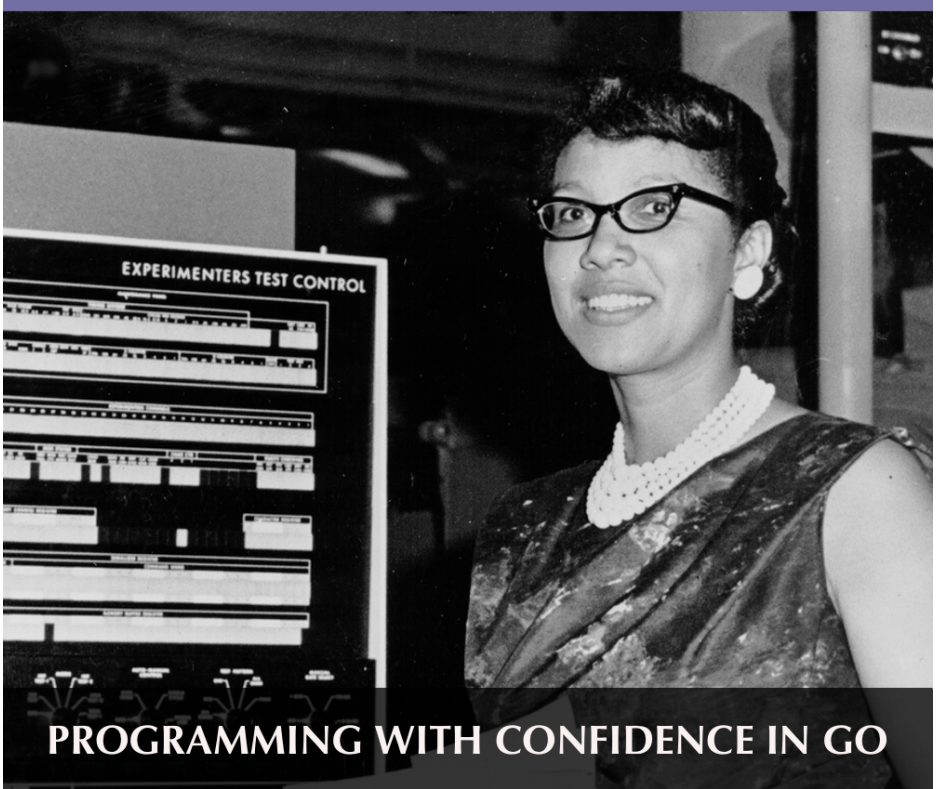
THE POWER OF GO
TESTS

"Engaging, friendly, informative, and snappy"
—Kevin Cunningham



PROGRAMMING WITH CONFIDENCE IN GO

JOHN ARUNDEL

Contents

Praise for *The Power of Go: Tests*

Brilliant. I read it with genuine pleasure. Well written, clear, concise, and effective.
—Giuseppe Maxia

I'd happily pay three times the price for John's books.
—Jakub Jarosz

The writing style is engaging, friendly, informative, and snappy.
—Kevin Cunningham

John's books are so packed with information, I learn something new every time I re-read them.
—Miloš Žižić

A great read—it's a treasure trove of knowledge on not just testing, but software design in general. Best of all, it's completely language-agnostic.
—Joanna Liana

I really enjoyed this book. The humour makes learning Go a lot more fun.
—Sean Burgoyne

This could get someone fired!
—David Bailey

The best introduction to mutation testing. John's writing style made me smirk, smile and vigorously nod my head as I was reading.
—Manoj Kumar

Introduction

> *When given the bug report, the developer responded, shaking his head vigorously, "Oh. That's not a bug."*
> *"What do you mean, 'that's not a bug'? It crashes."*
> *"Look," shrugged the developer. "It can crash or it can corrupt your data. Take your pick."*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

Programming is fun, even if it can sometimes feel a bit painful, like pounding nails into your head—and if it does, don't worry, it's not just you. We all feel that way from time to time, and it's not because we're bad programmers.

It's usually because we're trying to understand or untangle some complicated code that isn't working properly, and that's one of the hardest jobs a programmer will ever do. How do you avoid getting into a tangle like this in the first place?

Another not-so-fun thing is building large and complex projects from scratch, when we're running around trying to design the thing, figure out the requirements, and write the code, all without turning it into a big pile of spaghetti. How the heck do you do that?

But software engineers have learned a trick or two over the years. We've developed a way of programming that helps us organise our thinking about the product, guides us through the tricky early stages, leads us towards good designs, gives us confidence in the correctness of what we're doing, catches bugs before users see them, and makes it easier and safer for us to make changes in the future.

It's a way of programming that's more enjoyable, less stressful, and more productive than any other that I've come across. It also produces much better results. I'll share some of these insider secrets with you in this book. Let's get started!

1. Programming with confidence

> *It seemed that for any piece of software I wrote, after a couple of years I started hating it, because it became increasingly brittle and terrifying.*
>
> *Looking back in the rear-view, I'm thinking I was reacting to the experience, common with untested code, of small changes unexpectedly causing large breakages for reasons that are hard to understand.*
> —Tim Bray, "Testing in the Twenties"

When you launch yourself on a software engineering career, or even just a new project, what goes through your mind? What are your hopes and dreams for the software you're going to write? And when you look back on it after a few years, how will you feel about it?

There are lots of qualities we associate with good software, but undoubtedly the most important is that it be *correct*. If it doesn't do what it's supposed to, then almost nothing else about it matters.

Self-testing code

How do we know that the software we write is correct? And, even if it starts out that way, how do we know that the minor changes we make to it aren't introducing bugs?

One thing that can help give us confidence about the correctness of software is to write *tests* for it. While tests are useful whenever we write them, it turns out that they're especially useful when we write them *first*. Why?

The most important reason to write tests first is that, to do that, we need to have a clear idea of how the program should behave, from the user's point of view. There's some thinking involved in that, and the best time to do it is before we've written any code.

Why? Because trying to write code before we have a clear idea of what it should do is simply a waste of time. It's almost bound to be wrong in important ways. We're also likely to end up with a design which might be convenient from the point of view of the *implementer*, but that doesn't necessarily suit the needs of users at all.

Working test-first encourages us to develop the system in small *increments*, which helps prevent us from heading too far down the wrong path. Focusing on small, simple chunks of user-visible behaviour also means that everything we do to the program is about making it more valuable to users.

Tests can also guide us toward a good design, partly because they give us some experience of using our own APIs, and partly because breaking a big program up into small, independent, well-specified modules makes it much easier to understand and work on.

What we aim to end up with is *self-testing code*:

> *You have self-testing code when you can run a series of automated tests against the code base and be confident that, should the tests pass, your code is free of any substantial defects.*
>
> *One way I think of it is that as well as building your software system, you simultaneously build a bug detector that's able to detect any faults inside the system. Should anyone in the team accidentally introduce a bug, the detector goes off.*
> —Martin Fowler, "Self-Testing Code"

This isn't just because well-tested code is more reliable, though that's important too. The real power of tests is that they make developers happier, less stressed, and more productive as a result.

> *Tests are the Programmer's Stone, transmuting fear into boredom. "No, I didn't break anything. The tests are all still green." The more stress I feel, the more I run the tests. Running the tests immediately gives me a good feeling and reduces the number of errors I make, which further reduces the stress I feel.*
> —Kent Beck, "Test-Driven Development by Example"

The adventure begins

Let's see what writing a function test-first looks like in Go. Suppose we're writing an old-fashioned text adventure game, like Zork, and we want the player to see something like this:

> *Attic*
> The attics, full of low beams and awkward angles, begin here in a relatively tidy area which extends north, south and east. You can see here a battery, a key, and a tourist map.

Adventure games usually contain lots of different locations and items, but one thing that's common to every location is that we'd like to be able to list its contents in the form of a sentence:

```
You can see here a battery, a key, and a tourist map.
```

Suppose we're storing these items as strings, something like this:

```
a battery
a key
a tourist map
```

How can we take a bunch of strings like this and list them in a sentence, separated by commas, and with a concluding "and"? It sounds like a job for a function; let's call it `ListItems`.

What kind of test could we write for such a function? You might like to pause and think about this a little.

One way would be to call the function with some specific *inputs* (like the strings in our example), and see what it returns. We can predict what it should return when it's working properly, so we can compare that prediction against the actual result.

Here's one way to write that in Go, using the built-in `testing` package:

```
func TestListItems(t *testing.T) {
    t.Parallel()
    input := []string{
        "a battery",
        "a key",
        "a tourist map",
    }
    want := "You can see here a battery, a key, and a tourist map."
    got := game.ListItems(input)
    if want != got {
        t.Errorf("want %q, got %q", want, got)
```

```
    }
}
```

(Listing game/1)

Don't worry too much about the details for now; we'll deal with them later. The gist of this test is as follows:

1. We call the function `game.ListItems` with our test inputs.
2. We check the result against the expected string.
3. If they're not the same, we call `t.Errorf`, which causes the test to fail.

Note that we've written this code as though the `game.ListItems` function already *exists*. It doesn't. This test is, at the moment, an exercise in imagination. It's saying *if* this function existed, here's what we think it should return, given this input.

But it's also interesting that we've nevertheless made a number of important design decisions as an unavoidable part of writing this test. First, we have to *call* the function, so we've decided its name (`ListItems`), and what package it's part of (`game`).

We've also decided that its parameter is a slice of strings, and (implicitly) that it returns a single result that is a string. Finally, we've encoded the exact behaviour of the function into the test (at least, for the given inputs), by specifying exactly what the function should produce as a result.

> *The original description of test-driven development was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output.*
> *When describing this to older programmers, I often hear, "Of course. How else could you program?"*
> —Kent Beck

Naming something and deciding its inputs, outputs, and behaviour are usually the hardest decisions to make about any software component, so even though we haven't yet written a single line of code for `ListItems`, we've actually done some pretty important *thinking* about it.

And the mere fact of writing the test has also had a significant influence on the *design* of `ListItems`, even if it's not very visible. For example, if we'd just gone ahead and written `ListItems` first, we might well have made it print the result to the terminal. That's fine for the real game, but it would be difficult to test.

Testing a function like `TestItems` requires *decoupling* it from some specific output device, and making it instead a *pure function*: that is, a function whose result is deterministic, depends on nothing but its inputs, and has no side-effects.

Functions that behave like this tend to make a system easier to understand and reason about, and it turns out that there's a deep synergy between *testability* and good design, which we'll return to later in this book.

Verifying the test

So what's the next step? Should we go ahead and implement `ListItems` now and make sure the test passes? We'll do that in a moment, but there's a step we need to take first. We need some feedback on whether the *test* itself is correct. How could we get that?

It's helpful to think about ways the test could be *wrong*, and see if we can work out how to catch them. Well, one major way the test could be wrong is that it might not fail when it's supposed to.

Tests in Go pass by default, unless you explicitly make them fail, so a test function with no code at all would always pass, no matter what:

```
func TestAlwaysPasses(t *testing.T) {}
```

That test is so obviously useless that we don't need to say any more. But there are more subtle ways to accidentally write a useless test. For example, suppose we mistakenly wrote something like this:

```
if want != want {
    t.Errorf("want %q, got %q", want, got)
}
```

A value always equals itself, so this `if` statement will never be true, and the test will never fail. We might spot this just by looking at the code, but then again we might not.

I've noticed that when I teach Go to my students, this is a concept that often gives them trouble. They can readily imagine that the function itself might be wrong. But it's not so easy for them to encompass the idea that the *test* could be wrong. Sadly, this is something that happens all too often, even in the best-written programs.

Until you've seen the test fail as expected, you don't really have a test.

So we can't be *sure* that the test doesn't contain logic bugs unless we've seen it fail when it's supposed to. When *should* the test fail, then? When `ListItems` returns the wrong result. Could we arrange that? Certainly we could.

That's the next step, then: write just enough code for `ListItems` to return the wrong result, and verify that the test fails in that case. If it doesn't, we'll know we have a problem with the test that needs fixing.

Writing an incorrect function doesn't sound too difficult, and something like this would be fine:

```
func ListItems(items []string) string {
    return ""
}
```

Almost everything here is dictated by the decisions we already made in the test: the function name, its parameter type, its result type. And all of these need to be there in

order for us to call this function, even if we're only going to implement enough of it to return the wrong answer.

The only real choice we need to make here, then, is what actual result to return, remembering that we want it to be *incorrect*.

What's the simplest incorrect string that we could return given the test inputs? Just the empty string, perhaps. Any other string would also be fine, provided it's not the one the test expects, but an empty string is the easiest to type.

