

# Testing; how, what, why

*GopherChina 2019*

# Testing, in Go

How

What

Why

This is a presentation about testing in Go. We're going to talk about

how to test,

what you should test,

why you should test it.

# How to test

There are a wide range of experience levels in the room so lets start by talking about how to write unit tests for a Go package.

```
package split

import "strings"

// Split slices s into all substrings separated by sep and
// returns a slice of the substrings between those separators.
func Split(s, sep string) []string {
    var result []string
    i := strings.Index(s, sep)
    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):]
        i = strings.Index(s, sep)
    }
    return append(result, s)
}
```

Let's say we have this string splitting function

This function takes a string and a separator and returns a slice of the input string broken on the separator. How can we write a unit test for this function?

```
package split

import (
    "reflect"
    "testing"
)

func TestSplit(t *testing.T) {
    got := Split("a/b/c", "/")
    want := []string{"a", "b", "c"}
    if !reflect.DeepEqual(want, got) {
        t.Fatalf("expected: %v, got: %v", want, got)
    }
}
```

We start with a file in the same directory, with the same package name

We call Split with some inputs, then compare it to the result we expected.

## Tests are just like regular Go functions

1. The name of the test function must start with **Test**
2. The test function must take one argument of type **\*testing.T**. A **\*testing.T** is a type injected by the testing package itself, to provide ways to print, skip, and fail the test.

Tests are just regular Go functions with a few rules.

## Tests live in `_test.go` files

**Your code**

```
% ls -al
total 16
drwxr-xr-x  4 dfc  staff  128  7 Apr 14:34 .
drwxr-xr-x 106 dfc  staff 3392  6 Feb 13:11 ..
-rw-r--r--  1 dfc  staff  401  7 Apr 14:42 split.go
-rw-r--r--  1 dfc  staff  956  7 Apr 17:45 split_test.go
```

**Your code's tests**

Because test functions are just regular public Go functions, we don't want to include them as part of our package's API.

Instead, we want to only compile them in the context of running our tests. To do this the go test commands understands that files ending with `_test.go` are only compiled in `_test scope_`.

Now we know that a package's unit tests live alongside the package's code in the same directory.

## How do you run your package's tests?

```
% go test  
PASS  
ok      split    0.005s
```

So, now we have a function, and its tests, we can run them.

How do we do that? We type `go test`

`Go test` will run the tests in the current package, the directory your in.



# How do you run all package's tests?

```
(~/src/github.com/heptio/contour) % go test ./...
?      github.com/heptio/contour/apis/contour/v1beta1 [no test files]
?      github.com/heptio/contour/apis/generated/clientset/versioned [no test files]
?      github.com/heptio/contour/apis/generated/clientset/versioned/fake [no test files]
?      github.com/heptio/contour/apis/generated/clientset/versioned/scheme [no test files]
?      github.com/heptio/contour/apis/generated/clientset/versioned/typed/contour/v1beta1 [no test files]
?      github.com/heptio/contour/apis/generated/clientset/versioned/typed/contour/v1beta1/fake [no test files]
?      github.com/heptio/contour/apis/generated/informers/externalversions [no test files]
?      github.com/heptio/contour/apis/generated/informers/externalversions/contour [no test files]
?      github.com/heptio/contour/apis/generated/informers/externalversions/contour/v1beta1 [no test files]
?      github.com/heptio/contour/apis/generated/informers/externalversions/internalinterfaces [no test files]
?      github.com/heptio/contour/apis/generated/listers/contour/v1beta1 [no test files]
ok     github.com/heptio/contour/cmd/contour 0.032s
ok     github.com/heptio/contour/internal/contour 0.052s
ok     github.com/heptio/contour/internal/dag 0.039s
?      github.com/heptio/contour/internal/debug [no test files]
ok     github.com/heptio/contour/internal/e2e 0.488s
ok     github.com/heptio/contour/internal/envoy 0.053s
ok     github.com/heptio/contour/internal/grpc 0.455s
?      github.com/heptio/contour/internal/httpsvc [no test files]
ok     github.com/heptio/contour/internal/k8s 0.039s
ok     github.com/heptio/contour/internal/metrics 0.032s
```

If you have many packages in your project you can use the glob operator `./...` to run some or all of your tests.

Here's an example from the product I work on.

## What about code coverage?

```
% go test -coverprofile=c.out
PASS
coverage: 100.0% of statements
ok      split    0.010s
% go tool cover -func=c.out
split/split.go:8:      Split          100.0%
total:                  (statements)  100.0%
```

The next question is, what is the coverage of this package? Luckily the go tool has a built in branch coverage. We can invoke it like this

Which tells us we have 100% branch coverage, which isn't really surprising, there's only one branch in this code, a loop.

(click)

If we want to dig in to the coverage report the coverage tool has several options to print the coverage report. The first is to break down the coverage per function;

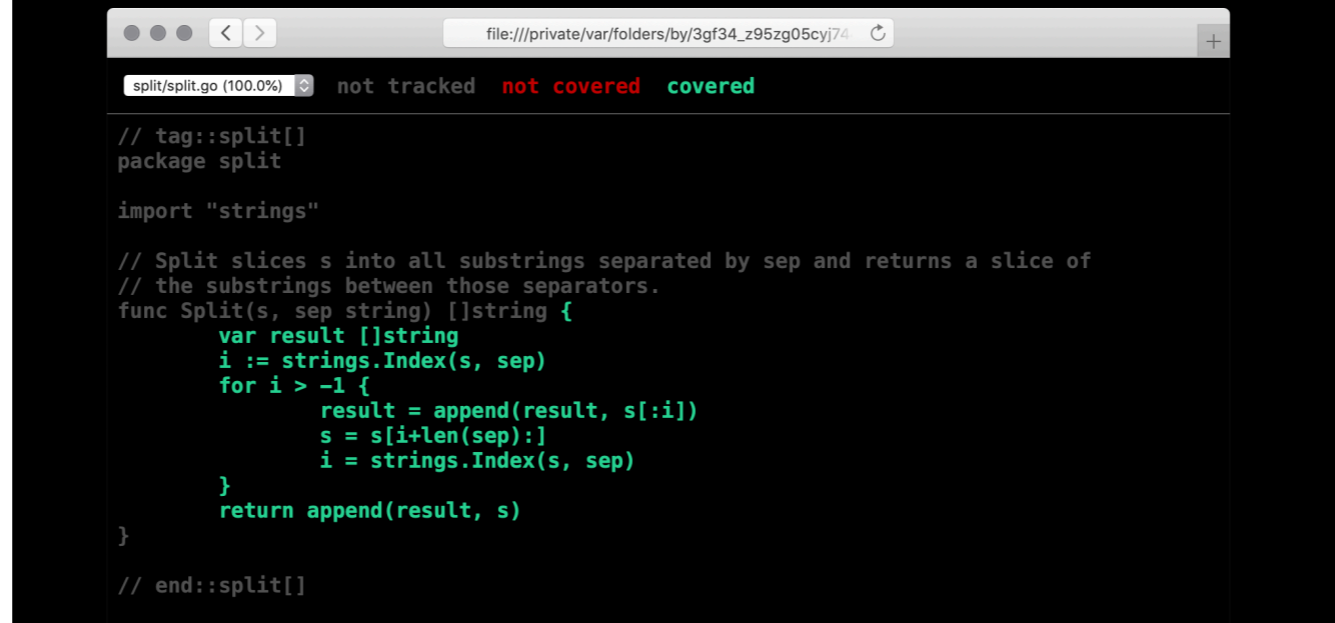
Which isn't that exciting as we only have one function in this package, but I'm sure you'll find more exciting packages to test the coverage of.

