Extracted from:

# Powerful Command-Line Applications in Go

## Build Fast and Maintainable Tools



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Powerful Command-Line Applications in Go

## Build Fast and Maintainable Tools

Ricardo Gerardi
*edited by Brian P. Hogan*

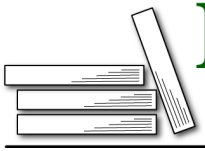
# Powerful Command-Line Applications in Go

## Build Fast and Maintainable Tools

Ricardo Gerardi

The Pragmatic Bookshelf

Raleigh, North Carolina





For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Brian P. Hogan
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



ISBN-13: 978-1-68050-696-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2021

*To my beloved wife Kassia. My best friend, my greatest supporter. Sadly she passed away due to cancer before seeing this book completed.*

*To my four incredible daughters Gisele, Livia, Elena, and Alice. The reason for everything.*

## Testing with Table-Driven Testing

When you're writing tests for your command-line tool, you often want to write test cases that cover different variations of the function or tool usage. By doing this, you ensure that the different parts of your code are working, increasing the reliability of your tests and tool. For example, to test the filterOut() function from the *walk* tool, it's a good idea to define test cases for the different conditions such as filtering with or without extension, matching or not, and minimum size.

One of the benefits of Go is that you can use Go itself to write test cases. You don't need a different language or external frameworks. By leveraging Go, you use all the language's features to help define your test cases. A common pattern for writing test cases that cover different variations of the function you're testing is known as *table-driven testing*. In this type of testing, you define your test cases as a slice of anonymous struct, containing the data required to run your tests and the expected results. You then iterate over this slice using loops to execute all test cases without repeating code. The Go testing package provides a convenient function Run() that runs a subtest with the specified name. Let's use this approach to test this version of the tool.

Create a new file called actions_test.go in the same directory as your actions.go file. Add the package definition and the import statement at the top of this file:

fileSystem/walk/actions_test.go
```
package main

import (
  "os"
  "testing"
)
```

You'll use the package os to handle file details; and the testing package that provides functions required to test your Go code.

Now, create a test function to test the filterOut() function.

fileSystem/walk/actions_test.go
```
func TestFilterOut(t *testing.T) {
```

Add the anonymous slice of struct with the definition of the test cases. The struct fields represent the values that we'll use for each test such as the test's name, file to read, extension to filter, minimum file size, and the expected test result:

fileSystem/walk/actions_test.go
```
testCases := []struct {
```

```
  name     string
  file     string
  ext      string
  minSize  int64
  expected bool
}{
  {"FilterNoExtension", "testdata/dir.log", "", 0, false},
  {"FilterExtensionMatch", "testdata/dir.log", ".log", 0, false},
  {"FilterExtensionNoMatch", "testdata/dir.log", ".sh", 0, true},
  {"FilterExtensionSizeMatch", "testdata/dir.log", ".log", 10, false},
  {"FilterExtensionSizeNoMatch", "testdata/dir.log", ".log", 20, true},
}
```

Each element of the slice represents a test case. For example, the first test case's name is "FilterNoExtension". This uses the file testdata/dir.log, the extension to filter is blank, the minimum size is zero, and we expect this test to return the Boolean value false. This is similar for the remaining test cases, each with different values.

Once you have the test cases defined, add the for loop to iterate over each test case. For each case, call the t.Run() method, providing the test name as the first parameter and an anonymous function of type func(t *testing.T) as the second parameter. Inside the anonymous function run the tests using the test case attributes defined before:

fileSystem/walk/actions_test.go

```
  for _, tc := range testCases {
    t.Run(tc.name, func(t *testing.T) {
      info, err := os.Stat(tc.file)
      if err != nil {
        t.Fatal(err)
      }

      f := filterOut(tc.file, tc.ext, tc.minSize, info)

      if f != tc.expected {
        t.Errorf("Expected '%t', got '%t' instead\n", tc.expected, f)
      }
    })
  }
}
```

For these tests, you first retrieve the file's attributes using the function os.Stat(). Then execute the filterOut() function providing these attributes and the test case parameters. Finally, compare the result with the expected result from the test case, failing the test if they don't match.

Now, let's add the integration test cases. Save the file actions_test.go, create a file main_test.go, and edit it. Include the package definition and the import list:

fileSystem/walk/main_test.go
```
package main

import (
  "bytes"
  "testing"
)
```

You'll use the package bytes to manipulate slices of bytes (such as the output of the tool) and the testing package that provides functions required to test your Go code.