Running tests with `go test`

Let's run the test and check that it does fail as we expect it to:

```
go test
```

```
--- FAIL: TestListItems (0.00s)
    game_test.go:18: want "You can see here a battery, a key, and
    a tourist map.", got ""
FAIL
exit status 1
FAIL    game    0.345s
```

Reassuring. We *know* the function doesn't produce the correct result yet, so we expected the test to detect this, and it did.

If, on the other hand, the test had *passed* at this stage, or perhaps failed with some different error, we would know there was a problem. But it seems to be fine, so now we can go ahead and implement `ListItems` for real.

Here's one rough first attempt:

```
func ListItems(items []string) string {
    result := "You can see here"
    result += strings.Join(items, ", ")
    result += "."
    return result
}
```

(Listing game/1)

I really didn't think too hard about this, and I'm sure it shows. That's all right, because we're not aiming to produce elegant, readable, or efficient code at this stage. Trying to write code from scratch that's both correct *and* elegant is pretty hard. Let's not stack the odds against ourselves by trying to multi-task here.

In fact, the only thing we care about right *now* is getting the code correct. Once we have that, we can always tidy it up later. On the other hand, there's no point trying to beautify code that doesn't work yet.

> *The goal right now is not to get the perfect answer but to pass the test. We'll make our sacrifice at the altar of truth and beauty later.*
> —Kent Beck, "Test-Driven Development by Example"

Let's see how it performs against the test:

```
--- FAIL: TestListItems (0.00s)
    game_test.go:18: want "You can see here a battery, a key, and
    a tourist map.", got "You can see herea battery, a key, a
    tourist map."
```

Well, that looks *close*, but clearly not exactly right. In fact, we can improve the test a little bit here, to give us a more helpful failure message.

Using `cmp.Diff` to compare results

Since part of the result is correct, but part isn't, we'd actually like the test to report the *difference* between `want` and `got`, not just print both of them out.

There's a useful third-party package for this, `go-cmp`. We can use its `Diff` function to print just the differences between the two strings. Here's what that looks like in the test:

```
func TestListItems(t *testing.T) {
    t.Parallel()
    input := []string{
        "a battery",
        "a key",
        "a tourist map",
    }
    want := "You can see here a battery, a key, and a tourist map."
    got := game.ListItems(input)
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing game/2)

Here's the result:

```
--- FAIL: TestListItems (0.00s)
    game_test.go:20:   strings.Join({
                "You can see here",
        -       " ",
                "a battery, a key,",
        -       " and",
```

```
            " a tourist map.",
        }, "")
```

When two strings differ, `cmp.Diff` shows which parts are the same, which parts are only in the first string, and which are only in the second string.

According to this output, the first part of the two strings is the same:

```
"You can see here",
```

But now comes some text that's only in the first string (`want`). It's preceded by a minus sign, to indicate that it's missing from the second string, and the exact text is just a space, shown in quotes:

```
-       " ",
```

So that's one thing that's wrong with `ListItems`, as detected by the test. It's not including a space between the word "here" and the first item.

The next part, though, `ListItems` got right, because it's the same in both `want` and `got`:

```
"a battery, a key,",
```

Unfortunately, there's something else present in `want` that is missing from `got`:

```
-       " and",
```

We forgot to include the final "and" before the last item. The two strings are otherwise identical at the end:

```
" a tourist map.",
```

You can see why it's helpful to show the *difference* between `want` and `got`: instead of a simple pass/fail test, we can see how close we're getting to the correct result. And if the result were very long, the diff would make it easy to pick out which parts of it weren't what we expected.

Let's make some tweaks to `ListItems` now to address the problems we detected:

```
func ListItems(items []string) string {
    result := "You can see here "
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/3)

A bit ugly, but who cares? As we saw earlier, we're not trying to write beautiful code at this point, only correct code. This approach has been aptly named "Shameless Green":

> *The most immediately apparent quality of Shameless Green code is how very simple it is. There's nothing tricky here. The code is gratifyingly easy to comprehend. Not only that, despite its lack of complexity this solution does extremely well.*
> —Sandi Metz & Katrina Owen, "99 Bottles of OOP: A Practical Guide to Object-Oriented Design"

In other words, shameless green code passes the tests in the simplest, quickest, and most easily understandable way possible. That kind of solution may not be the *best*, as we've said, but it may well be good enough, at least for now. If we suddenly had to drop everything and ship right now, we *could* grit our teeth and ship this.

So does `ListItems` work now? Tests point to yes:

```
go test
```

```
PASS
ok      game    0.160s
```

The test is passing, which means that `ListItems` is behaving correctly. That is to say, it's doing what we asked of it, which is to format a list of three items in a pleasing way.

New behaviour? New test.

Are we asking *enough* of `ListItems` with this test? Will it be useful in the actual game code? If the player is in a room with exactly three items, we can have some confidence that `ListItems` will format them the right way. And four or more items will probably be fine too.

What about just two items, though? From looking at the code, I'm not sure. It *might* work, or it might do something silly. Thinking about the case of *one* item, though, I can see right away that the result won't make sense.

The result of formatting a slice of *no* items clearly won't make sense either. So what should we do? We could add some code to `ListItems` to handle these cases, and that's what many programmers would do in this situation.

But hold up. If we go ahead and make that change, then how will we know that we got it right? We can at least have some confidence that we won't break the formatting for three or more items, since the test would start failing if that happened. But we won't have any way to know if our new code correctly formats two, one, or zero items.

We started out by saying we have a specific job that we want `ListItems` to do, and we defined it carefully in advance by writing the test. `ListItems` now does that job, since it passes the test.

If we're now deciding that, on reflection, we want `ListItems` to do *more*, then that's perfectly all right. We're allowed to have new ideas while we're programming: indeed, it would be a shame if we didn't.

But let's adopt the rule "new behaviour, new test". Every time we think of a new behaviour we want, we have to write a test for it, or at least extend an existing passing test so that it fails for the case we're interested in.

That way, we'll be forced to get our ideas absolutely clear before we start coding, just like with the first version of `ListItems`. And we'll also know when we've written *enough* code, because the test will start passing.

This is another point that I've found my students sometimes have difficulty with. Often, the more experienced a programmer they are, the more trouble it gives them. They're so used to just going ahead and writing code to solve the problem that it's hard for them to insert an extra step in the process: writing a new *test*.

Even when they've written a function test-first to start with, the temptation is then to start extending the behaviour of that function, without pausing to extend the test. In that case, just saying "New behaviour, new test" is usually enough to jog their memory. But it can take a while to thoroughly establish this new habit, so if you have trouble at first, you're not alone. Stick at it.

Test cases

We could write some new test functions, one for each case that we want to check, but that seems a bit wasteful. After all, each test is going to do exactly the same thing: call `ListItems` with some input, and check the result against expectations.

Any time we want to do the same operation repeatedly, just with different data each time, we can express this idea using a *loop*. In Go, we usually use the `range` operator to loop over some slice of data.

What data would make sense here? Well, this is clearly a slice of test *cases*, so what's the best data structure to use for each case?

Each case here consists of two pieces of data: the strings to pass to `ListItems`, and the expected result. Or, to put it another way, `input` and `want`, just like we have in our existing test.

One of the nice things about Go is that any time we want to group some related bits of data into a single value like this, we can just define some arbitrary `struct` type for it. Let's call it `testCase`:

```
func TestListItems(t *testing.T) {
    type testCase struct {
        input []string
        want  string
    }
    ...
```

(Listing game/4)

How can we refactor our existing test to use the new `testCase` struct type? Well, let's start by creating a slice of `testCase` values with just one element: the three-item case we already have.

```
...
cases := []testCase{
    {
        input: []string{
            "a battery",
            "a key",
            "a tourist map",
        },
        want:
        "You can see here a battery, a key, and a tourist map.",
    },
}
...
```

(Listing game/4)

What's next? We need to loop over this slice of cases using `range`, and for each case, we want to pass its `input` value to `ListItems` and compare the result with its `want` value.

```
...
for _, tc := range cases {
    got := game.ListItems(tc.input)
    if tc.want != got {
        t.Error(cmp.Diff(tc.want, got))
    }
}
```

(Listing game/4)

This looks very similar to the test we started with, except that most of the test body has moved inside this loop. That makes sense, because we're doing exactly the same thing in the test, but now we can do it repeatedly for multiple *cases*.

This is commonly called a *table test*, because it checks the behaviour of the system given a table of different inputs and expected results. Here's what it looks like when we put it all together:

```
func TestListItems(t *testing.T) {
    type testCase struct {
        input []string
        want  string
```

```
    }
    cases := []testCase{
        {
            input: []string{
                "a battery",
                "a key",
                "a tourist map",
            },
            want:
            "You can see here a battery, a key, and a tourist map.",
        },
    }
    for _, tc := range cases {
        got := game.ListItems(tc.input)
        if tc.want != got {
            t.Error(cmp.Diff(tc.want, got))
        }
    }
}
```

(Listing game/4)

First, let's make sure we didn't get anything wrong in this refactoring. The test should still pass, since it's still only testing our original three-item case:

```
PASS
ok      game    0.222s
```

Great. Now comes the payoff: we can easily add more cases, by inserting extra elements in the `cases` slice.

Adding cases one at a time

What new test cases should we add at this stage? We could add lots of cases at once, but since we feel pretty sure they'll all fail, there's no point in that.

Instead, let's treat each case as describing a new behaviour, and tackle one of them at a time. For example, there's a certain way the system should behave when given *two* inputs instead of three, and it's distinct from the three-item case. We'll need some special logic for it.

So let's add a single new case that supplies two items:

```
{
```

```
    input: []string{
        "a battery",
        "a key",
    },
    want: "You can see here a battery and a key.",
},
```

(Listing game/5)

The value of `want` is up to us, of course: what we want to happen in this case is a product design decision. This is what I've decided *I* want, with my game designer hat on, so let's see what `ListItems` actually does:

```
--- FAIL: TestListItems (0.00s)
    game_test.go:36:   strings.Join({
            "You can see here a battery",
        +   ",",
            " and a key.",
          }, "")
```

Not bad, but not perfect. It's inserting a comma after "battery" that shouldn't be there.

Now let's try to fix that. For three or more items, we'll always want the comma, and for two, one, or zero items, we won't. So the quickest way to get this test to pass is probably to add a special case to `ListItems`, when `len(items)` is less than 3:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) < 3 {
        return result + items[0] + " and " + items[1] + "."
    }
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/5)

Again, this isn't particularly elegant, nor does it need to be. We just need to write the minimum code to pass the current failing test case. In particular, we don't need to worry about trying to pass test cases we don't *have* yet, even if we plan to add them later:

Add one case at a time, and make it pass before adding the next.

The test passes for the two cases we've defined, so now let's add the one-item case:

```
{
    input: []string{
        "a battery",
    },
    want: "You can see a battery here.",
},
```

(Listing game/6)

Note the slightly different word order: "you can see here a battery" would sound a little odd.

Let's see if this passes:

```
--- FAIL: TestListItems (0.00s)
panic: runtime error: index out of range [1] with length 1
[recovered]
```

Oh dear. `ListItems` is now panicking, so that's even worse than simply failing. In the immortal words of Wolfgang Pauli, it's "not even wrong".

Quelling a panic

Panics in Go are accompanied by a stack trace, so we can work our way through it to see which line of code is the problem. It's this one:

```
return result + items[0] + " and " + items[1] + "."
```

This is being executed in the case where there's only one item (`items[0]`), so we definitely can't refer to `items[1]`: it doesn't exist. Hence the panic.

Let's treat the one-item list as another special case:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) == 1 {
        return "You can see " + items[0] + " here."
    }
    if len(items) < 3 {
        return result + items[0] + " and " + items[1] + "."
    }
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
```

```
    result += "."
    return result
}
```

(Listing game/6)

This eliminates the panic, and the test now passes for this case.

Let's keep going, and add the zero items case. What should we expect `ListItems` to return?