~/**.bashrc**

```
cover () {  
    local t=$(mktemp -t cover)  
    go test $COVERFLAGS -coverprofile=$t $@ \  
        && go tool cover -func=$t \  
        && unlink $t  
}
```

This is so useful for me I have a shell alias which runs the test coverage and the report in one command

# HTML coverage view



```
// tag::split[]
package split

import "strings"

// Split slices s into all substrings separated by sep and returns a slice of
// the substrings between those separators.
func Split(s, sep string) []string {
    var result []string
    i := strings.Index(s, sep)
    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):]
        i = strings.Index(s, sep)
    }
    return append(result, s)
}

// end::split[]
```

If you want to explore your code's coverage you can use the `-html` flag which will produce a html report

The green areas have coverage, the red areas dont.

## Recap

- ★ Wrote some code
- ★ Wrote a test
- ★ 100% test coverage



So, we wrote one test case, got 100% coverage, brilliant. Who thinks that we're done?

[ show of hands ]

You're right, we have good branch coverage but we probably need to test some of the boundary conditions.

```
package split

import (
    "reflect"
    "testing"
)

func TestSplit(t *testing.T) {
    got := Split("a/b/c", "/")
    want := []string{"a", "b", "c"}
    if !reflect.DeepEqual(want, got) {
        t.Fatalf("expected: %v, got: %v", want, got)
    }
}
```

[ 07:00 ]

For example, our string is some kind of path, what happens if we try to split it on comma?

(click)

What about if there are no separators in the source string?

(click)

Now we have a bunch of test cases, this is good, yes? Good coverage of the boundary conditions.

What are some bad things you can see about our test file? Yes, its getting long.

(click)

Well, long isn't a problem, if you have a complicated function you will have to test all the edge cases. There's something more subtle wrong with this test file.

The problem is there is a lot of duplication, for each test case only the want and got are different, oh, and the name.

```

func TestSplit(t *testing.T) {
    type test struct {
        input string
        sep    string
        want  []string
    }

    tests := []test{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
    }

    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
        }
    }
}

```

What we'd like to do is set up all the inputs and expected outputs and feed them to the test harness. This is a great time to introduce a table driven test.

So let's talk about this, we declare a structure to hold our test inputs and expected outputs. We're going to make this a local declaration, just inside the function because we probably want to reuse the type test for other tests in this package.

```
func TestSplit(t *testing.T) {
    tests := []struct {
        input string
        sep    string
        want   []string
    }{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
    }

    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
        }
    }
}
```

In fact, we don't even need to give it a name, we can use an anonymous struct literal to reduce the boilerplate a little more.

You can even, in some cases eliminate the field names, but that's probably going to harm readability and means you have to declare each field in the struct.



```
func TestSplit(t *testing.T) {
    tests := []struct {
        input string
        sep   string
        want  []string
    }{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
        {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}}, // trailing sep
    }

    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
        }
    }
}
```

[ 11:00 ]

What will happen if our string to be split has a trailing separator?

Now we have our test table adding a test for this is as simple as adding a new line.

And when we run go test, we get

(click)

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:24: expected: [a b c], got: [a b c ]
FAIL
exit status 1
FAIL    split    0.006s
```

Cool, so there are a few things to talk about here.

The first is by rewriting the test from a function to a row in a table we've lost the name of the failing test. We added a comment in the test file to call out this case, but we don't have access to that.

There are a few ways to resolve this, and you'll see these styles in use go code bases because this style of testing is very much still evolving.

```
func TestSplit(t *testing.T) {
    tests := []struct {
        input string
        sep   string
        want  []string
    }{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
        {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}}, // trailing
    sep
    }

    for i, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("test %d: expected: %v, got: %v", i+1, tc.want, got)
        }
    }
}
```

As tests are stored in a slice we can print out the index of the test

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:24: test 4: expected: [a b c],
got: [a b c ]
FAIL
exit status 1
FAIL    split    0.005s
```

Now when we run this we get

Which is a little better. We know this is the fourth test, although we have to do a little bit of fudging because slice indexing—and range iteration—is zero based. It requires consistency across your test cases; if some use zero base reporting and others use one based, it going to be confusing

And if the list of test cases is long, it could be difficult to count braces to figure out exactly which line is test case 4.

```
func TestSplit(t *testing.T) {
    tests := []struct {
        name string
        input string
        sep string
        want []string
    }{
        {name: "simple", input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {name: "wrong sep", input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {name: "no sep", input: "abc", sep: "/", want: []string{"abc"}},
        {name: "trailing sep", input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("%s: expected: %v, got: %v", tc.name, tc.want, got)
        }
    }
}
```

Another common pattern is to include a name field in the test fixture.

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:25: trailing sep: expected: [a b
c], got: [a b c ]
FAIL
exit status 1
FAIL    split    0.005s
```

Now when the test fails we have a descriptive name for what the test was doing—we no longer have to try to figure it out from the arguments—and a string we can search on.

```

func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep    string
        want   []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
        }
    }
}

```

[ 15:00 ]

And it turns out we can dry this up even more

Using a map literal syntax we define our test cases not as a slice of structs, but a map of test names to test fixtures.

There's also a side benefit of using a map that is going to potentially improve the utility of our tests. Can you guess what it is?

[ show of hands]

Storing test cases in a map means  
they are run in an *undefined* order

Yes, that's right. Map iteration is undefined—some people say random, it's not even defined to be random, although in practice it is—which means each of our tests are going to be run in an undefined order.

This is super useful for spotting conditions where two test pass when run in a certain order, but not otherwise. If you find that happens you probably have a lot of global state and your tests are not hermetic



```

func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep    string
        want   []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
        }
    }
}

```

But before we go on to fix the test there are two remaining issues with our test harness. Can anyone think of what they are?

[show of hands]

```

func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep    string
        want   []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
        }
    }
}

```

The first is we're calling `t.Fatalf` when one of the test cases fails.

This means the first failing test case and we stop testing the other cases.

Because test cases are run in an undefined order, if we refactor our function—which we're just about to do—and have a test failure, we'd like to know was that the only one or is it just the first.

The testing package will do this for us if we go to the effort to write out each test case as its own function, but can we fix the test harness to report all the test failures at the same time? Can you give me a way we can do it?

```

func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep    string
        want   []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Errorf("%s: expected: %v, got: %v", name, tc.want, got)
            // execution continues
        }
    }
}

```

Yes, we could change the `t.Fatalf` to `t.Errorf`. `t.Errorf` marks the test as failing, but does not terminate execution like `t.Fatalf` does.

But say we had more than one assertion, its going to be complicated to remember to skip those other checks—they might panic—with a `continue` or something.

## Sub tests

*New in Go 1.7!*

[ 20:00 ]

Fortunately in Go 1.7 a feature was added to the testing package called sub tests. Sub tests let us write a test harness as if it were a single test function.

```

func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep   string
        want  []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        t.Run(name, func(t *testing.T) {
            got := Split(tc.input, tc.sep)
            if !reflect.DeepEqual(tc.want, got) {
                t.Fatalf("expected: %v, got: %v", tc.want, got)
            }
        })
    }
}

```

This is what a table driven sub test looks like

Because the subtest now has a name we get that name automatically printed out in any test runs.

Each subtest is its own anonymous function We can therefore use Fatal, Skip, etc and all the other testing.T helpers, while retaining the compactness of a table driven test

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a
b c ]
FAIL
exit status 1
FAIL    split    0.005s
```

Because the subtest now has a name we get that name automatically printed out in any test runs

```
% go test -run=.*trailing -v
=== RUN   TestSplit
=== RUN   TestSplit/trailing_sep
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: []string{"a",
        "b", "c"}, got: []string{"a", "b", "c", ""}
FAIL
exit status 1
```

You can also run sub tests by name using the `-run`` flag

```
split_test.go:25: expected: [a b c], got: [a b c ]
```

Ok, now we're ready to fix the test case. Let's look at the error ...

Can you spot the problem? Clearly the slices are different, that's what `reflect.DeepEqual` is upset about, but spotting the actual difference isn't easy; you have to spot the extra space after the `c`.

