./database"
14  )
15
16  type Page struct {
17    Slug string
18    Title string
19    Body string
20  }
21
22  // for validation
23  var titleCreationValidator = regexp.MustCompile("^[a-zA-Z0-9 ]+$")
24  var loadPageValidator = regexp.MustCompile("^[a-z0-9_]+$")
25
26  // Create a page, validate the title while doing it.
27  func CreatePage(title string, body string) (*Page, os.Error) {
28    if !titleCreationValidator.MatchString(title) { // is it an OK title?
29      return nil, os.NewError("Title contains invalid characters.")
30    }
31    return &Page{generateSlug(title), title, body}, nil
32  }
33
34  // Generate a simple slug, the title should already be made up of alphanumerics
      only
35  func generateSlug(title string) string {
36    return strings.Replace(strings.ToLower(title), " ", "_", -1)
37  }
38
39  // Load a page
40  func LoadPage(slug string) (*Page, os.Error) {
41    if !loadPageValidator.MatchString(slug) { // is the slug OK?
42      return nil, os.NewError("invalid slug")
43    }
44    pageDoc, err := database.LoadPage(slug)
45    if err != nil {
46      return nil, err
```

```
47    }
48    return CreatePage(pageDoc.Title, pageDoc.Body)
49  }
50
51  // Save the page to the database
52  func (p *Page) Save() (os.Error) {
53    pageDoc := map[string]string{
54      "slug" : p.Slug,
55      "title" : p.Title,
56      "body" : p.Body,
57    }
58    return database.SavePage(pageDoc)
59  }
60
61  // Render the page as HTML.
62  func (p *Page) RenderTemplate(w http.ResponseWriter, tmpl string) {
63    t, ferr := template.ParseFile("inc/"+tmpl+".html", nil)
64    if ferr != nil {
65      panic(os.NewError("template not found, panic!!"))
66    }
67    pr, err := p.RenderMarkup(tmpl)
68    if err != nil {
69      t.Execute(w, p)
70      return
71    }
72    t.Execute(w, pr)
73    return
74  }
75
76  // Render the wiki markup. this only supports links so far.
77  func (p *Page) RenderMarkup(handler string) (*Page, os.Error) {
78    if !strings.Contains(handler, "view") { // we only render markup if called from
          the view handler
79      return nil, os.NewError("not called from the view-handler, don't markup")
80    }
81    linkMarkup := regexp.MustCompile("\\[[A-Za-z09 ]+\\]")
82    parsedBody := linkMarkup.ReplaceAllStringFunc(p.Body, linkHelper)
83    return CreatePage(p.Title, parsedBody)
84  }
85
86  // A helper function to parse link strings
87  func linkHelper(s string) string {
88    slug := generateSlug(s[1:(len(s)-1)])
89    return "<a href='/view/"+slug+"'>"+s[1:(len(s)-1)]+"</a>"
90  }
```

A.3: Implementation of the page package

database.go

```
1  // Copyright 2011 Tobias Eriksson. All rights reserved.
2  // Use of this source code is governed by a BSD-style
3  // license that can be found in the COPYING file.
4
5  package database
6
7  import(
8    "github.com/mikejs/gomongo/mongo"
9    "os"
```

```
10    "strings"
11  )
12
13  type PageDoc struct {
14    Slug string
15    Title string
16    Body string
17  }
18
19  var db *mongo.Database
20
21  func ConnectToDB(url string, database string) (err os.Error){
22    conn, err := mongo.Connect(url)  // get a connection
23    db = conn.GetDB(database)
24    return
25  }
26
27  // Load a page from the database
28  func LoadPage(slug string) (*PageDoc, os.Error){
29    slugBSON, slugError := mongo.Marshal(map[string]string{ "slug" : slug})
30    if slugError != nil {
31      return nil, slugError
32    }
33    coll := db.GetCollection("pages") // get the 'pages' collection from the db
34    defer coll.Drop()
35
36    findBSON, findErr := coll.FindOne(slugBSON)
37    if findErr != nil {
38      return nil, findErr
39    }
40    var foundPage PageDoc
41    mongo.Unmarshal(findBSON.Bytes(), &foundPage)
42    return &foundPage, nil
43
44  }
45
46  // Save a page to the database
47  func SavePage(pageDoc map[string]string) os.Error {
48    idBSON, idError := mongo.Marshal(map[string]string{ "slug" : pageDoc["slug"],
          })
49    if idError != nil {
50      return idError
51    }
52
53    coll := db.GetCollection("pages") // get the 'pages' collection from the db
54    defer coll.Drop()
55
56    pageBSON, pageDocErr := mongo.Marshal(pageDoc)   // create a BSON document
57    if pageDocErr != nil {
58      return pageDocErr
59    }
60
61    _, findErr := coll.FindOne(idBSON) // check if the page is already in the
          database
62    if findErr != nil {
63      if strings.Contains(findErr.String(), "cursor failure") { // this means that
            nothing could be found
64        coll.Insert(pageBSON)                              // and insert it
65        return nil
66      }
67      return findErr
```

```
68    }
69    updateErr := coll.Update(idBSON, pageBSON)
70
71    if updateErr != nil {
72      return updateErr
73    }
74
75    return nil
76  }
```

A.4: Implementation of the database package