```
{
    input: []string{},
    want:  "",
},
```

(Listing game/7)

Just the empty string seems reasonable. We could have it respond "You see nothing here", but it would be a bit weird to get that message every time you enter a location that happens to have no items in it, which would probably be true for most locations.

Running this test case panics again:

```
--- FAIL: TestListItems (0.00s)
panic: runtime error: index out of range [0] with
length 0 [recovered]
```

And we can guess what the problem is without following the stack trace: if `items` is empty, then we can't even refer to `items[0]`. Another special case:

```
if len(items) == 0 {
    return ""
}
```

(Listing game/7)

This passes.

Refactoring

We saw earlier that it doesn't matter how elegant the code for `ListItems` looks, if it doesn't do the right thing. So now that it *does* do the right thing, we're in a good place, because we have options. If we had to ship right now, this moment, we could actually do that. We wouldn't be delighted about it, because the code is hard to read and maintain, but *users* don't care about that. What *they* care about is whether it solves their problem.

But maybe we don't have to ship right now. Whatever extra time we have in hand, we can now use to refactor this correct code to make it nicer. And, while there's always a risk of making mistakes or introducing bugs when refactoring, we have a safety net: the test.

The definition of "refactoring", by the way, is changing code without changing its *behaviour* in any relevant way. Since the test defines all the behaviour we consider relevant, we can change the code with complete freedom, relying on the test to tell us the moment the code starts *behaving* differently.

Since we have four different code paths, depending on the number of input items, we can more elegantly write that as a `switch` statement with four cases:

```
func ListItems(items []string) string {
    switch len(items) {
    case 0:
        return ""
    case 1:
        return "You can see " + items[0] + " here."
    case 2:
        return "You can see here " + items[0] + " and " +
            items[1] + "."
    default:
        return "You can see here " +
            strings.Join(items[:len(items)-1], ", ") +
            ", and " + items[len(items)-1] + "."
    }
}
```

(Listing game/8)

Did we break anything or change any behaviour? No, because the test still passes. Could we have written `ListItems` from the start using a `switch` statement, saving this refactoring step? Of course, but we've ended up here anyway, just by a different route.

In fact, all good programs go through at least a few cycles of refactoring. We shouldn't even try to write the final program in a single attempt. Instead, we'll get much better results by aiming for *correct* code first, then iterating on it a few times to make it clear, readable, and easy to maintain.

> *Writing is basically an iterative process. It is a rare writer who dashes out a finished piece; most of us work in circles, returning again and again to the same piece of prose, adding or deleting words, phrases, and sentences, changing the order of thoughts, and elaborating a single sentence into pages of text.*
> —Dale Dougherty & Tim O'Reilly, "Unix Text Processing"

Well, that was easy

No doubt there's more refactoring we could do here, but I think you get the point. We've developed some correct, reasonably readable code, with accompanying tests that completely define the behaviour users care about.

And we did it without ever having to really *think* too hard. There were no brain-busting problems to solve; we didn't have to invent any complicated algorithms previously unknown to computer science, or that you might be tested on in some job interview. We didn't use any advanced features of Go, just basic strings, slices, and loops.

That's a good thing. If code is hard to write, it'll be hard to understand, and even harder to debug. So we *want* the code to be easy to write.

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
> —Brian Kernigan & P.J. Plauger, "The Elements of Programming Style"

If we find that writing the code is hard, we'll reduce the scope of our ambition so that we're solving some simpler problem instead. We'll keep simplifying the problem in this way until the code becomes easy and obvious. Then we can gradually build back up to the real problem we started with.

At every stage in this process, we had the confidence that comes from having a good test. It caught our occasional missteps, showed us from moment to moment how close we were getting to the correct solution, and what still remained to be done.

As we generated new ideas during development, it was easy to add them as new test cases, and we only had to do a very little work to make them pass. Once we had all the cases passing, we were able to confidently refactor the entire function, without worrying about breaking anything or introducing subtle bugs.

If you're not used to this way of programming with tests, it might all seem a bit long-winded for a relatively simple function. But the process is a lot quicker to *do* than it is to explain step by step.

In a real-life programming situation, it would probably take me a couple of minutes to write this test, two or three minutes to get it passing, and maybe another couple of minutes to refactor the code. Let's generously say ten minutes is a reasonable time to build something like `ListItems` using this workflow.

Could we do it faster if we didn't bother about the test? Perhaps. But *significantly* faster? I doubt it. To write a working `ListItems` from scratch that handles all these cases would take me a good ten minutes, I think. Indeed, it would actually be harder work, because I'd have to *think* very carefully at each stage and painstakingly work out whether the code is going to produce the correct result.

Even then, I wouldn't be completely confident that it was correct without seeing it *run* a few times, so I'd have to go on and write some throwaway code just to call `ListItems`

with some inputs and print the result. And it probably *wouldn't* be correct, however carefully and slowly I worked. I would probably have missed out a space, or something. So I'd have to go back and fix that.

> *It will feel slow at first. The difference is, when we're done, we're really done. Since we're getting closer to really done, the apparent slowness is an illusion. We're eliminating most of that long, painful test-and-fix finish that wears on long after we were supposed to be done.*
> —Ron Jeffries, "The Nature of Software Development"

In other words, the test-first workflow isn't slow at all, once you're familiar with it. It's quick, enjoyable, and productive, and the result is correct, readable, self-testing code.

Watching an experienced developer build great software fast, guided by tests, can be a transformative experience:

> *I grew a reporting framework once over the course of a few hours, and observers were absolutely certain it was a trick. I must have started with the resulting framework in mind. No, sorry. I've just been test driving development long enough that I can recover from most of my mistakes faster than you can recognize I've made them.*
> —Kent Beck, "Test-Driven Development by Example"

Sounds good, now what?

Maybe you're intrigued, or inspired, by the idea of programming with confidence, guided by tests. But maybe you still have a few questions. For example:

- How does Go's `testing` package work? How do we write tests to communicate intent? How can we test error handling? How do we come up with useful test data? What about the test cases we didn't think of?
- What if there are bugs still lurking, even when the tests are passing? How can we test things that seem untestable, like concurrency, or user interaction? What about command-line tools?
- How can we deal with code that has too many dependencies, and modules that are too tightly coupled? Can we test code that talks to external services like databases and network APIs? What about testing code that relies on time? And are mock objects a good idea? How about assertions?
- What should we do when the codebase has no tests at all? For example, legacy systems? Can we refactor and test them in safe, stress-free ways? What if we get in trouble with the boss for writing tests? What if the existing tests are no good? How can we improve testing through code review?
- How should we deal with tests that are optimistic, persnickety, over-precise, redundant, flaky, failing, or just slow? And how can tests help us improve the design of the system overall?

Well, I'm glad you asked. Enjoy the rest of the book.

2. Tools for testing

> *In every mainstream programming language, there are dozens of software packages intended for test construction. It is almost enough to convince programmers that to write effective tests they must acquire a whole new set of skills.*
>
> *Go's approach to testing can seem rather low-tech in comparison. It relies on one command, `go test`, and a set of conventions for writing test functions that go test can run. The comparatively lightweight mechanism is effective for pure testing, and it extends naturally to benchmarks and systematic examples for documentation.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

What tools do we have available in the standard `testing` package, and how can we use them to write effective tests?

If you've written tests in Go before, much of this will be familiar to you, but there may be one or two things you haven't come across.

Go's built-in testing facilities

> *If you cannot explain a program to yourself, the chance of the computer getting it right is pretty small.*
> —Bob Frankston, quoted in Susan Lammers, "Programmers at Work"

Let's assume that you're now at the point where you can explain to the computer exactly what you want. How do we express that as a test in Go?

An automated test is just another kind of computer program. So, in principle, we could just write some Go function that calls the system under test with some inputs, checks the results, and reports whether or not they're correct.

To streamline this process, though, the Go standard library provides a package named `testing` that we can import and use in our test code. The facility to build, run, and report test results is also built into the Go tool itself. And it's delightfully simple and easy to use, especially if you're used to more complicated test frameworks in other languages.

So how do we write something so that Go recognises it as a test? Let's try.

The first step towards making something a test is choosing the name of the file we put it in. Go looks for test code in files whose names end with `_test.go`. No other files will be considered when running tests. So the `_test` in the name is magical to Go.

It's convenient, but not mandatory, to put source files containing test code in the same folder as the package they're testing. For example, suppose we have some file `service.go` that implements a package `service`.

Let's assume we've initialised a new Go module, by running `go mod init service` in the containing folder, creating a `go.mod` file that identifies the module. What else needs to be present for us to be able to run some tests?

We have a `service.go` file, and there may or may not be any functions in it, but to be valid Go it must at least begin with a `package` declaration:

```
package service
```

So, where should we put our tests? Let's create a new file in this folder named `service_test.go`. So now the folder looks like this:

```
service/
    go.mod
    service.go
    service_test.go
```

One nice thing about using Go's `testing` package, as opposed to rolling our own, is that test code is ignored when we build the system for release. So there's no need to worry about bloating the executable binary with testing code that doesn't benefit users. We can write as many tests as we want, without adding even a byte to the production system.

All Go files, as we've just seen, have to start with a `package` declaration, saying what package they belong to. What package should we declare for our test file?

We could use package `service`, and that would work, but let's use `service_test` instead, to make it clear that this is *test* code, not part of the system itself:

```
package service_test
```

And we'll be using the standard library `testing` package, so we'll need to import that:

```
import "testing"
```

Writing and running tests

Fine. Now we're ready to write some actual tests. How do we do that?

When we eventually come to run these tests, Go will start by looking for all files ending in `_test.go` and inspecting them to see if they contain tests. What is it looking for, then?

Go tests are ordinary functions, defined with the `func` keyword, but what makes them special is that their names start with the word `Test`. For example:

```
func TestNewReturnsValidService(t *testing.T)
```

It doesn't matter to Go what the rest of the function name is, though it serves to identify the test (and we can use that to provide a concise description of the behaviour, as we'll see). It just needs to start with the word `Test...` for Go to identify it as a test.

We also have to take a single parameter, conventionally named `t`, which is a pointer to a `testing.T`, and we'll see what that's for in a moment. But other than that, there's nothing else we need to do to have a valid test. The function body can be completely empty:

```
func TestAlwaysPasses(t *testing.T) {
    // It's all good.
}
```

As simple-minded as it seems, this is not only a valid test, but it also passes:

```
go test
```

```
PASS
ok      service 0.158s
```

As we saw in the first chapter, if we don't explicitly tell a test to fail, it will be considered to have passed. No news is good news, as far as tests are concerned.

And that's how we write and run tests. You can see what Donovan and Kernighan mean when they say Go's testing system is relatively lightweight and low-tech. You don't have

to write lots of boilerplate, or download special libraries, or do any code generation or anything like that.

All you need to do is put your code in a file whose name ends with `_test.go`, import the `testing` package, and write a function whose name starts with `Test`. Congratulations, you're programming with tests!

They're not very useful tests yet, of course, but we have to start somewhere. Suppose we wanted to write a test that always fails, instead of one that always passes? How could we do that?

We saw that test functions take this mysterious parameter `t *testing.T`. If we try to write a test without it, we get a compiler error:

```
func TestAlwaysPasses() {}
```

```
wrong signature for TestAlwaysPasses, must be: func
TestAlwaysPasses(t *testing.T)
```

So that `t` is important to tests, in some way. What does it do? Well, one thing it can do is cause the test to fail, if we call its `Error` method:

```
func TestAlwaysFails(t *testing.T) {
    t.Error("oh no")
}
```

Let's see what that looks like:

```
go test
```

```
--- FAIL: TestAlwaysFails (0.00s)
    service_test.go:8: oh no
FAIL
exit status 1
FAIL    service 0.292s
```

Interpreting test output

There are a few interesting things about this output. First, we saw a big `FAIL` message telling us that some test has failed. Then we saw the *name* of the test that failed, and how long it took to fail (almost no time, which is hardly surprising).

Next, we have the name of the source file containing the test (`service_test.go`), and the exact line number where the failure occurred (line 8). In this case, that's the call to `t.Error`. Most IDEs will make this text clickable, so that you can jump straight to the offending line in the editor, which is useful.

After that, we see the message `oh no`, which is what we passed to `t.Error` in the code. This is where we can explain to the user (that is, the person running the tests) exactly what went wrong, in as much detail as they need to fix the problem.

Finally, we see the results for this package as a whole (`FAIL`), and for the module that contains the package (`FAIL`). If any individual test fails, Go considers the package it's testing to have failed too, and also the module it belongs to.

A test that always fails is not much more use than one that always passes. What we need is to be able to *decide* whether or not to fail the test, based on some condition.

That condition depends on what behaviour we're testing, of course, so let's say the behaviour we're interested in is that we can detect that the service is running. We might suppose some `Running` function, for example, which returns `true` when the service is running, and `false` when it isn't. How could we test that?

Something like this might work:

```
func TestRunningIsTrueWhenServiceIsRunning(t *testing.T) {
    t.Parallel()
    service.Start()
    if !service.Running() {
        t.Error(false)
    }
}
```

(Listing `service/1`)

Finally, a useful test!

Since we haven't implemented the code for `Start` or `Running` yet, we feel instinctively that this test should fail. If it passes, well, either there's something wrong with the logic of the test, or the problem was just easier than we thought.

A phrase you'll often hear in this context is "red, green, refactor". It's a concise way of describing a three-step cycle of software development. Here, "red" means a failing test, while "green" means a passing one.

In other words, following the red-green-refactor cycle means that we start by writing a test for the behaviour. This test will fail, of course, since we haven't implemented the behaviour yet. That's "red". We're not there yet, since the test isn't even *compiling*, but let's see how far we've got:

```
go test
```

```
# service_test [service.test]
./service_test.go:8:2: undefined: service
./service_test.go:9:6: undefined: service
FAIL    service [build failed]
```

Well, that's fair enough. We've been blithely calling a function in the `service` package, but as with any third-party Go package, we need to import it before we can use it:

```
import "service"
```

How about now?