(click)



```
for name, tc := range tests {
    t.Run(name, func(t *testing.T) {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %#v, got: %#v", tc.want, got)
        }
    })
}
```

We can improve the output if we switch to the %#v printf syntax;

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: []string{"a", "b", "c"}, got: []string{"a", "b", "c", ""}
FAIL
exit status 1
FAIL    split    0.005s
```

Now when we run out test we get

Ok, now it's clear that the problem is there is an extra blank element in the slice.

But before we go to fix this I want to talk a little bit more about choosing the right way to present test failures.

Our Split function is simple, it takes a primitive string and returns a slice of strings, but what if it worked with structs, or worse, pointers to structs.

```
func main() {
    type T struct {
        I int
    }
    x := []*T{{1}, {2}, {3}}
    y := []*T{{1}, {2}, {4}}

    fmt.Printf("%v %v\n", x, y)
    fmt.Printf("%#v %#v\n", x, y)
}
```

```
% go run .  
[0xc000096000 0xc000096008 0xc000096010]  
[0xc000096018 0xc000096020 0xc000096028]  
[*main.T{(*main.T)(0xc000096000), (*main.T)  
(0xc000096008), (*main.T)(0xc000096010)}  
[*main.T{(*main.T)(0xc000096018), (*main.T)  
(0xc000096020), (*main.T)(0xc000096028)}]
```

## Pretty print, please

<https://github.com/k0kubun/pp>

<https://github.com/davecgh/go-spew>

<https://github.com/google/go-cmp>

The problem of printing a complex go structure are not new. There are dozens of them, like pp, or go-spew, but pretty printing is only one half of the problem, we still have reflect.DeepEqual

So I want to introduce the go-cmp library from Google. This was introduced about two years ago by Joe Tsai. He gave a talk about it at GopherCon in 2017

The screenshot shows a web browser window displaying the GitHub repository page for 'google/go-cmp'. The browser's address bar shows 'github.com'. The repository name 'google / go-cmp' is at the top left, with 'Watch 23', 'Star 858', and 'Fork 58' buttons to its right. Below the repository name are tabs for 'Code', 'Issues 8', 'Pull requests 4', 'Projects 0', and 'Insights'. The current file path is 'Branch: master go-cmp / README.md', with 'Find file' and 'Copy path' buttons. A commit by 'ferhatelmas' is shown, with the message 'Fix some typos in comments and readme (#54)' and commit hash '3f298f3' on '24 Nov 2017'. Below this, it says '3 contributors' with three profile icons. The file statistics are '45 lines (30 sloc) | 1.6 KB', with 'Raw', 'Blame', and 'History' buttons. The main content is the README for 'Package for equality of Go values', which includes tags 'godoc', 'reference', and 'build passing'. The text describes the package as a more powerful and safer alternative to 'reflect.DeepEqual' and lists its primary features, starting with 'When the default behavior of equality does not suit the needs of the test, custom equality functions can override the'.

google / go-cmp

Watch 23 Star 858 Fork 58

Code Issues 8 Pull requests 4 Projects 0 Insights

Branch: master go-cmp / README.md Find file Copy path

ferhatelmas Fix some typos in comments and readme (#54) 3f298f3 on 24 Nov 2017

3 contributors

45 lines (30 sloc) | 1.6 KB Raw Blame History

## Package for equality of Go values

godoc reference build passing

This package is intended to be a more powerful and safer alternative to `reflect.DeepEqual` for comparing whether two values are semantically equal.

The primary features of `cmp` are:

- When the default behavior of equality does not suit the needs of the test, custom equality functions can override the

```
func main() {
    type T struct {
        I int
    }
    x := []*T{{1}, {2}, {3}}
    y := []*T{{1}, {2}, {4}}

    fmt.Println(cmp.Equal(x, y)) // false
}
```

The goal of the compare library is it is specifically to compare two values. This is similar to `reflect.DeepEqual`, but it has more capabilities. You can of course write

```
func main() {  
    type T struct {  
        I int  
    }  
    x := []*T{{1}, {2}, {3}}  
    y := []*T{{1}, {2}, {4}}  
  
    diff := cmp.Diff(x, y)  
    fmt.Printf(diff)  
}
```

But far more useful for us with our test function is the Diff method which will produce a textual description of what is different between the two values, recursively.



```
% go run .  
{[*]main.T}[2].I:  
  -: 3  
  +: 4
```

So the first difference was in the slice of \*T's, at index 2, the value of I was expected to be 3, but was actually 4.

go-cmp is a very useful library, I don't have time to go into it in more detail here—watch Joe's talk. Let's apply go-cmp to our test harness and see what we get

```
func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep    string
        want   []string
    }{
        "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for name, tc := range tests {
        t.Run(name, func(t *testing.T) {
            got := Split(tc.input, tc.sep)
            diff := cmp.Diff(tc.want, got)
            if diff != "" {
                t.Fatalf(diff)
            }
        })
    }
}
```

```
% go test
--- FAIL: TestSplitTrailingSep (0.00s)
    split_test.go:35: expected: [a b c],
got: [a b c ]
FAIL
exit status 1
FAIL    split    0.005s
```

So now instead of this

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:27: {[]string}[?->3]:
            -: <non-existent>
            +: ""

FAIL
exit status 1
FAIL    split    0.006s
```

We get this. Now our test isn't just telling us that what we got and what we wanted were different, its telling us how they differed.

The results tell us that the strings are different lengths, the third index in the fixture didn't exist, but in the actual output we got an empty string, "".

```
// Split slices s into all substrings separated by sep and
// returns a slice of
// the substrings between those separators.
func Split(s, sep string) []string {
    var result []string
    i := strings.Index(s, sep)
    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):]
        i = strings.Index(s, sep)
    }
    if len(s) > 0 {
        result = append(result, s)
    }
    return result
}
```

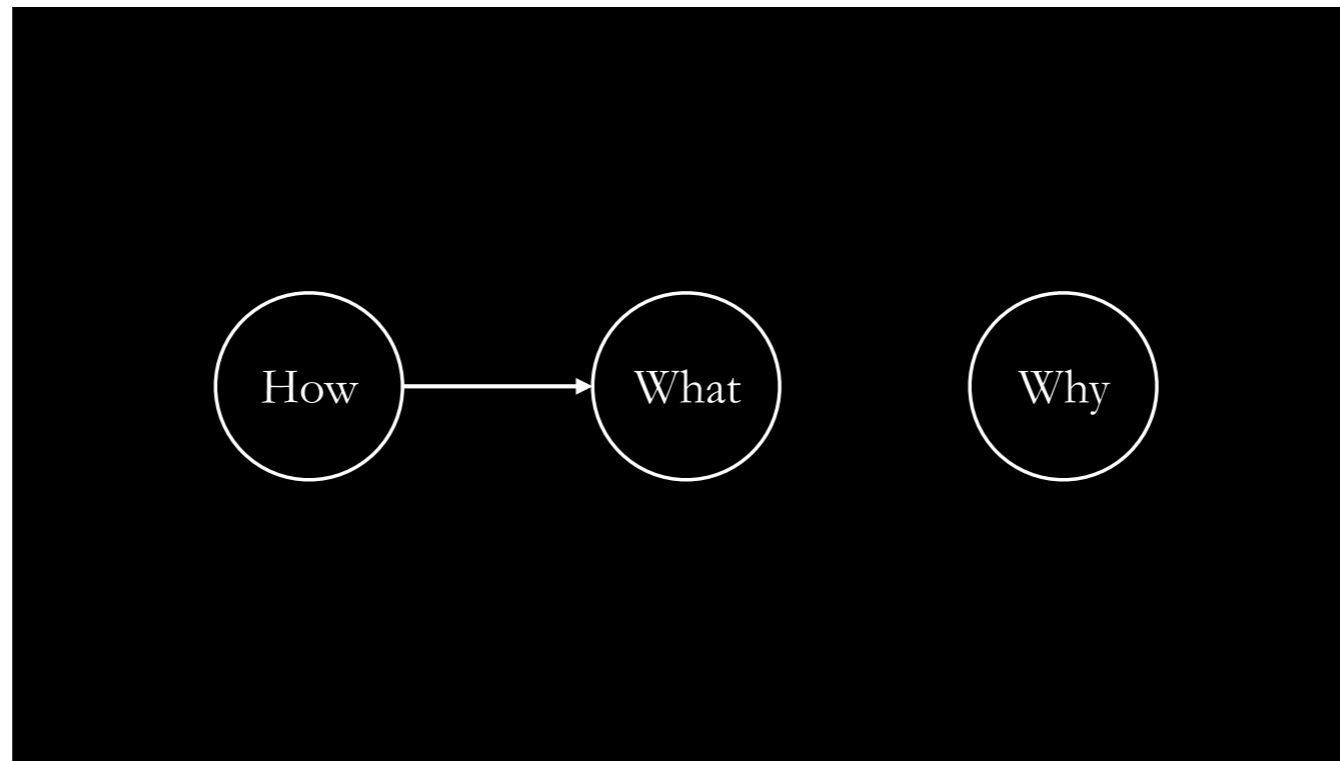
Now we can fix our split function to deal with a trailing separator

```
% go test  
PASS  
ok      split    0.005s
```