Follow the same approach to test variations of the integration tests. Start by defining the test cases using the anonymous struct, followed by the loop to test each case. The main difference is that you use the run() function defined in main.go instead of the function filterOut(). Write the integration tests:

fileSystem/walk/main_test.go
```
func TestRun(t *testing.T) {
  testCases := []struct {
    name     string
    root     string
    cfg      config
    expected string
  }{
    {name: "NoFilter", root: "testdata",
      cfg:      config{ext: "", size: 0, list: true},
      expected: "testdata/dir.log\ntestdata/dir2/script.sh\n"},
    {name: "FilterExtensionMatch", root: "testdata",
      cfg:      config{ext: ".log", size: 0, list: true},
      expected: "testdata/dir.log\n"},
    {name: "FilterExtensionSizeMatch", root: "testdata",
      cfg:      config{ext: ".log", size: 10, list: true},
      expected: "testdata/dir.log\n"},
    {name: "FilterExtensionSizeNoMatch", root: "testdata",
      cfg:      config{ext: ".log", size: 20, list: true},
      expected: ""},
    {name: "FilterExtensionNoMatch", root: "testdata",
      cfg:      config{ext: ".gz", size: 0, list: true},
      expected: ""},
  }

  for _, tc := range testCases {
    t.Run(tc.name, func(t *testing.T) {
      var buffer bytes.Buffer

      if err := run(tc.root, &buffer, tc.cfg); err != nil {
        t.Fatal(err)
      }

      res := buffer.String()
```

```
      if tc.expected != res {
        t.Errorf("Expected %q, got %q instead\n", tc.expected, res)
      }
    })
  }
}
```

Save the main_test.go file and use a terminal to create the files required for testing. We need to create the directory containing the files we defined in the test cases earlier. We will use Go's convention and name this directory testdata, similarly to what we did in Writing Tests for the Markdown Preview Tool, on page ?, so that the Go build tool ignores it when compiling the program.

```
$ mkdir -p testdata/dir2
$ echo "Just a test" > testdata/dir.log
$ touch testdata/dir2/script.sh
$ tree testdata
testdata
├── dir2
│   └── script.sh
└── dir.log

1 directory, 2 files
```

Execute the tests using the go test -v tool:

```
$ go test -v
=== RUN   TestFilterOut
=== RUN   TestFilterOut/FilterNoExtension
=== RUN   TestFilterOut/FilterExtensionMatch
=== RUN   TestFilterOut/FilterExtensionNoMatch
=== RUN   TestFilterOut/FilterExtensionSizeMatch
=== RUN   TestFilterOut/FilterExtensionSizeNoMatch
--- PASS: TestFilterOut (0.00s)
    --- PASS: TestFilterOut/FilterNoExtension (0.00s)
    --- PASS: TestFilterOut/FilterExtensionMatch (0.00s)
    --- PASS: TestFilterOut/FilterExtensionNoMatch (0.00s)
    --- PASS: TestFilterOut/FilterExtensionSizeMatch (0.00s)
    --- PASS: TestFilterOut/FilterExtensionSizeNoMatch (0.00s)
=== RUN   TestRun
=== RUN   TestRun/NoFilter
=== RUN   TestRun/FilterExtensionMatch
=== RUN   TestRun/FilterExtensionSizeMatch
=== RUN   TestRun/FilterExtensionSizeNoMatch
=== RUN   TestRun/FilterExtensionNoMatch
--- PASS: TestRun (0.00s)
    --- PASS: TestRun/NoFilter (0.00s)
    --- PASS: TestRun/FilterExtensionMatch (0.00s)
    --- PASS: TestRun/FilterExtensionSizeMatch (0.00s)
    --- PASS: TestRun/FilterExtensionSizeNoMatch (0.00s)
    --- PASS: TestRun/FilterExtensionNoMatch (0.00s)
```

```
PASS
ok      pragprog.com/rggo/fileSystem/walk       0.005s
```

Notice that Go executes all test cases for each test function, using the test name you configured to present the results. This makes it easier to reference each test and troubleshoot them in case a test doesn't pass.

Since the tool is passing all tests, let's try it out. First, create a small directory tree in the /tmp directory that you can explore with your program. This structure will contain some *.txt* files and some *.log* files:

```
$ mkdir -p /tmp/testdir/{text,logs}
$ touch /tmp/testdir/file1.txt
$ touch /tmp/testdir/text/{text1,text2,text3}.txt
$ touch /tmp/testdir/logs/{log1,log2,log3}.log
$ ls /tmp/testdir/
file1.txt  logs  text
```

Now try your command-line tool, providing the -root parameter set to the newly created /tmp/testdir:

```
$ go run . -root /tmp/testdir/
/tmp/testdir/file1.txt
/tmp/testdir/logs/log1.log
/tmp/testdir/logs/log2.log
/tmp/testdir/logs/log3.log
/tmp/testdir/text/text1.txt
/tmp/testdir/text/text2.txt
/tmp/testdir/text/text3.txt
```

All the files in the specified directory tree are listed. You can display only log files by providing the *.log* extension to the -ext parameter, like this:

```
$ go run . -root /tmp/testdir/ -ext .log
/tmp/testdir/logs/log1.log
/tmp/testdir/logs/log2.log
/tmp/testdir/logs/log3.log
$
```

You can also filter results based on the file size, but I'll leave that as an exercise for you to do later.

This initial version of the tool lists all the files in a directory tree, but listing the names isn't useful. So we'll add another action to make this tool more useful.