```
go test
```

```
# service_test [service.test]
./service_test.go:9:10: undefined: service.Start
./service_test.go:10:14: undefined: service.Running
FAIL    service [build failed]
```

A different error message means progress. This time it's because, although we successfully imported the `service` package, Go hasn't found any definition for functions called `Start`, or `Running`. And no wonder, because we haven't written them.

The `Start` function isn't what we're interested in here, so let's assume we've written it. It starts the service, waits to confirm that it's running, and then returns. So, assuming this is independently tested, we can rely on the fact that if `Start` returns, then the service is really running.

The test will then proceed, by calling the function we're actually *testing*, which is `Running`.

The "magic package": testing functions that don't exist

If you use an IDE or code editor that automatically highlights problems, you may find that it's actually working against you here. For example, the calls to `Start` and `Running` may be highlighted in red, with an angry squiggle and the same compiler error that we just saw. If you're not used to writing code test-first, this may cause you some anxiety.

In effect, the IDE has carefully trained us to believe that every time we write something that causes a squiggly red line, we've done something wrong: we should stop and fix it before continuing. Our own editor is exerting psychological pressure on us not to write tests! That's most unhelpful.

It would be nice if there were a way to tell the editor "Hey, I'm working test-first over here! Quit red-lining my stuff until I'm ready." But until that happens, the only thing we can do is keep calm and finish the test before dealing with the red lines.

It takes a certain amount of imagination, then, to call functions that don't exist. But once you've got the hang of it, you can use this technique to design APIs that seem to fit beautifully with the code that calls them.

> *When I write a test, half that test is not just to validate the code is working as expected, it's to validate the API is intuitive and simple to use.*
> —Bill Kennedy, "Package Oriented Design"

I sometimes call this the *magic package* approach: write your test, or calling code, as though there were some wonderful, magical package that does whatever you want. If you come to some difficult problem, or just some tedious bit-shuffling paperwork, just assume the magic package will take care of it, and glide straight over the difficulty.

If it would be convenient to have some function that takes the exact inputs you have available, and returns just the result you want, wave your coding wand and imagine it. If there's awkward paperwork to be done, and you wish there were some handy abstraction that will take care of it for you, use a suitable function in the magic package.

Once you're done imagining, and you've written some clear, straightforward code that solves the problem elegantly using the hypothetical magic package, all you need to do then is implement the package. With a simple transfiguration spell, we've turned a tricky design problem into an easier coding problem.

The important thing is to *write the code you'd want to read*, and worry about the rest later:

> *We want each test to be as clear as possible an expression of the behavior to be performed by the system or object. While writing the test, we ignore the fact that the test won't run, or even compile, and just concentrate on its text; we act as if the supporting code to let us run the test already exists.*
>
> *When the test reads well, we then build up the infrastructure to support the test. We know we've implemented enough of the supporting code when the test fails in the way we'd expect, with a clear error message describing what needs to be done. Only then do we start writing the code to make the test pass.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

Let's write, then, in the `service.go` file, the absolute minimum we need to get this test to compile. Something like this:

```
func Running() bool {
    return false
}
```

(Listing `service/1`)

Remember, we're not trying to get this functionality working yet. First we need to see the test fail. And we can't see that until it compiles. So we're doing the minimum necessary to make that happen, however silly or trivial the code might be. We might call this, after Metz and Owen, "shameless red".

Now it should compile, so let's run it again. But before we do, let's do a thought experiment I call "being the test". What will the result be, exactly? Fail, yes, but at which line of the test? With what message?

Validating our mental models

The reason this is useful is that when we're programming, we're essentially building a mental model of the solution, and developing that model inside the computer at the same time. If the model in our heads doesn't match the computer's, things will go wrong very quickly, and in a deeply puzzling way.

So a crucial step before running any test is to consult your *own* model of the software and ask it what the result will be. Then, check that the computer's model agrees. If not, then *someone's* model is wrong (maybe both). If you see a different failure than you expected, *stop*. Think. Is the test expecting the wrong thing, or are you?

Once you have a reasoned prediction of the result, run the test and find out whether it matches your forecast:

```
go test
```

```
--- FAIL: TestRunningIsTrueWhenServiceIsRunning (0.00s)
    service_test.go:11: false
```

And there we are. We're at "red". Now we can go ahead and fill in the code for `Running`, still in the most shameless and expedient way possible, until the test passes: that will be the "green" stage. Finally, "refactor": we'll indulge our inner engineer, and modify the code as much as necessary to make it clear, readable, and efficient.

If you ever get a bit lost, or aren't sure how to proceed, "red, green, refactor" is a great guide for getting back on track. We'll follow this cycle in everything we do in this book.

You know how to write tests, how to run them, and most importantly of all, how to make them fail. And that's all you need in order to program with tests in Go. This simplicity is refreshing, especially if you're used to languages where tests require a lot of boilerplate and scaffolding. Even some alternative Go testing frameworks have somewhat intimidating APIs: `testify`, for example, has many *hundreds* of methods just in its `assert` package.

Fortunately, you won't need any of those, because you can write all the tests you'll ever need using just `t.Error`. But there are a few other little tricks that the `testing` package has up its sleeve, if we choose to use them. Let's take a quick tour of what's available.

Concurrent tests with `t.Parallel`

Go is justly praised for its first-class concurrency support, and the `testing` package is no exception. By calling `t.Parallel()`, any test can declare itself suitable to run concurrently with other tests:

```
func TestCouldTakeAWhile(t *testing.T) {
    t.Parallel()
    ... // this could take a while
}
```

If you have more than one CPU, which is likely, then you can speed up your tests by running them in parallel like this. And since the point of tests is *fast feedback*, that's an extremely good idea. Let us therefore adopt this resolution: *All tests should be parallel.*

In other words, call `t.Parallel()` in every test, unless there's some really good reason why you can't.

Failures: `t.Error` and `t.Errorf`

If we're writing really short, focused tests, with well-chosen names, then we don't need to say much more in the failure message. We'll discuss this again in more detail in the chapter on "communicating with tests" later in this book. For now, let's say simply that a good test *name* is a sentence describing the expected behaviour:

> Running returns `true` when service is running.

Supposing this test fails, then, the failure message doesn't need to be very elaborate. The mere fact that a test called "`Running` is `true` when service is running" is *failing* tells the reader everything they need to know about the current behaviour of `Running`: it must be returning `false`. We don't need to say much more than that in the failure message itself.

But sometimes there are other possible outcomes, and it may be useful to know which one we're seeing. For example, since many Go functions return `error` along with some data value, it's as important to test that `error` result as any other. We might write something like:

```
if err != nil {
    t.Error(err)
}
```

Conveniently, we can pass any type of value as the argument to `Error`. And not just any type, but any *number* of values of any type. We could use this to show, for example, both what we wanted and what we actually *got*:

```
want := 4
got := Add(2, 2)
if want != got {
    t.Error("want", want, "got", got)
}
```

This gives a failure message like the following:

```
want 4 got 0
```

It would have been *sufficient*, if not exactly helpful, to just say something like "failed", or "wrong answer", but this is a good deal better. The exact value of the wrong answer can be a useful clue to *why* it's wrong.

If we need more precise control over the failure data, then we can use `t.Errorf` instead. This puts all the string-formatting power of the `fmt` package at our command.

Whereas `Error` just takes a list of data values, `Errorf` takes a *format* string (hence the `f`), followed by the data values it describes. For example:

```
want := "hello"
got := Greeting()
if want != got {
    t.Errorf("want %q, got %q", want, got)
}
```

Notice that we used the `%q` (quoted string) format verb here to make the data more visible. If the result is unexpectedly empty, for example, it's easier to see that when the string is quoted, like this:

```
want "hello", got ""
```

Abandoning the test with `t.Fatal`

It's worth knowing that `t.Error` (or its format-savvy equivalent `t.Errorf`) marks the test as failed, but doesn't stop it. There may be more things to test, even though this one has failed, so the test will continue.

This can be useful. Here's an example where we check three things in succession, and use `t.Error` to report each one if it fails:

```
x, y, z := DoStuff()
if !x {
    t.Error("not x")
}
if !y {
    t.Error("not y")
}
if !z {
    t.Error("not z")
}
```

If all three conditions are true, the test would pass, but what happens if some of them are true and some false? In that case, we would see a failure message like this:

```
not x
not z
```

In a case like this it's likely useful to know exactly which combination of `x`, `y`, and `z` is the case, because that could give valuable clues to what's gone wrong.

> *A good test should not give up after one failure but should try to report several errors in a single run, since the pattern of failures may itself be revealing.*

—Alan Donovan & Brian Kernighan, "The Go Programming Language"

So we don't want to stop the test as soon as we see the first failure: instead, we want to check all the conditions, and produce a full report.

In other situations, though, it might be better to bail out of the test as soon as something has failed. For example, suppose we need to set up some test fixture, such as a data file. Opening a file can always fail, so we need to check that error:

```
f, err := os.Open("testdata/input")
if err != nil {
    t.Fatal(err)
}
```

If we can't open the file, then there's no point continuing with the test; attempting to read from `f` won't succeed, because it's `nil`. In that case, whatever else we do in this test will be a waste of time at best, and fail confusingly at worst.

Whatever we ultimately `want`, it certainly won't be what we `got`, but that's not because of a failure in the system under test. It's because we couldn't construct our test fixture correctly. In a situation like this, we want to fail the test *and* stop immediately, which is what `t.Fatal` does.

So while `Error` and `Fatal` are both ways of causing a test to fail, `Error` fails and continues, while `Fatal` fails and bails. If you're ever in doubt which one to use, ask yourself "Is there any useful information to be gained by continuing with the test?" If not, use `Fatal` (or its format-string equivalent, `Fatalf`).

Use `t.Error` when it's worth continuing with the test; use `t.Fatal` when it's not.

Writing debug output with `t.Log`

Speaking of useful information, sometimes we'd like the test to be able to output things that aren't failure messages. For example, intermediate values obtained during some multi-stage process:

```
want := "right"
got := StageOne()
got = StageTwo(got)
got = StageThree(got)
if want != got {
    t.Errorf("want %v, got %v", want, got)
}
```

If it turns out after `StageThree` that we don't have what we want, it may be difficult to work out why not. Where exactly did the process go wrong? Was it at `StageOne`, `Sta-`

geTwo, or StageThree? We don't have any information about that, and the failure message can't report it, because those intermediate results no longer exist.

If we happen to know exactly what the results of StageOne and StageTwo should be, then we can test for them in the usual way, and use Fatal to fail the test if they're incorrect. But if we can't know them in advance, it may still be helpful to see what they *were* in the event that the test fails.

This is where t.Log comes in useful:

```
got := StageOne()
t.Log("StageOne result", got)
got = StageTwo(got)
t.Log("StageTwo result", got)
```

Supposing the test passes, then we'll see no output at all: tests should be silent unless there's something to say. But when it *does* fail, we'll see the output of these calls to t.Log along with the failure message:

```
StageOne result right
StageTwo result wrong
want right, got wrong
```

Why not just use fmt.Println to print out these messages? Well, for one thing, this would print even when the test *passes*, and that's probably not what you want:

```
go test
```

```
StageOne result right
StageTwo result right
PASS
ok      service 0.168s
```

Tests should pass silently and fail loudly. We don't want to see this junk cluttering up our test output; if the tests are passing, that's all we want to know. So we don't use fmt to print log messages in tests; we use t.Log instead, so that the information only appears on failure.

Also, when tests are running in parallel, messages printed by fmt can appear confusingly out of sequence. It's hard to tell which test they came from. By contrast, t.Log messages are automatically collated with the results of the test that produced them.

Conveniently, only the log messages from failing tests are printed. So you can log as much as you like with t.Log in every test, and you won't be swamped with unwanted output when some other test fails.

Test flags: `-v` and `-run`

As we've seen, the `go test` command can take an argument specifying a package, or list of packages, to test. Most commonly, we want to test all the packages from the current directory downwards, which looks like this:

```
go test ./...
```

And there are some useful switches that alter the behaviour of the `go test` command. For example, the `-v` (for "verbose") flag prints out the names of each test as it runs, its results, and any log messages, regardless of failure status:

```
go test -v
```

```
=== RUN   TestMultiStageProcess
    service_test.go:24: StageOne result right
    service_test.go:26: StageTwo result right
--- PASS: TestMultiStageProcess (0.00s)
PASS
ok      service 0.153s
```

Some people *always* run their tests with `-v`. This can make it hard to see if any tests are failing. If running tests *always* produces several screens worth of output, you could easily miss an important message among the irrelevant bumf.

Verbose tests are tedious company, like verbose people. "Be quiet unless there's something to say" is the best advice in both cases. Running tests in verbose mode can be a useful way to learn about a new system, but when we're developing new behaviour, we want the tests' feedback to be brief, and to the point.

For efficiency, `go test` doesn't bother recompiling your code unless it's changed, so test results are cached for each of the packages you're testing. This helps keeps test suites fast, especially when you have a lot of packages.

If a particular package is unchanged since the last test run, you'll see its cached results, marked accordingly:

```
go test .
```

```
ok      service 0.177s
```

```
go test .
```

```
ok      service (cached)
```

Sometimes you may want to force running a test for some reason, even though its result is cached. In this case, you can use the `-count` flag to override the cache:

```
go test -count=1 .
```

```
ok      service 0.140s
```

It's usually a good idea to run *all* your tests, all the time, but sometimes you may want to run a single test in isolation. For example, while you're in the middle of a big refactoring, this will inevitably cause a few tests to fail, and it's wise to focus on fixing one at a time.

To run a single test in this way, use the `-run` flag with the test name:

```
go test -run TestRunningIsTrueWhenServiceIsRunning
```

In fact, the argument to the `-run` flag is a regular expression, so you can also use it to run a *group* of related tests, whose names match the expression you supply.

The easiest such expression is a substring that matches all the tests you want to run, like this:

```
go test -run TestDatabase
```

```
--- FAIL: TestDatabaseOperation1 (0.00s)
--- FAIL: TestDatabaseOperation2 (0.00s)
--- FAIL: TestDatabaseOperation3 (0.00s)
--- FAIL: TestDatabaseOperation4 (0.00s)
--- FAIL: TestDatabaseOperation5 (0.00s)
```

Assistants: `t.Helper`

Although some setup is usually necessary at the start of a test, this shouldn't be too long or complicated, or it will be in danger of obscuring the main point of the test.

If it's because there's a lot of paperwork involved in calling your function, then the test may be helping you to identify a design smell. Would real users have to do this paperwork too? If so, consider redesigning your API to eliminate or automate it.

But if the paperwork *can't* be eliminated, it can always be moved into another function. By creating a new abstraction to take care of this for us, we hide the irrelevant mechanics of the test setup. And by giving it an informative name, we also let the reader know what's going on:

```
func TestUserCanLogin(t *testing.T) {
    createTestUser(t, "Jo Schmo", "dummy password")
    ... // check that Jo Schmo can log in
}
```

There are probably many lines of uninteresting setup code concealed behind the call to `createTestUser`. By invoking an abstraction in this way, we hide irrelevant detail, and help to focus the reader's attention on what's really important in the test.

It seems likely that there could be some error while creating the test user, but checking and handling a return value from `createTestUser` would add clutter back into

the main test, where we don't want it. If we don't want `createTestUser` to return any errors it encounters, what should it do instead?

Well, note that we pass this test's `t` parameter as an argument to `createTestUser`. Armed with that, it can fail the test itself if it needs to. For example:

```
func createTestUser(t *testing.T, user, pass string) {
    t.Helper()
    ... // create user ...
    if err != nil {
        t.Fatal(err)
    }
}
```

Notice the call to `t.Helper` here. This marks the function as a test helper, meaning that any failures will be reported at the appropriate line in the *test*, not in this function:

```
--- FAIL: TestUserCanLogin (0.00s)
    user_test.go:8: some error creating user
```

Calling `t.Helper` effectively makes the helper function invisible to test failures. This makes sense, since we're usually more interested in knowing which *test* had a problem.

Since tests should tell a story, well-named helpers can make that story easier to read:

```
user := login(t, "Jo Schmo")
defer logout(t, user)
... // test something with 'user'
```

`t.TempDir` and `t.Cleanup`

If the test needs to create or write data to files, we can use `t.TempDir` to create a temporary directory for them:

```
f, err := os.Create(t.TempDir()+"/result.txt")
```

This directory is unique to the test, so no other test will interfere with it. When the test ends, this directory and all its contents will also be automatically cleaned up, which is very handy: it saves *us* having to do that.

When we do need to clean something up ourselves at the end of a test, we can use `t.Cleanup` to register a suitable function:

```
res := someResource()
t.Cleanup(func() {
    res.GracefulShutdown()
    res.Close()
```

```
})
... // test something with 'res'
```

The function registered by `Cleanup` will be called once the test has completed. `Cleanup` also works when called from inside test helpers:

```
func createTestDB(t *testing.T) *DB {
    db := ... // create db
    t.Cleanup(func() {
        db.Close()
    })
    return db
}
```

If we had instead written `defer db.Close()` here, then that call would happen when the *helper* returns, which isn't what we want. We need `db` to stay open so that the test can use it. But when the *test* is done, the cleanup func will be called to close `db`.

Tests are for failing

Some tests almost seem determined to overlook any potential problems and merely confirm the prevailing supposition that the system works. I think this is the wrong attitude. You *want* your tests to fail when the system is incorrect, that's the point. If a test can never fail, it's not worth writing.

For example, I recently came across something like the following:

```
func TestNewThing(t *testing.T) {
    t.Parallel()
    _, err := thing.NewThing(1, 2, 3)
    if err != nil {
        t.Fatal(err)
    }
}
```

(Listing `thing/1`)

What are we really testing here? If we were to make the writer's train of thought explicit, it might go something like this:

1. `NewThing` doesn't return an error.
2. Therefore, it works.

But is this really a valid deduction? What is it that `NewThing` is actually supposed to do, anyway? Let's go over the clues.

It looks like `NewThing` takes some arguments: `1`, `2`, and `3`. Presumably, these must affect the behaviour of `NewThing` in some way. If they don't, it seems pointless to take them as parameters.

So are just *any* values okay? Or are some values allowed and some not? In other words, does `NewThing` need to *validate* its arguments?

The fact that it returns `error` is a useful clue here. One of the possible errors we might get is that the arguments are invalid. If that *is* what's supposed to happen, then it's a shame, because this test doesn't actually test that.

But there's a more serious omission in this test. Presumably `NewThing` is supposed to actually *do* something, as well as merely not returning an error. We can guess from the name that it's a constructor for a type called `Thing`, which implies that it returns some value of type `*Thing`, perhaps, along with the error.

However, the test ignores that value completely, using the blank identifier (`_`). So it doesn't really test that `NewThing` does anything at all other than return a `nil` error. The purpose of the test is simply to confirm what the developer thinks they already know: that the function is correct.

> *Tests that are designed to confirm a prevailing theory tend not to reveal new information.*
> —Michael Bolton, "How Testers Think"

Why is this a problem, and what could we do about it?

Detecting useless implementations

One useful way to think about the value of a given test is to ask:

> What incorrect implementations would still pass this test?

Let's do that thought experiment now, and see if we can write a version of `NewThing` that's obviously incorrect, but satisfies the test.

Here's an easy one:

```
func NewThing(x, y, z int) (*Thing, error) {
    return nil, nil
}
```

(Listing `thing/1`)

If a test is to be useful at all, it should be able to detect a null implementation like this. If it can't, what is it really testing?

Could we extend this test to force `NewThing` to do something useful? What checks would we need to add?

One obvious improvement would be to *not* ignore the first result value. Instead, we could assign it to a variable, and test something about it. What could we test?

Well, if the result *is* a pointer to `Thing`, then we can at least check that it's not `nil`:

```
func TestNewThing(t *testing.T) {
    t.Parallel()
    got, err := thing.NewThing(1, 2, 3)
    if err != nil {
        t.Fatal(err)
    }
    if got == nil {
        t.Error("want non-nil *Thing, got nil")
    }
}
```

(Listing `thing/2`)

This is already much better than what we started with. But are we still missing something? In other words, can we still write a useless implementation of `NewThing` that passes this test?

You bet we can. Here's a simple one:

```
func NewThing(x, y, z int) (*Thing, error) {
    return &Thing{}, nil
}
```

(Listing `thing/2`)

Whoops. We've ignored `x`, `y`, and `z` altogether, and returned an *empty* `Thing` struct instead of `nil`. This is nevertheless enough to satisfy the test, and that's a problem.

Feeble tests

It turns out it's not enough to ask whether `NewThing` returns something that's not `nil`. It actually needs to be a *specific* non-nil value. Otherwise we could trivially pass the test by returning a pointer to *any* struct, including an empty one.

So what more searching question *should* we ask about the result struct? As we saw earlier, if the function takes `x`, `y`, and `z` as arguments, then it makes sense to assume that they should *affect* the resulting `Thing` in some way. But how?

The simplest answer might be that these argument values end up as fields on the result struct. Could we devise a way to test that?

Certainly. We know what the values of `x`, `y`, and `z` *are*, because we supplied them in the

first place. So an easy way to test that `NewThing` actually *used* them is to look at the result and ask if it contains our input values.

Here's an extended version of the test asking exactly that:

```
func TestNewThing(t *testing.T) {
    t.Parallel()
    x, y, z := 1, 2, 3
    got, err := thing.NewThing(x, y, z)
    if err != nil {
        t.Fatal(err)
    }
    if got.X != x {
        t.Errorf("want X: %v, got X: %v", x, got.X)
    }
    if got.Y != y {
        t.Errorf("want Y: %v, got Y: %v", y, got.Y)
    }
    if got.Z != z {
        t.Errorf("want Z: %v, got Z: %v", z, got.Z)
    }
}
```

(Listing thing/3)

This is a big improvement over the original test, which asked almost nothing useful about the behaviour of `NewThing`.

Let's refer to such tests, which don't test enough about the system, as *feeble*. A feeble test doesn't test as much as it thinks it does.

By contrast, this test now fails against our ridiculous null implementation of `NewThing`:

```
--- FAIL: TestNewThing (0.00s)
    thing_test.go:15: want X: 1, got X: 0
    thing_test.go:18: want Y: 2, got Y: 0
    thing_test.go:21: want Z: 3, got Z: 0
```

And rightly so, since `NewThing` currently does nothing useful. We can now detect that problem.

Do you have any feeble tests in your projects? Take a few minutes to find one, and disenfeeble it. Then read on.

Comparisons: `cmp.Equal` and `cmp.Diff`

In the previous example we compare each field of some struct `got` with some expected value, failing if it doesn't match. For complicated structs, though, this explicit field-by-field checking could get quite laborious. Can we do better?

We can make the test shorter, clearer, and more comprehensive by comparing the *entire* struct against some expected value. That way, we can specify our `want` value as a single struct literal, and check it with just one comparison.

The `go-cmp` package, which we encountered in the first chapter, is very helpful for this. Its powerful `cmp.Equal` function will compare any two values for deep equality:

```
import "github.com/google/go-cmp/cmp"

func TestNewThing(t *testing.T) {
    t.Parallel()
    x, y, z := 1, 2, 3
    want := &thing.Thing{
        X: x,
        Y: y,
        Z: z,
    }
    got, err := thing.NewThing(x, y, z)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing thing/4)

And if the structs *are* different, `cmp.Diff` will report only those fields that differ, which can be very helpful with big structs containing a lot of information.

Here's what that would look like with our non-functioning version of `NewThing`:

```
--- FAIL: TestNewThing (0.00s)
    thing_test.go:22:   &thing.Thing{
        -       X: 1,
        +       X: 0,
        -       Y: 2,
        +       Y: 0,
```

```
        -       Z: 3,
        +       Z: 0,
          }
```

Let's see if we can fix `NewThing` to at least pass this test:

```
func NewThing(x, y, z int) (*Thing, error) {
    return &Thing{
        X: x,
        Y: y,
        Z: z,
    }, nil
}
```

(Listing `thing/5`)

Maybe this still isn't enough for our `Thing` to actually be useful, but we've forced the implementer of `NewThing` to at least do *some* work.