Now our tests pass

```
% cover
PASS
coverage: 100.0% of statements
ok      split    0.007s
split/split.go:8:      Split      100.0%
total:                (statements) 100.0%
```

And we check our coverage to make sure we haven't introduced any uncovered branches



[ 30:00 ]

We've talked about how to write a good Go test, so lets step back a bit and talk about what kind of things should be tested.

(click)



# How should we choose what to test?

How should we choose what parts of our code to test?

Should we test everything?

[show of hands]

## How should we choose when to write our tests?

A slightly different question is how should we choose when to write our tests?

Should we write them after we write the code?

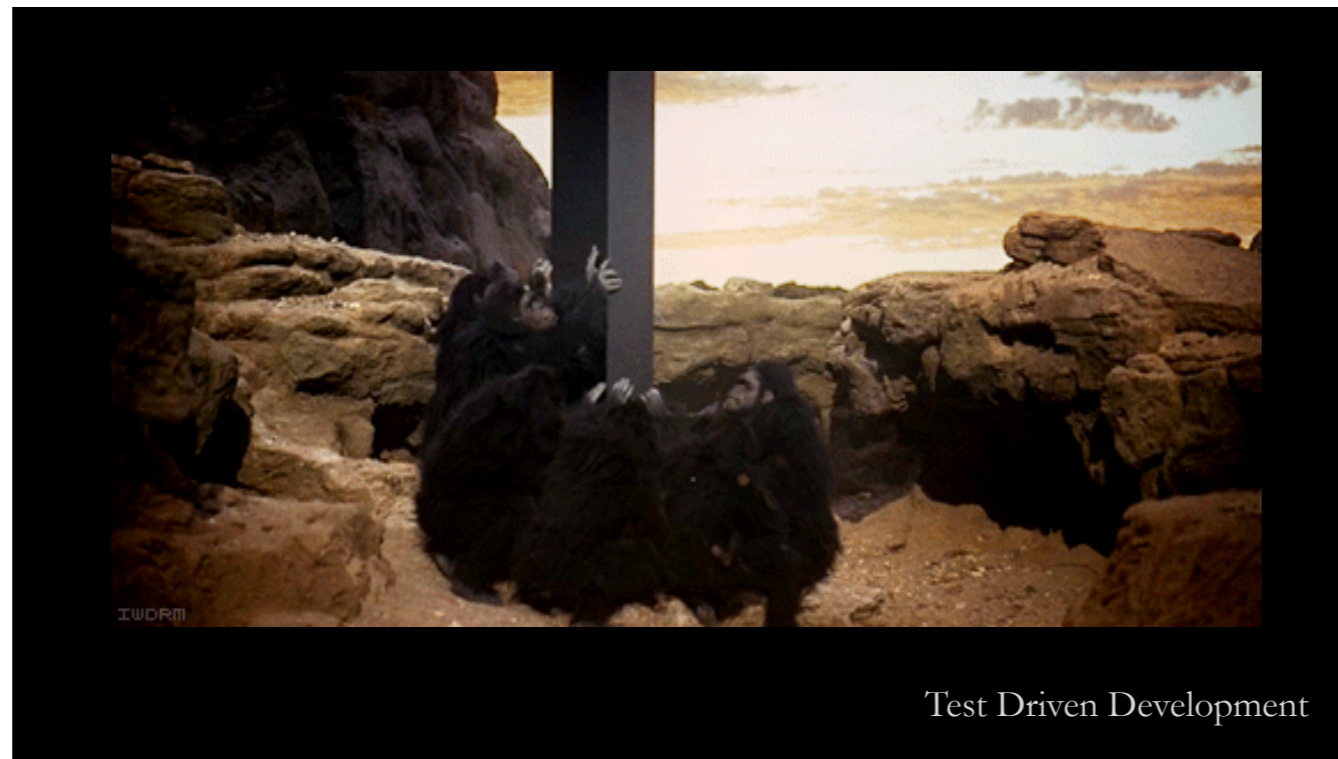
Should someone else write them? Maybe a QA group? Maybe a team integrating with your code?

Should we write them before we write our code? Should the architects of the project write them?

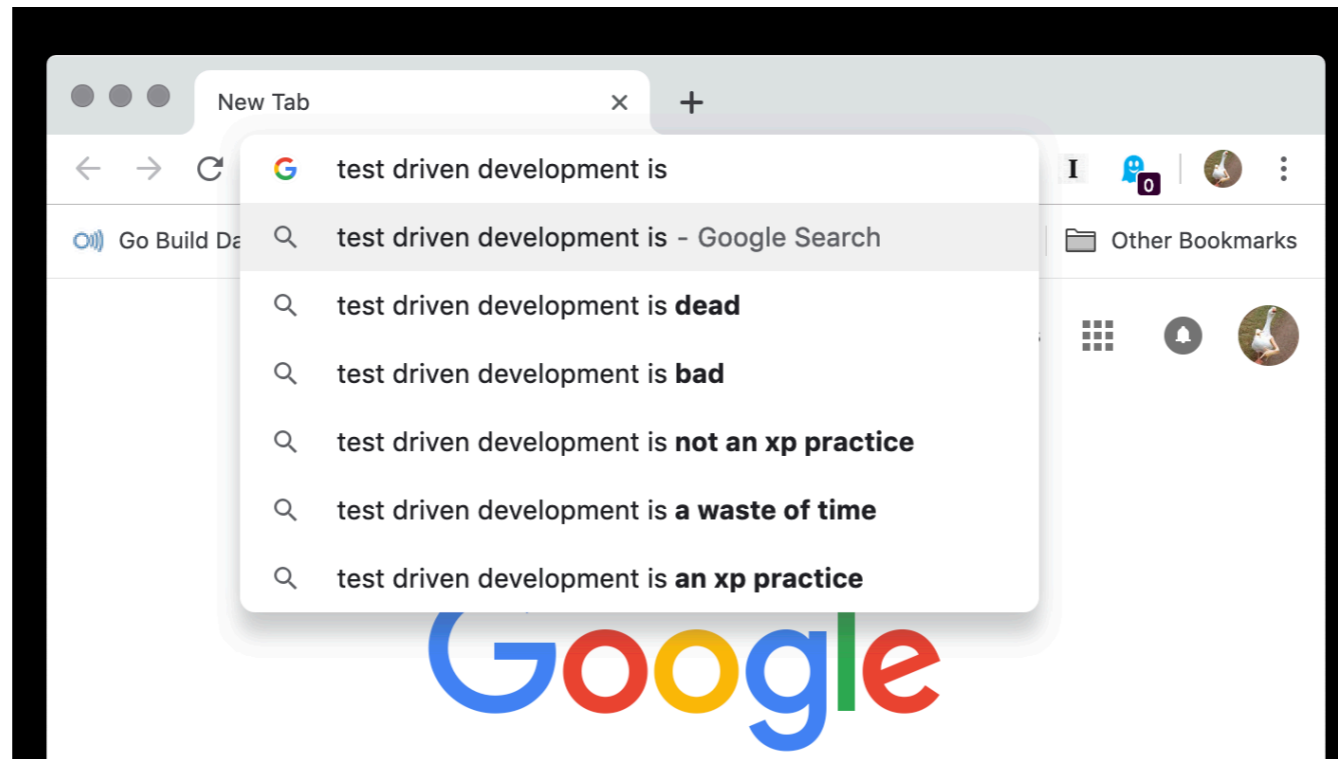
**I think tests should be written at  
the same time as we write our code**

My answer to this is we, you and I, the developers of the code, should write the tests at the same time as we're writing our code.