Now that we've talked a little about *how* to write tests in Go, let's turn our attention to *why* we write those tests. The answer might seem obvious: to make sure the program is correct. And we've seen that tests can also help guide the design of the program in fruitful directions.

But tests can serve another purpose, too: they can be a powerful tool for *communicating* our ideas to others, or even our future selves. Let's talk about this in the next chapter.

3. Communicating with tests

> *Tests are stories we tell the next generation of programmers on a project. They allow a developer to see exactly what an application is made of and where it started.*
> —Roy Osherove, "The Art of Unit Testing"

What are tests, actually? One interesting and slightly unusual way to think about them is as a way of *communicating*. Not with the computer, except in a shallow sense.

Rather, we're really communicating with our fellow programmers, our successors, and even our future selves. So what *is* it that we're communicating, in fact? What's the point of a test?

Tests capture intent

Tests aren't just about verifying that the system works, because we could do that (slowly) by hand. The deeper point about tests is that they capture *intent*. They document what was in our minds when we built the software; what user problems it's supposed to solve; how the system is supposed to behave in different circumstances

and with different inputs.

As we're writing the tests, they serve to help us clarify and organise our thoughts about what we actually want the system to do. Because if we don't know that, how on earth can we be expected to code it?

> *No amount of elegant programming or technology will solve a problem if it is improperly specified or understood to begin with.*
> —Milt Bryce, "Bryce's Laws"

So the first person you need to communicate with when you're writing a test is yourself. Start by describing the required behaviour in words: "When the user does X, then Y happens, and so on." As you do this, the very act of description will prompt you to fill in some of the blanks: what if this happens? What if you get that input?

This won't be straightforward. If it were, this step wouldn't be necessary, but it is. Our brains are wonderful at ignoring irrelevant details, and picking out what's really important. They evolved that way. If there's a sabre-toothed cat leaping at you, you don't need to know what colour its stripes are. You just need to run like hell.

But computers don't work like that. If they did, we could just type one sentence and be done: *Do what I mean!* Instead, we have to explain every step, every twist and turn, and every change in circumstance.

In other words, we have to be clear.

> *The most important single aspect of software development is to be clear about what you are trying to build.*
> —Bjarne Stroustrup, "The C++ Programming Language"

The first question we need to ask ourselves before writing a test, then, is:

What are we really testing here?

Until we know the answer to that, we won't know what test to write. And until we can express the answer in *words*, ideally as a short, clear sentence, we can't be sure that the test will accurately capture our intent.

Test names should be sentences

So now that we have a really clear idea about the behaviour we want, the next step is to communicate that idea to someone *else*. The test as a whole should serve this purpose, but let's start with the test *name*.

You know that test functions in Go need to start with the word `Test`, but the rest of the function name is up to us. Usually, we don't think too hard about this part.

But maybe we're missing a trick. The name of the test isn't just paperwork, it's an opportunity for communication. And communication has a lot more to do with great software engineering than you might think:

> *It's possible that a not-so-smart person, who can communicate well, can do much better than a super smart person who can't communicate well. That is good news because it is much easier to improve your communication skills than your intelligence.*
> —Kevin Kelly, "103 Bits of Advice I Wish I Had Known"

Tests communicate by failing. When this test fails, as it surely will at some time or other, what will be printed out before anything else? *Its name*.

So, just like the specific failure string that we pass to `t.Error` or `t.Fatal`, the name of the test is also a message. What should it say? It should describe the behaviour of the system that the test checks. In other words, the name of the test is the behaviour that it's designed to *disprove*.

Let's take an example. Suppose we have some function called `Valid` that takes some input and checks whether or not it's valid. It doesn't matter what the input is, or what "valid" means in this context. Let's just say that the input is valid if it's the exact string `"valid input"`. Any other string will be considered invalid.

To write the test, we'll start by writing out the behaviour explicitly as a sentence, as concisely as we can without sacrificing relevant detail:

```
Valid returns true for valid input
```

It may not be possible to express in detail what "valid input" *means* in a single short sentence, but that's okay. The job of this initial sentence is to describe the *focus* of this test, not every detail about it.

Now it's time to expand this into runnable code. Here's the test that most Go programmers would probably write, if they're not used to thinking of tests as a communication tool:

```
func TestValid(t *testing.T) {
    t.Parallel()
    want := true
    got := valid.Valid("valid input")
    if want != got {
        t.Errorf("want %t, got %t", want, got)
    }
}
```

(Listing valid/1)

Failures are a message to the future

There doesn't seem to be *too* much wrong with this, and indeed there isn't; it's a perfectly reasonable test. But it doesn't *communicate* as much as it could. When it fails, what do we see?

```
--- FAIL: TestValid (0.00s)
    valid_test.go:12: want true, got false
```

How useful is this output? Remember, we won't be looking at the code for the test function when we see this. All we'll see is this message, and it might be in the middle of a long list of other output. It conveys no context about what's going on, what the test was trying to do, what the input was, or what this failure means.

All we know is what we see: "TestValid: want true, got false". That's pretty opaque. It leaves the reader with many questions:

- What value were we inspecting?
- Why did we expect it to be `true`?
- For what input?
- What does it mean that we got `false` instead?
- In what respect is the system not behaving as specified?

In order to solve the problem indicated by this failing test, we need to do a fair amount of work. We have to find the relevant test, read it through and understand what it's doing overall, look at the specific line that's failing, understand why it might have failed, and only *then* go and fix the system code.

Multiply this by more than a few test failures, and it's clear that we're inadvertently creating a lot of unfair extra work for some poor, harassed developer in the future. This is a serious problem, not least because that developer might well be *us*.

Can we do more to help? Is there any work we could do in advance of the test failure, to make it easier to diagnose when it does happen?

Let's start with the *name* of the test. Something vague like `TestValid` is no help (what *about* `Valid`?) Instead, we could use the test name to describe the required behaviour in detail.

We already have a short sentence that does this:

```
Valid returns true for valid input
```

Why shouldn't we simply remove the spaces from this sentence and use the result as the name of the test?

```
func TestValidIsTrueForValidInput(t *testing.T) {
```

Remember, when the test fails, this name will be the first thing printed out:

```
--- FAIL: TestValidIsTrueForValidInput (0.00s)
```

In fact, this name conveys so much information, we may not need to say very much else. If `Valid` is supposed to be `true` for valid input, and we're seeing this failure, then we can immediately infer that it must have, in fact, been `false` for valid input.

It's still good practice to have `t.Error` report the actual result, even if there's only one possibility, as in this case. So here's the updated test:

```
func TestValidIsTrueForValidInput(t *testing.T) {
    t.Parallel()
    if !valid.Valid("valid input") {
        t.Error(false)
    }
}
```

(Listing valid/2)

So the complete failure message, including the name of the test, is:

```
--- FAIL: TestValidIsTrueForValidInput (0.00s)
    main_test.go:20: false
```

And even though the failure message is much shorter than our original version, it actually gives much more information *when combined* with a test name in the form of a sentence.

Are we testing all important behaviours?

Taking advantage of an otherwise neglected communication channel—the test name—helps make future test failures easier to fix. But there's another benefit, too.

In order to write a useful sentence describing the system's behaviour, we needed to include three things: the *action*, the *condition*, and the *expectation*. In other words:

Test names should be ACE: they should include Action, Condition, and Expectation.

For example, in the test for our `Valid` function, these are as follows:

- Action: calling `Valid`
- Condition: with valid input
- Expectation: returns `true`

By being explicit about these, the test sentence can also suggest other possibilities. For example, how should `Valid` behave under different *conditions*? What should it do when given *invalid* input?

And that's something we need to know. While it's important that `Valid` is true for valid input, it's equally important (if not more important) that it's false for invalid input. After all, it might just *always* be true, which would pass this test, but still not be useful in the real program.

In other words, we need to think about implementations of `Valid` that would pass the "valid input" test, but nevertheless be incorrect. Here's a simple example:

```
func Valid(input string) bool {
    return true
}
```

(Listing valid/3)

The *point* of `Valid` is to reject invalid input, but we don't have to look too hard at the code to see that, in fact, it can never do this. Yet our test will pass, and if we check coverage, we'll see that the function is 100% covered by the test.

Many real tests are just like this. The developers feel good about the system, because the tests pass, and the code is well covered. But they're not looking closely enough at what the tests are really *testing*. Let's not make the same mistake.

After we've written a test, then, it's important to ask ourselves the question:

What are we really testing here?

It's possible to set out with a very clear intention, yet end up writing a test that doesn't actually do quite what we meant it to. We need to review the code afterwards and see whether it really expresses the intent captured by the original behaviour sentence. If it does, are there *other* important behaviours remaining that we haven't yet tested?

> *By saying everything two ways—both as code and as tests—we hope to reduce our defects enough to move forward with confidence. From time to time our reasoning will fail us and a defect will slip through. When that happens, we learn our lesson about the test we should have written and move on.*
> —Kent Beck, "Test-Driven Development by Example"

The power of combining tests

What we've realised by going through this thought process is that `Valid` actually has *two* important behaviours, and we've only tested one of them: that it returns `true` for valid input.

That's the "happy path", if you like, and it's important, but it's also easy to pass with a completely broken implementation of `Valid`, as we've seen.

To have confidence that `Valid` can actually *discriminate* between valid and invalid inputs, we need to test the "sad path" too. For example:

```
func TestValidIsFalseForInvalidInput(t *testing.T) {
    t.Parallel()
    if valid.Valid("invalid input") {
        t.Error(true)
    }
}
```

(Listing valid/4)

Of course, this test is also insufficient *by itself*. If this were the only test we had, it would be easy to pass by simply having `Valid` always return `false`, whatever its input.

The point is that the two tests *combined* can give us confidence in the correctness of `Valid`. Each test tells us something useful about the system's behaviour on its own, but not the whole story. Only by running them together can we detect faulty implementations such as always returning a fixed value.

Another way to think about this is there are two possible behaviours of `Valid`, so we need to cover both of them with tests. Remember, test behaviours, not functions. Each behaviour of the system is a distinct code path that needs to be exercised by some test if we're to have confidence in it.

The "lazy" `Valid` implementation might seem unrealistic, but it's the sort of thing that can easily happen in large projects, with many developers and testers, and not enough time. Maybe the implementer had multiple failing tests, and needed to "fix" them quickly to bypass the normal deployment checks. They might even have left a comment to remind themselves to implement the function properly later:

```
return true // TODO: write real implementation
```

Unfortunately, tests can't read comments. And, if we're being honest with ourselves, `TODO` comments like this almost never result in action anyway. They represent the graveyard of things we'll never get around to.

Bugs aren't always as obvious as this, though. If only they were, our work would be a lot easier. Let's look at another possible incorrect implementation of `Valid`:

```
func Valid(input string) bool {
    return strings.Contains(input, "valid input")
}
```

(Listing `valid/4`)

This almost *looks* reasonable, doesn't it? And it will pass the "valid input" test. But it's still wrong. Can you see how? If not, think about what would happen if we called `Valid` with the string `"invalid input"`.

Recall that we defined the input as valid if it's *exactly* the string `valid input`. In other words, it's not good enough for `input` just to contain that string: it also mustn't contain anything *else*. So `strings.Contains` isn't the right thing to use here.

This is why we also need a test that `Valid` returns false when given input that contains `valid input` as a substring. We can imagine an incorrect implementation that's equivalent to `strings.Contains`, so we need a bug detector that detects this.

Even when we're being very disciplined and writing tests first, it's easy to end up writing this sort of bug. And because we *have* passing tests, this can give us a completely unjustified confidence in the system.

Over time, this can create a dangerous complacency. The tests have been passing for months! So the system *must* be correct. Mustn't it?

Just as a car needs regular inspection and service, it's a good idea for us to take a really close look at our tests every so often and ask:

What are we really testing here?

And, as we've seen, one effective way to answer this is to imagine some *incorrect* implementations that could nevertheless pass the test. Then we extend the tests so that we can automatically detect and eliminate such bugs.

The thing that most people forget about code is that it *changes* over time. Even if the implementation is correct *now*, someone might look at it in a few months or years and ask, "Could I refactor this function to just return `true`?" If they try this, and it passes the tests, they will quite justifiably assume that their refactoring is correct.

That's why we need to write more robust tests. Remember the "bug detector" that Martin Fowler talks about? It needs to detect not only the bugs that we write today, but also the bugs that we—or someone else—might write in the future.