Does anyone know what this practice is called?



yes, test driven development



So, show of hands, who likes TDD?

Who sorta likes it?

Who thinks is a bad idea?

Who's tried to love TDD and hit a roadblock?

I think I'm probably in the latter category. I think that TDD is the right way to develop software — testing something after its built is soooo waterfall,

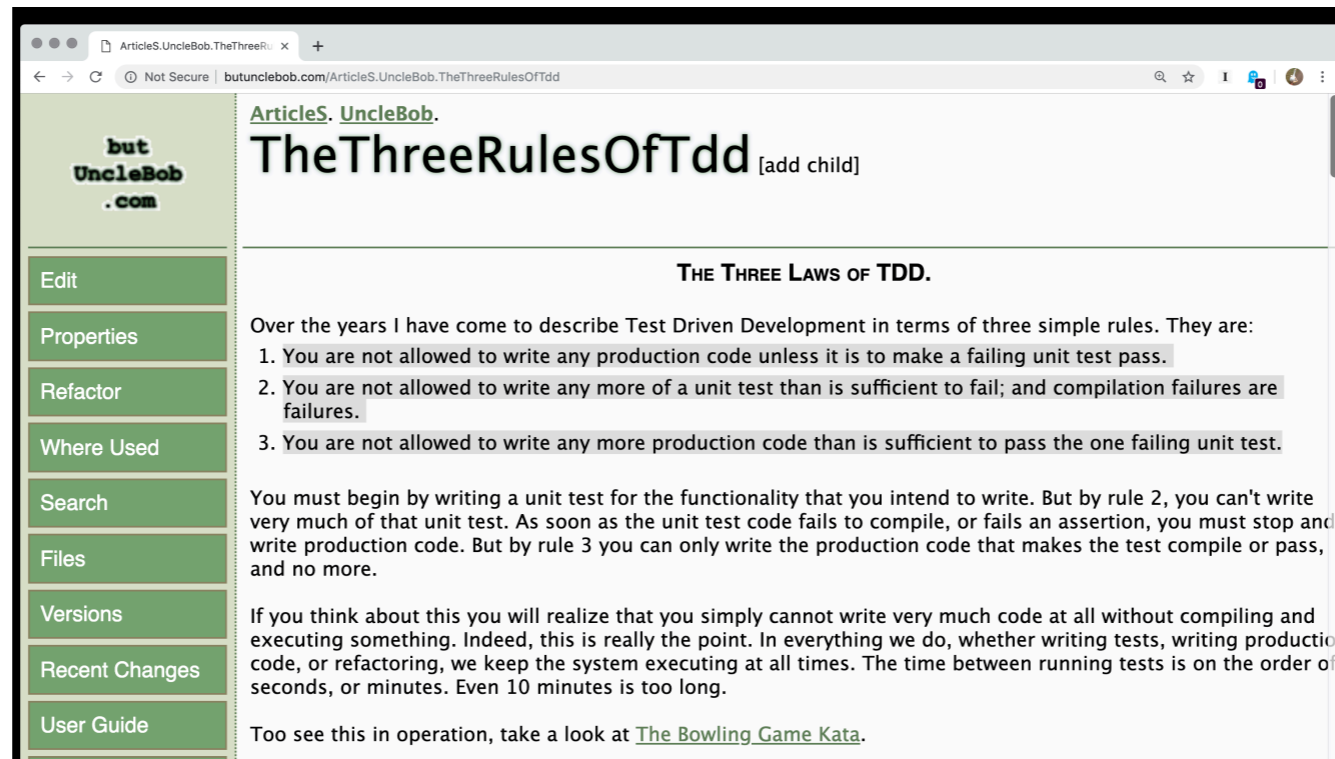
its just that trying to apply these TDD ideas as we're taught, seems to always feel like swimming up stream

**I hold it as an article of faith that  
writing tests at the same time as the  
code is a good thing.**

Never the less, I hold as an article of faith that writing tests concurrently with writing the code is a good thing.

Who agree's with me?

Who disagrees with me?



I don't have a software engineering background, I came to this industry from the sysadmin administration school of hard knocks.

This is how I was introduced to TDD.

[ read ]

If you've ever seen Bob Martin talk, you probably know his metre

[ read ]

And this all sounds very sensible, and to hear Martin speak of the benefits of never being more than a minute or so from having shippable code, never having the build broken again, was intoxicating.

Beck and others call this Red/Green/Refactor, write a failing test, fix the test, clean up the code.

## My life with TDD

If I knew what I wanted to write, then writing the failing test first was easy.

If I *didn't* know what I was writing, the cost of experimentation *started* at 200%; change the code, fix the test.

This cost went up and to the right when one function called another.

I wrote most of the gorilla/http package like this. It was wonderful to know when I'd implemented a feature, I'd also implemented the tests for that feature. That was the canonical definition of `_done_` for me.

Nothing pisses me off more than being in a standup and hearing "i've fixed the bug, I just have to write a test for it"

But, as I'm sure you all appreciate, its not all beer and skittles when it comes to TDD

When I didn't know what I wanted to write, I was doing research exploring the problem space by writing code, then working TDD style was twice as expensive; every time I back tracked on the code, I needed to first back track on the test.

Worse, depending on the degree I wanted to course correct, the cost was higher than 200%.



**Changing well covered code felt  
safe, changing the tests felt like  
cheating**

I was deeply concerned about rewriting my tests all the time.

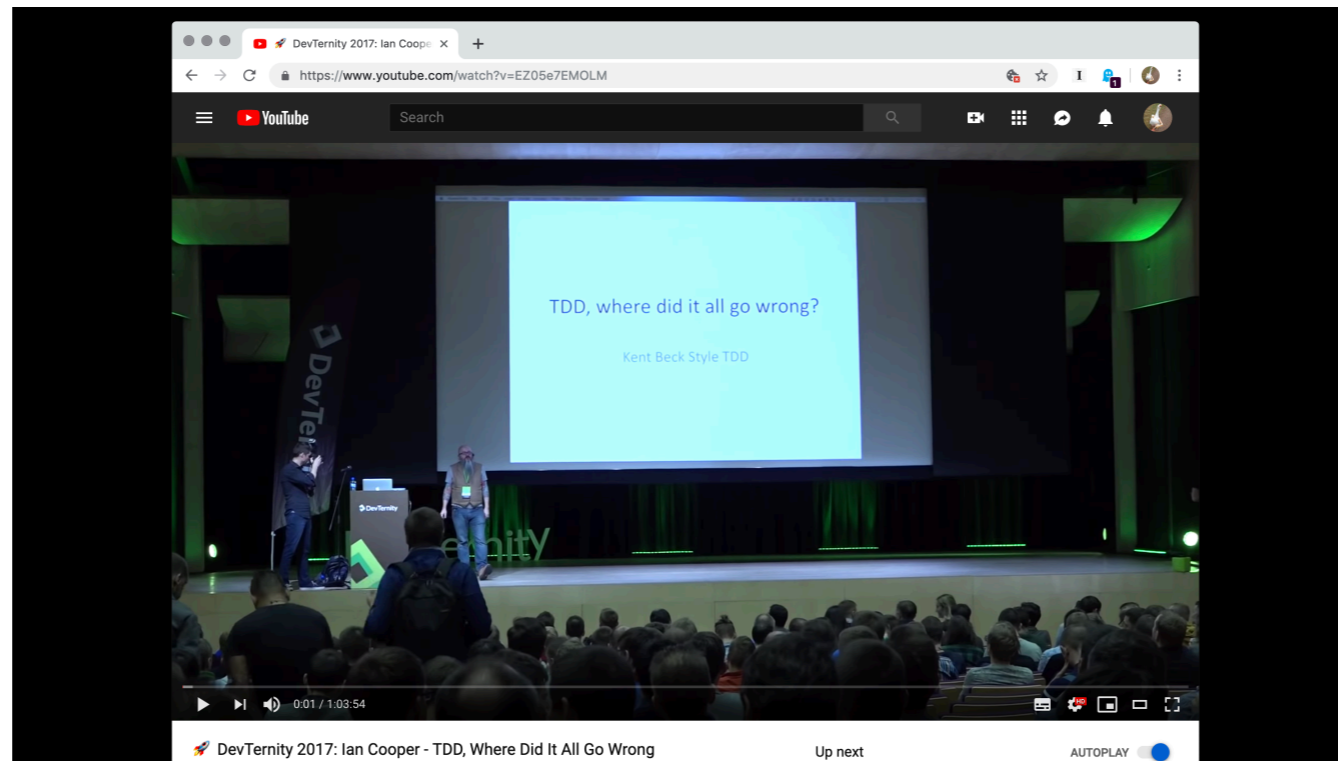
It goes like this, test's check that your code works how it says on the tin, they are our insurance policy, but who tests our tests?

The best way to get the tests passing is just to delete the ones that fail.



Is this the answer? Screw this testing nonsense, its only slowing us down?

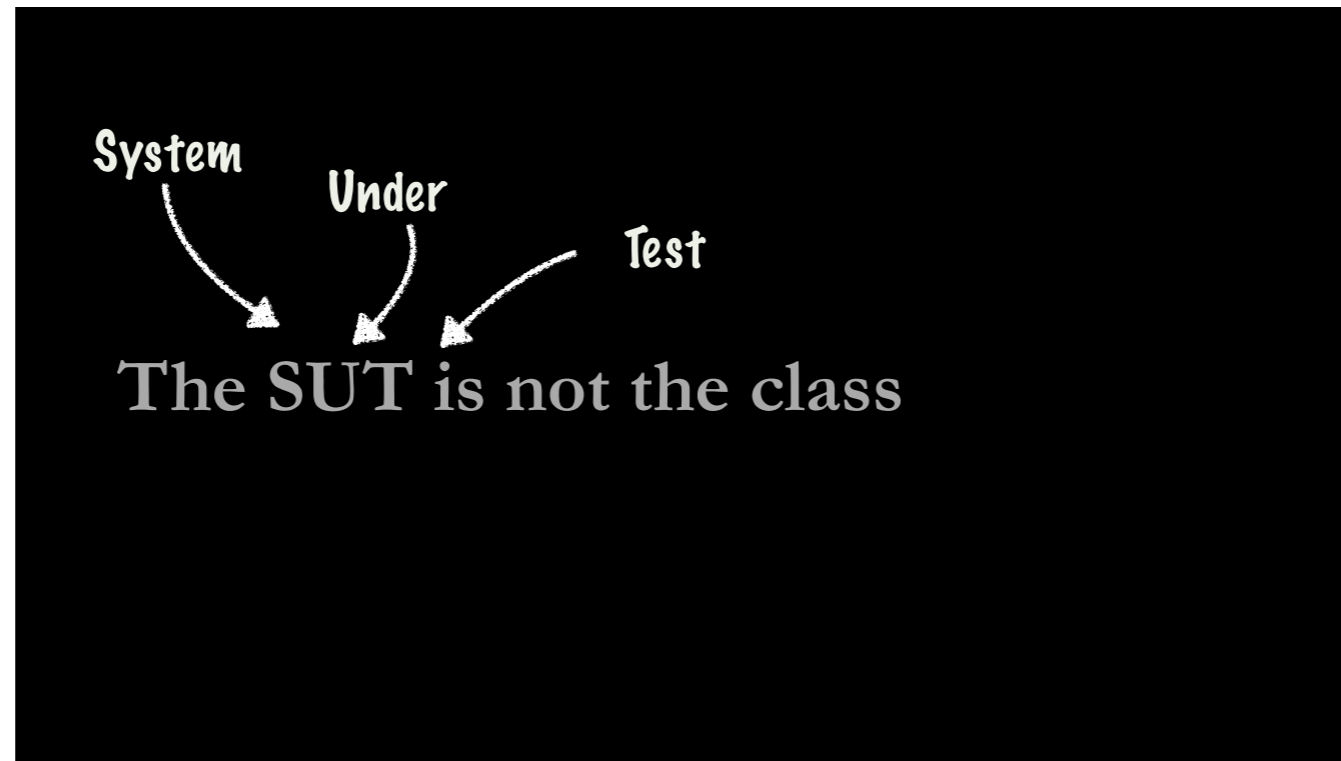
Constantly rewriting tests as I developed a piece of software was a lot of unnecessary work, and it felt like cheating. It wasn't testing, it was just making the test assert the current behaviour. If that behaviour changed, I'd change the test to match, and that was the wrong way around, IMO. That's not a test at this point, it's an accessory after the fact



Then one day I ran into this amazing presentation by Ian Cooper titled; TDD, we're did it all go wrong

I strongly recommend you watch this presentation. I know this might be a bit difficult in china, but I'll include a link to the video at the end of the presentation.

It contains a lot more wisdom than just the two take aways I'm going to focus on



The first was the system under test, the unit, is not the class

Now Ian's a dot net kind of fellow so he can be forgiven for saying the C word.

We're Go programmers, we try not to use that word in polite company

If we think about the idea of system under test, the unit, and ask the question more broadly

What is the unit of code in a C program?

What is the system under test in a C program, what is the “unit” that we are testing? I argue its a function

**What is the unit of code in a Java program?**

What is the system under test in a Java program, what is the “unit”? If you said “class” then you should watch Ian’s presentation.

Classical testing in java is the class, the methods in the class.

What is the unit of code in a Go program?

What is the system under test in a Go program, what is the “unit”?

Watching Ian’s talk, this is the question that I asked myself

“Go packages embody the spirit of the  
UNIX philosophy. Go packages  
interact with one another via interfaces.  
Programs are composed, just like the  
UNIX shell, by combining packages  
together.”

*—Me, a few years ago*

And it got me thinking about something I'd said a few years ago. I was really high on Doug McIlroy and I was smitten with the idea that Go packages were composed into programs.

[ read ]

And approaching this argument from Cooper's assertion that the unit of code is not a type, or function, or a method, or a class, suddenly it all clicked for me.