Reading tests as docs, with `gotestdox`

The combined power of our two tests is too much for even the laziest and most malicious implementer. Now the quickest and easiest way to get the tests passing is simply to write a *correct* implementation of `Valid`. Here's one:

```
func Valid(input string) bool {
    return input == "valid input"
}
```

(Listing `valid/5`)

And since most people will understandably get their jobs done in the quickest, easiest way possible, it's a good idea to make the easiest way also the correct one. These tests do exactly that.

We now have two very short, very focused, and very important tests for `Valid`. Let's see what their names tell us, as behaviour sentences:

```
func TestValidIsTrueForValidInput(...)

func TestValidIsFalseForInvalidInput(...)
```

We now have a complete description of the important behaviour of `Valid`. It's almost like documentation. Indeed, we could take just these test names, strip out the `func` boilerplate, add some spaces back in, and print them out like this:

```
Valid is true for valid input
Valid is false for invalid input
```

These are the sentences we started with: the ones that guided our writing of the original tests. They perfectly capture our intent. They say what we're really testing here.

And wouldn't it be handy to automate this process, so we could just run our test code through some translator that prints out these sentences?

Funny you should ask that. There's a tool called gotestdox you might like to try. Let's install it:

```
go install github.com/bitfield/gotestdox/cmd/gotestdox@latest
```

If we run it in our package directory, it will run the tests, and report the results, while simultaneously translating the test names into readable sentences:

```
gotestdox
```

```
valid:
 ✔ Valid is true for valid input (0.00s)
 ✔ Valid is false for invalid input (0.00s)
```

Why the name `gotestdox`? Because these aren't just test results: they're *docs*. This isn't a new idea, it turns out:

> *My first "Aha!" moment occurred as I was being shown a deceptively simple utility called `agiledox`, written by my colleague, Chris Stevenson. It takes a JUnit test class and prints out the method names as plain sentences.*
>
> *The word "test" is stripped from both the class name and the method names, and the camel-case method name is converted into regular text. That's all it does, but its effect is amazing.*
>
> *Developers discovered it could do at least some of their documentation for them, so they started to write test methods that were real sentences.*
> —Dan North, "Introducing BDD"

When you see your test names printed out as sentences, it's suddenly clear how much more they could communicate. You'll start thinking about your tests in a new way, by writing names that are real sentences.

One of the first steps I take when code-reviewing a student's project is to run its test names through `gotestdox`, and show them the result. It's a very effective way of getting them to think about their test names as a useful communication channel, and usually has a significant effect on improving the tests.

Most developers, as we've seen, instinctively name their tests after the *action* involved: calling `Valid`, for example. That's a good start. But by naming the test with a complete *sentence*, we're forced to also include the two other relevant facts about the test. Those are the *conditions* under which `Valid` is being tested, and our corresponding *expectations* about what it should do in that case.

Without these, a test name doesn't really tell us much. A name like `TestValid` isn't a sentence. If you try to read it that way, as a statement about the behaviour of `Valid`, it's incomplete:

> `Valid`...

Now you know how to complete that sentence, don't you? By saying what `Valid` *does*, and *when*. For example, "`Valid` returns true for valid input."

Running the test suite through a tool like `gotestdox` can point out a lot of opportunities to add useful information to our test names, and maybe even some new tests.

> *I see a lot of Go unit tests without the condition, just a list of expectations (or contrariwise, just a list of states without expectations). Either way, it can easily lead to blind spots, things you are not testing that you should be.*
> —Michael Sorens, "Go Unit Tests: Tips from the Trenches"

Definitions: "does", not "should"

If you're now convinced that test names should be sentences, or you're at least intrigued by the possibilities of this idea, let's think a little more about what those sentences might say, and how they should say it.

First, as we've seen, the sentence should describe the behaviour of taking some *action* under some particular *conditions*, and it should say specifically what our *expectation* is in that case. For example:

> Valid is true for valid input.

Notice that we didn't say, for example:

> Valid *should be* true for valid input.

It's tempting to include "should" in every test sentence, especially if we're writing the test first. I don't think that's necessary, and we can keep our test sentences short and to the point by omitting words like "should", "must", and "will". Just say what it *does*.

A good way to think of this is to imagine that every test sentence implicitly ends with the words "...when the code is correct". Let's try that out:

> Valid is true for valid input *when the code is correct*.

We wouldn't say that `Valid` *should* be true when the code is correct, we would say it *is* true! This isn't an aspiration, in other words: it's a *definition*.

> *Let me ask you, what's the purpose of your test? To test that "it works"? That's only half the story. The biggest challenge in code is not to determine whether "it works", but to determine what "it works" means.*
> —Kevlin Henney, "Program with GUTs"

`gotestdox` will not only format this definition for us in a helpful way, it will also report on its *status*. If the test is passing, the sentence will be preceded by a checkmark:

```
✔ Valid is true for input (0.00s)
```

If the test fails, an `x` before the test sentence shows that the relevant part of the system is not yet correct:

```
 x Valid is false for invalid input (0.00s)
```

A sentence is about the right size for a unit

If you find that it's difficult to express the behaviour you're testing as a single concise sentence, then that can be a useful signal in itself. You may be trying to cram too much behaviour into a single test.

> *If it takes more than a sentence to explain what you're doing, that's almost always a sign that it's too complicated.*
> —Sam Altman, "How to Start a Startup"

To make things easier for both yourself and anyone trying to read the tests, split the behaviour up into multiple tests instead.

> *What to call your test is easy: it's a sentence describing the next behaviour in which you are interested. How much to test becomes moot: you can only describe so much behaviour in a single sentence.*
> —Dan North, "Introducing BDD"

Indeed, maybe you're trying to cram too much behaviour into a single *unit*. That's a smell, too. Turning that around, we might say that a well-designed unit should have no more behaviour than can be expressed in a few short sentences, each of which can be translated directly into a test.

It turns out that the information contained in a single sentence corresponds quite well to the amount of behaviour that a unit should have. In both cases, it's about how much complexity our minds are comfortable dealing with in a single chunk.

And if you can't actually express your test in terms of some user-visible behaviour, or as fulfilling some external requirement for the system, then maybe you're not really testing anything useful. Perhaps you should just skip this test altogether.

> *Your unit test name should express a specific requirement. This requirement should be somehow derived from either a business requirement or a technical requirement. If your test is not representing a requirement, why are you writing it? Why is that code even there?*
> —Roy Osherove, "Naming Standards for Unit Tests"

Let's look at some examples from the Go project itself, as reported by `gotestdox`:

```
std/encoding/csv:
 ✔ Read simple (0.00s)
 ✔ Read bare quotes (0.00s)
 ✔ Read bare double quotes (0.00s)
 ✔ Read bad double quotes (0.00s)
 ✔ Read bad field count (0.00s)
 ✔ Read trailing comma EOF (0.00s)
 ✔ Read non ASCII comma and comment (0.00s)
```

```
 ✔ Read quoted field multiple LF (0.00s)
 ✔ Read multiple CRLF (0.00s)
 ✔ Read huge lines (0.00s)
...
```

The moment any of these tests fails, it'll be really clear what specific behaviour is not happening as expected:

```
 x Read huge lines (0.00s)
```

Keeping behaviour simple and focused

Limiting the behaviour under test to what can be described in a single sentence not only helps us focus on what's important, it can help prevent us *over-engineering* the system.

When we're just writing a function directly, without being guided by tests, it can be difficult to know where to stop. Should we think about this or that edge case? What could go wrong here? Should we define some custom error type to return if we hit this problem? You could keep writing code all day and not be sure when you're done.

By contrast, a test tells us exactly when we're done, because it starts passing! It's a way of limiting scope in advance, and forcing us to decide in advance what we do and don't consider important—at least, at this stage.

Good design is not only about what we have the imagination to put in, but also what we have the clarity of purpose to leave out:

> *The difference between a good and a poor architect is that the poor architect succumbs to every temptation and the good one resists it.*
> —Ludwig Wittgenstein, "Culture and Value"

Again, tests act as a useful quality filter for our ideas. Of course we *have* lots of ideas as we're programming ("What about this? What about that?"), and that's great. But the disciplined programmer will simply note these ideas down on the list of possible user stories for later, and continue to focus on the current task.

It's quite likely that we'll come along and extend the required behaviour later on, when we know more about the system. But at every stage, the test specifies the behaviour we care about right now, and defines what "done" means for that particular behaviour. And that starts with expressing the name of the test as a short sentence.

Shallow abstractions: "Is this even worth it?"

Sometimes when we write the required behaviour as a sentence we find, not that the unit under test is doing too much, but that it's doing too *little*. This can be a clue that the behaviour you're testing isn't actually necessary, or operates at too low a level to be worth the cost of building and using it.

Abstractions, to be useful, should be *deep*: that is, they should conceal powerful machinery behind a simple API. A shallow abstraction, on the other hand, just creates paperwork and doesn't provide any value for users. We've all come across plenty of those.

> *The best modules are deep: they have a lot of functionality hidden behind a simple interface. On the other hand, a shallow module is one whose interface is relatively complex in comparison to the functionality that it provides.*
>
> *Shallow modules don't help much in the battle against complexity, because any benefit they provide is negated by the cost of learning and using their interfaces.*
> —John Ousterhout, "A Philosophy of Software Design"

If your abstraction is too shallow, that'll often become clear as you try to write the test sentence for it. If the sentence implies that the abstraction really doesn't do much, take this as a cue to redesign it.

Try to increase the ratio between the complexity of its API and the amount of *work* it actually does for the user. The bigger this ratio, the deeper the abstraction, and thus the more leverage it gives you.

Don't worry about long test names

You might be concerned that naming tests as sentences will result in some awkwardly long names for your test functions:

```
TestValidIsTrueForValidInput
```

True, we probably wouldn't want to use function names as long as this in our *system* code. Good code is concise, and in normal programming we tend to choose names that are as short as possible, while still remaining clear. But *test* names are a little different.

We never *call* these functions, so we don't *refer* to those names in code. And users of our package won't see them, because they're not part of the public API. So if their names don't matter to anyone but us, why shouldn't we make them something meaningful?

A tool like `gotestdox`, which gives us a clear picture of exactly what information our tests are communicating, can be a very useful thinking aid. It can help us break up the system itself into more focused and self-contained units of behaviour. It can help us design better, more comprehensive, and more readable tests for those units.

And it can help bring about a fundamental change in the way we think about tests. We can switch from treating tests as dull but necessary paperwork, to using them as a clear, expressive, and automatically verifiable description of the system itself, in language that users understand.

The next time you write a test, then, you might like to try asking, not just "what are we really testing here?", but also "what *intent* is this test *communicating*?"

Crafting informative failure messages

While the name of the test is an important tool for communicating intent, its *failure messages* are arguably even more important.

When we're writing the *system* code we tend to focus on the happy path: what the function does when everything goes according to plan. Writing *test* code, though, is all about imagining errors that might happen, and bugs that might be present, and then designing experiments to detect them.

This is one reason that more experienced programmers write better tests: we've simply *written* more bugs over the years. Not only do we have a good sense of what kind of bugs are possible or likely in a given code path (off-by-one errors, for example), we've also learned something about the kind of mistakes that we personally tend to make.

We all have our own individual style of programming, and similarly, we all have particular blind spots and bad habits. Being aware of them and being able to catch them with tests is a crucial part of our development as software engineers.

If we accept that tests are for failing, then the implication is clear: the most important parts of the test must be the *failure messages*.

Again, this tends to be the area that inexperienced programmers pay the *least* attention to. The worst mistake you can make in a test, as we've seen, is not to catch the problem at all, but the next worst is to report nothing useful about it:

```
if want != got {
    t.Error("fail")
}
```

That's redundant. If we're seeing a message *at all* when running the tests, it means something failed. But what? We have no idea.

This is only marginally better:

```
if want != got {
    t.Error("want not equal to got")
}
```

Again, we could have inferred this from the fact that the test is failing in the first place. What would be useful here is some information about what *should* have happened:

```
if want != got {
    t.Errorf("want %v, but didn't get it", want)
}
```

Even this is a frustrating failure message to see, though. It tells us what was *supposed* to happen, and that it didn't, but we have no clue as to why not, or what happened instead. What's the first thing we would do to start diagnosing a failure like this?

We'd print out the *actual* value of `got`, of course. So the helpful test writer will save us this step, by anticipating it:

```
if want != got {
    t.Errorf("want %v, got %v", want, got)
}
```

If the values of `want` and `got` are long, complicated, or otherwise hard to read (slices of bytes, for example), we can use `cmp.Diff` again:

```
if want != got {
    t.Error(cmp.Diff(want, got))
}
```

In the case of a failure, the test output might look something like this:

```
  Thing{
    X: 10,
-   Y:  99,
+   Y:  100,
    Z: 5,
  }
```

Here, what was *expected* was `Y: 99`, but what we *got* was `Y: 100`. That information by itself may well be enough for the programmer to guess what the underlying problem is.