**The unit of software in Go is the  
package**

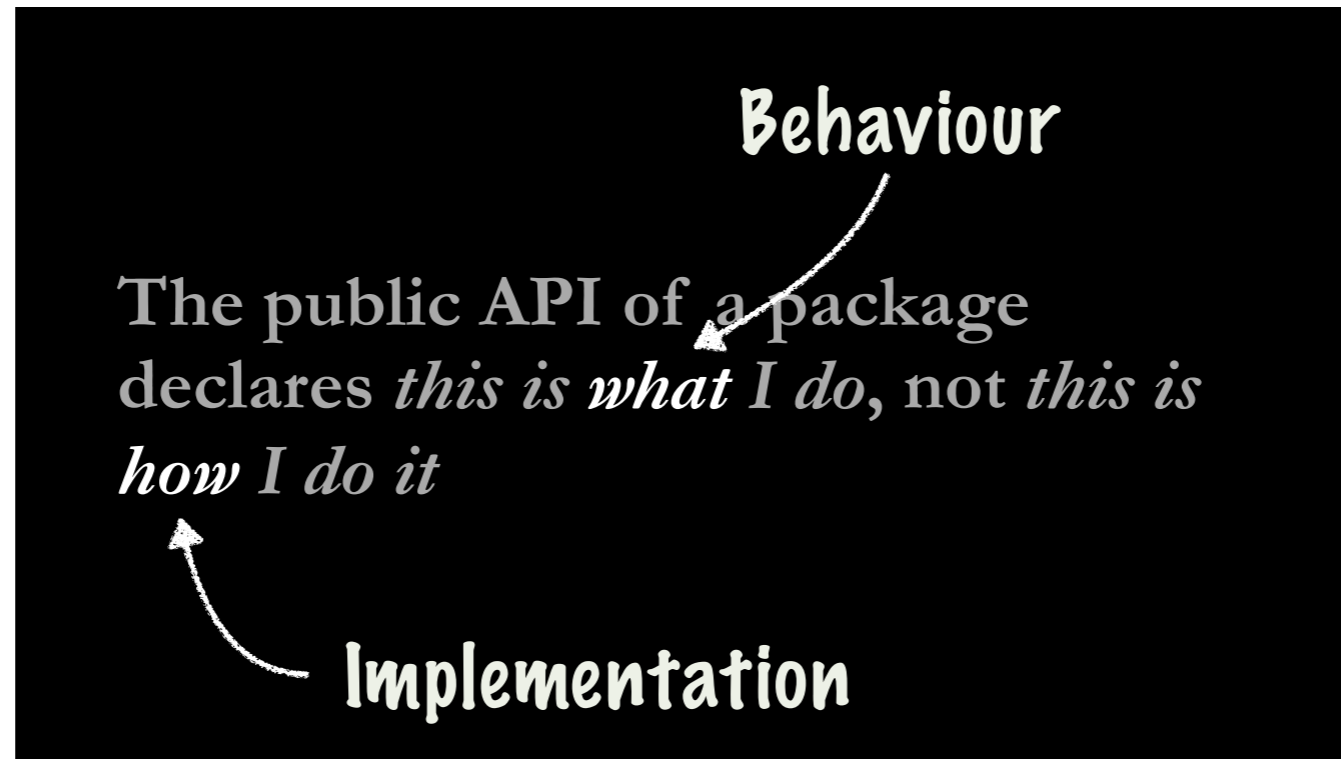
I argue that the unit of Go software is the package

## Test the behaviour of your unit, not its implementation

The second take away from Ian's talk is to test the behaviour of your unit, NOT the implementation

To an external observer, A caller of your code, they cannot observe its behaviour unless its exposed via a public api. Anything which is not exposed via a public api is thus an implementation detail

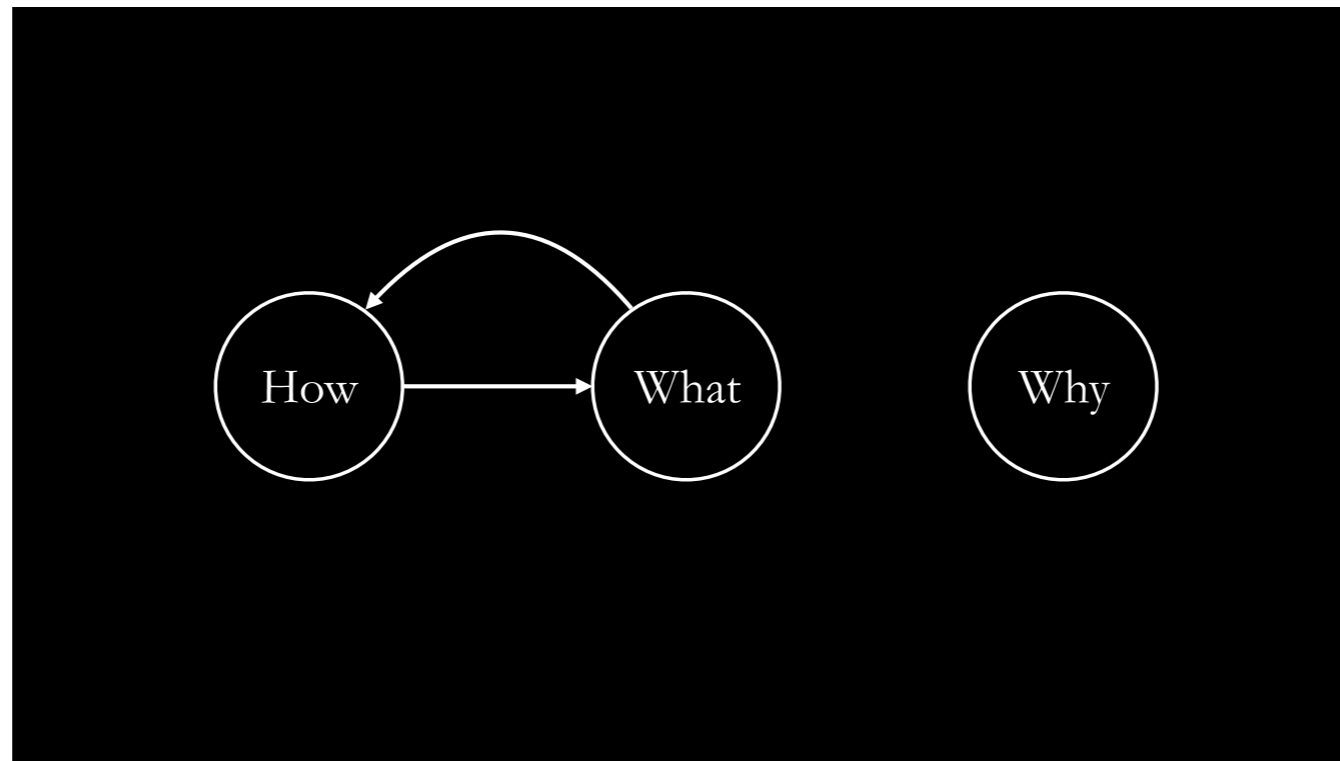
So we should test our packages behaviour via their public api



And even if you disagree about testing, you will agree that the public API of a package says “this is what I do, (click) not this is how I do it” (click)

When you have new behaviour that you want to guarantee, you should write a test

When you add new functionality to a public api, you should assert you’ve tested all its behaviour lest people begin to rely on `_incidental_` behaviour



So now we know what to test, lets step back and talk about how to test again

(click)

# The unit of code is the package

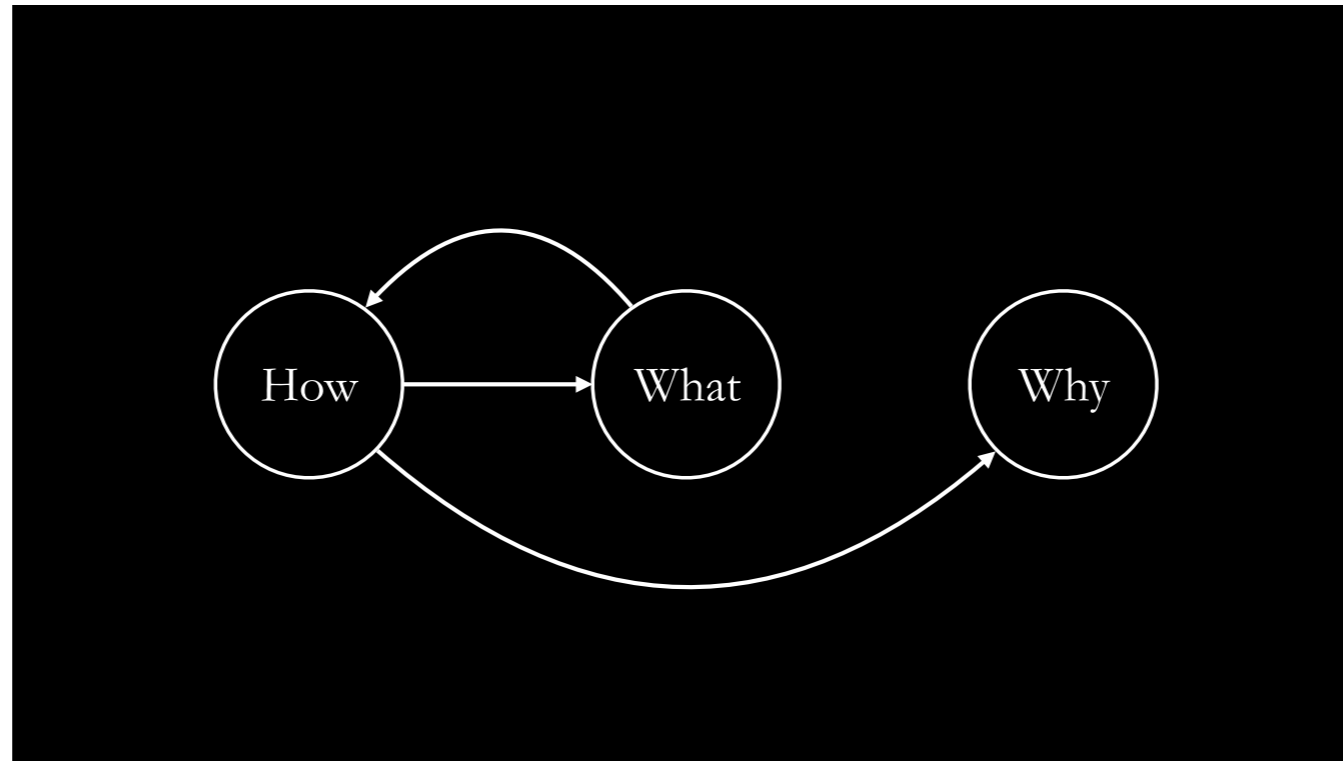
In Go the unit of code is the package. You only need to test the behaviour of your package that can be observed.

Code coverage is your guide. If there are branches that cannot be covered via the public API, delete that code.

When you refactor, use coverage to tell you where you need to add tests. Beware Hyrum's Law.

If you have high coverage, consider adding fuzz testing to check that you've covered all the edge cases.

As you've probably figured out, the big idea is the unit of code in Go is the package.



We've talked about how to test. We've talked about what to test. We've come back to How and said you should concentrate on testing your public API

(click)

The final part of this presentation is to ask the question, why test at all.

Even if you don't, *someone* will test  
your software

Now, I'm sure none of us think that software should be delivered without being tested first.

Even if that were true, your customers are going to test it, or at least use it, and so if nothing else, it would be good to discover any issues with the code before your customers, if not for the reputation of your company, at least professional pride

So, if we agree that software should be tested, the question is more, who should do that testing.

**The majority of testing should be  
done by development teams**

I argue that the majority of the testing should be done by development groups—us, the people in this room.



Testing should be, to the largest extent possible, automated

Moreover, testing should be automated, and thus the majority of these tests should be unit style tests.

## Just to be clear...

I am *not* saying you shouldn't write integration, function, or end to end tests.

I am also *not* saying that you shouldn't hire QA or integration test engineers.

To be clear, I am not saying you shouldn't write integration, functional, or end to end tests.

I am not saying that you shouldn't have a QA group, or integration test engineers, although I'd be prepared to place a small bet on two facts

## However ...

Nobody who considers themselves in a pure quality assurance role is in this room today.

Even for the largest companies present today, few of you have a dedicated QA group.

I'm willing to bet that there is nobody who considers themselves in a pure QA role in the audience today.

Is there anyone who considers their role to be purely quality or testing?

Thank you for coming, You're doing god's work. Come see me afterwards and I'll give you a prize.

However, I think my point stands, there are some big Chinese software companies represented here but the majority of you are in SWE or SRE. The heady days of 1.5 to 2 QA engineers to each software engineer of the 90's are long gone, they have been replaced by automated testing.

**Manual testing should not be the majority of your testing because manual testing is  $O(n)$**

Ok, so if we, as individual contributors are expected to test the software we write, why do we need to automate it? Why is a manual testing plan not good enough?

Manual testing of software or manual verification of a defect does not scale. As the number of manual tests grows engineers are tempted to skip them, or only execute the scenarios they think are involved. Manual testing is expensive, and boring, 99.9% of the tests that passed last time will pass again.

This means that your first response when given either a bug to fix or a feature to implement, should be to write a failing test. This doesn't need to be a unit test, but it should be an automated test. But once you've fixed the bug or added the feature you have the test case to prove it works — and you check them in together.

## Tests are a critical component of making sure you can always ship your master branch

As a development team, you are judged on your ability to deliver working software to the business.

Your super power is at any time anyone on the team should be confident that the master branch of your code is shippable—it may not have all the desired features, but what features are there should work as expected.

This means at any time you can deliver a release of your software to the business and the business can recoup its investment in your development R&D.

I cannot emphasise this enough. If you want the non technical parts of the business to believe you are heroes, you must never tell them “well, we can’t release for a few weeks because we’re in the middle of an important refactoring”

Again, I’m not saying you cannot refactor, but at every stage your product has to be shippable—your tests have to pass

The tdd doctrine of always being able to release your code simply by reverting the failing change is powerful for a delivery team.

## Tests lock in behaviour

We talked earlier that you want to test the behaviour of your APIs, not their implementations

Your tests are the contract about what your software does and does not do.

**Tests give you confidence to  
change someone else's code**

Lastly, and this is the biggest one for programmers working on a piece of code that has been through many hands.

Tests help you feel confident with change

Good test coverage promotes refactoring

## Take away suggestions

You should write tests.

You should write tests at the same time as you write your code.

Each Go package is a self contained unit.

Your tests should assert the observable behaviour of your package, not its implementation.

You should design your packages around their behaviour, not their implementation.

You should write tests

You should write tests in concert with the code

Each go package is a self contained unit — I don't say program because some of you will get hung up on the fact that most packages don't have a main() function.

I don't think that's an important distinction

Test your package's behaviour, not its implementation.

To do that you should write tests as if you were calling them from another package; if you cannot reach the code from another package, you cannot observe its behavior, you cannot make any assertions about its correctness.

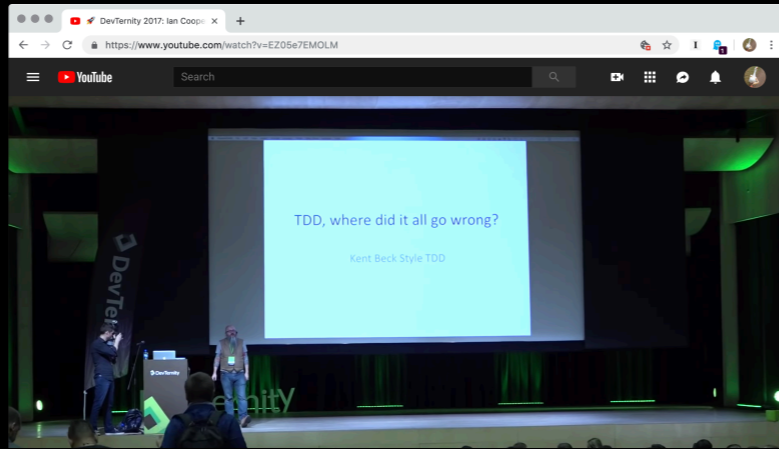
As soon as you start testing the internals you are coupling your tests to your implementation details. To change your implementation details, even though the behaviour is unchanged, you now have to change your tests.

You should test your code for high coverage of all public apis

You should delete any code you cannot reach from a public api

Design your packages around their behaviour, not the implementation. Describe your packages in terms of *what* they do, *not how* they do it





 Scan me