It's good to know that `want` doesn't equal `got`, and what the difference is, but even that's not particularly useful in isolation. A failure like this doesn't actually tell us what that mismatch *means*, in terms of the behaviour under test.

A good test failure should say something about what *user-visible problem* the test has detected:

```
if want != got {
    t.Error("wrong account balance after concurrent deposits",
        cmp.Diff(want, got))
}
```

If the test's job is to demonstrate that a particular *input value* causes a problem, that should also be included in the failure message:

```
if err != nil {
    t.Error("input %d caused unexpected error: %v", input, err)
}
```

Ideally, the developer reading the test failure message should have all the information they need to find and fix the problem, based purely on the *message*. They shouldn't have to read the *test* code at all.

Exercising failure messages

Failure messages, then, if they're well-crafted, are the most critical and valuable parts of any test. Isn't it strange, in that case, that they're never *seen* by the majority of programmers writing them?

You already know that it's important to see every test fail when you know the system is incorrect. It's our way of evaluating the test *logic*: does it detect the bug it's supposed to detect? And that's an important thing to check, but we need to do more.

Once the system is implemented and working (we think), we can start to evaluate the various failure messages of our tests. For each failure path, of which there may be several in any given test, we want to see the test *output*. We need to check that it makes sense, prints what it's supposed to print, and includes all the relevant information. How can we do that?

The test is designed to catch a problem in the system code, so if we want to know whether it will successfully do that, we need to introduce a problem *in the system code*.

Don't be tempted to tweak the test code, for example by changing the value of `want`. In fact, the *correct* value of `want` is one of the things we're supposed to see when the test fails. So changing it just to induce the failure would be counterproductive.

Suppose we have a test for some function `Double` that looks like this:

```
func TestDouble2Returns4(t *testing.T) {
    t.Parallel()
    want := 4
    got := double.Double(2)
    if want != got {
        t.Errorf("Double(2): want %d, got %d", want, got)
    }
}
```

(Listing double/1)

How should we induce this failure artificially, in order to check the output of the `t.Errorf` call? We could change `want` to 5, but that's silly. In order to know that the test can detect a bug in the function, we need to add a bug to the *function*, not the test.

For example, we could add a temporary change to the `Double` function to skew its result slightly:

```
func Double(n int) int {
    return n*2 + 1
}
```

(Listing double/1)

If *that* doesn't cause the test to fail, what would? If it does, though, we can have reasonable confidence that the test tests something useful about `Double`. Don't forget to remove your deliberately-inserted bug once you've checked this, though.

Executable examples

As we've seen, there's a lot we can communicate to readers using Go tests, starting with the test names, and continuing with the logic of the tests themselves. By using the system in a test, we can not just verify things about its behaviour, but also show *users* how they can call the system, and how they should expect it to behave.

That's great, but sometimes these two aims conflict a little. For example, many tests require a fair amount of code to set up an environment that resembles the one in which it would really be used. But *users* don't need to do that, because they're writing real programs, not tests. So the communication aspect gets slightly obscured by the verification aspect.

Similarly, while we might write wonderful documentation comments on each of our exported functions, our *tests* won't appear as part of the resulting autogenerated documentation. So someone reading about your package on `pkg.go.dev`, for example, will see your documentation, but not your code or tests.

What we'd like is a way to write code that will be included in the autogenerated documentation, *and* whose behaviour can be automatically verified in the same way as the tests. That's exactly what Go provides, with a feature called *executable examples*.

> *A good example can be a more succinct or intuitive way to convey the behavior of a library function than its prose description, especially when used as a reminder or quick reference.*
> *And unlike examples within comments, example functions are real Go code, subject to compile-time checking, so they don't become stale as the code evolves.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

An executable example is like a test function, but even simpler. Its name must begin with the word `Example`, but it doesn't take any parameters, or return anything:

```
func ExampleDouble() {
    fmt.Println(double.Double(2))
    // Output:
    // 4
}
```

(Listing `double/2`)

Very straightforward. In this example, we call `Double(2)` and print the result. A comment shows that the expected output is `4`.

So what? Well, the neat thing is that this example function is automatically included in the documentation for your package. Here's what that might look like on `pkg.go.dev`:

We can see that the program has been reformatted to put the example code inside a `main` function, and that the expected output is shown separately.

Indeed, you can run the example directly in your browser by clicking the *Run* button on the website. Try it with the examples from the `script` package:

- https://pkg.go.dev/github.com/bitfield/script

When you click *Run*, the example program is built, its dependencies downloaded, the resulting binary run, and the output displayed, all right in the browser. You can also edit the example to do whatever you like, and try out your modified program in the same way.

In fact, the examples are being run using the Go Playground, and you can do the same kind of thing by browsing to:

- https://go.dev/play/

and entering whatever program you like.

There are some common-sense restrictions on code that runs in the playground (and thus on executable examples running on `pkg.go.dev`). For example, there's no networking available, and the amounts of CPU, memory, and execution time are limited to

prevent abuse.

And, interestingly, you'll find that the time and date returned by `time.Now` in playground programs is always the same: 11pm on Tuesday, November 10, 2009. This is a significant date for Gophers, naturally, but it also means that the behaviour of time-related code is deterministic.

When you run your examples locally, on your own machine, using `go test`, these restrictions don't apply, but just be aware that they *will* apply to people running them on `go.pkg.dev`.

Your executable examples are not only included in the documentation for your package, but they're also run whenever you run `go test`. What exactly is being *tested* here, then, you might wonder? And how could such a test fail?

Let's try changing the comment showing the expected output, and see what happens. For example, suppose we make it "5", instead of "4":

```
func ExampleDouble() {
    fmt.Println(double.Double(2))
    // Output:
    // 5
}
```

Running `go test` produces this result:

```
--- FAIL: ExampleDouble (0.00s)
got:
4
want:
5
FAIL
```

So that `Output` comment is not *just* a comment, it turns out. It's also an *expectation* that the test machinery will verify.

We check the behaviour of the function, in other words, by asserting something about its standard output, ignoring any leading and trailing whitespace.

We can use examples to test functions that *print* to standard output by just calling them directly. Indeed, it's easier to test such functions using executable examples than it would be with a regular test:

```
func ExamplePrintHello() {
    PrintHello()
    // Output:
    // Hello
}
```

If the function doesn't happen to print anything, but instead returns a result, we can use `fmt.Println` to print that result, as in the `Double` example, and then verify it with the `Output` comment.

We can also match multiline output, if necessary:

```
func ExamplePrintHello() {
    PrintHello()
    PrintHello()
    // Output:
    // Hello
    // Hello
}
```

If we know what *lines* should be present in the output, but not in what *order* they will appear, we can use a different version of the `Output` comment:

```
func ExamplePrintMap() {
    m := map[int]bool{
        1: true,
        3: true,
        2: false,
    }
    for k, v := range m {
        fmt.Println(k, v)
    }
    // Unordered output:
    // 1 true
    // 2 false
    // 3 true
}
```

By specifying the example's *unordered* output, we're saying that each of these lines must be present in the output, but they can appear in a different order to that given. This will likely be true when ranging over a map, for example.

You might also be wondering how Go knows that `ExampleDouble`, for instance, should be attached to the documentation for the `Double` function. That's easy: whatever follows the word `Example` in the example name is taken to be the name of the function it documents.

So if your function is called `Greet`, and you want to provide an example for it, you would name that example function `ExampleGreet`. The documentation tool will then automatically associate the example with the `Greet` function.

What if you want to give more than one example for the same function? You can do that by adding a suffix after the function name:

```
func ExampleDouble_with2() {
    fmt.Println(double.Double(2))
    // Output:
    // 4
}

func ExampleDouble_with3() {
    fmt.Println(double.Double(3))
    // Output:
    // 6
}
```

(Listing double/2)

In fact, you can supply as many examples as you want, provided each has a unique name, whose suffix begins with a lowercase letter (for example, `with2`).

If you want to write an example for the whole *package*, rather than any specific function, just name it `Example`:

```
func Example() {
    // this demonstrates how to use the entire package
    ...
}
```

Otherwise, Go will assume that whatever follows the word `Example` is the name of the thing you're documenting.

This works with types, too. Suppose you define some type `User`:

```
type User struct {
    Name string
}
```

(Listing user/1)

As part of your documentation, you can supply an example in your test file:

```
func ExampleUser() {
    u := user.User{ Name: "Gopher" }
    fmt.Println(u)
    // Output:
    // {Gopher}
```

```
}
```

(Listing user/1)

What about *methods* on types? We can give examples for those too. Suppose you add a method on `User` named `NameString`, to return the user's name as a string:

```
func (u User) NameString() string {
    return u.Name
}
```

(Listing user/1)

You can add an example to your test file that shows what this method does:

```
func ExampleUser_NameString() {
    u := user.User{Name: "Gopher"}
    fmt.Println(u.NameString())
    // Output:
    // Gopher
}
```

(Listing user/1)

The convention is simple: the word `Example` is followed by the name of the type we're documenting (in this case `User`), then an underscore, then the name of the specific method (here it's `NameString`).

It's always a great idea to use code examples as part of our documentation anyway, but if they're just text, there's a danger that they'll become out of date and stop working. Indeed, they might never have worked in the first place.

We've probably all had the unpleasant experience of copying and pasting an example code snippet from someone's documentation, and finding not only that it doesn't behave the way we expected, but that it doesn't even compile. Not a good look for the project.

So it's nice that Go provides a way for us to supply examples that are automatically included in our documentation, that users can edit and run in a web browser without even needing Go installed, and that are also automatically checked every time we run `go test`.

Go itself uses executable examples quite widely in the standard library documentation (see the `strings` package documentation, for instance). But most third-party packages don't bother to include examples, which I think is a shame. It's an easy way to add a lot of value to your package, and also lends a rather professional appearance to the documentation.

In this chapter we've explored some interesting ways to think about tests as communication tools. In the next, we'll examine the tests themselves in greater detail, with a

focus on testing *error* behaviour.

4. Errors expected

> *While it is a known fact that programmers never make mistakes, it is still a good idea to humor the users by checking for errors at critical points in your programs.*
> —Robert D. Schneider, "Optimizing INFORMIX Applications"

We've talked a lot about tests that are designed to fail, and how to catch unexpected errors in the system code. But what about *expected* errors?

In other words, if part of the behaviour of your public API is to return error values in certain situations, how do you test that? In this chapter, we'll explore some of the possible ways.

Ignoring errors is a mistake

It's common for Go functions to return an error value as part of their results. It's especially common for functions to return "something and error", like this:

```
func CreateUser(u User) (UserID, error) {
```

How do we test a function like this? Let's look at a few of the wrong ways to do it first.

One *really* wrong way would be something like this:

```
func TestCreateUser(t *testing.T) {
```

```
    want := 1
    got, _ := CreateUser("some valid user")
    if want != got {
        t.Errorf("want user ID %d, got %d", want, got)
    }
}
```

We know from its signature that `CreateUser` returns an error as part of its API, but we ignore this result value completely in the test, by assigning it to the blank identifier (`_`).

And you can see why this is a bad idea, can't you? It's another example of the happy-path fixation. If there were some bug that caused `CreateUser` to return an error when it shouldn't, would this test detect it? Nope.

But there's a more subtle problem, too. What if there were a bug in the *test*? We appear to be testing the "valid input" behaviour, but what if we mistakenly passed *invalid* input instead? What would we see when we ran the test?

```
want user ID 1, got 0
```

Wait, what? We can stare at the code for `CreateUser` as long as we like, and we won't see the problem that's causing this, because it isn't there.

The problem is not that `want` doesn't equal `got`, although that happens to be true. The problem is that we shouldn't have even *looked* at `got`, because there was an error. By returning a zero user ID and an error, `CreateUser` is trying to tell us that something's gone wrong, but we're not listening.

This is something that often trips up people new to Go. In some languages, functions can signal an unhandled error condition using an *exception*, for example, which would cause a test like this to fail automatically. This isn't the case in Go, where we signal that `got` is invalid by also returning a non-nil error.

That puts the burden of checking the error on the caller: in this case, that's the test. So, as we can see, it's easy to make a mistake where an unchecked error results in the code proceeding as though `got` were valid when it's not.

We don't want to write tests for our tests (where would it end?), but we *can* write our tests in a defensive way, by always checking errors. At least then if we *do* make a mistake, the resulting failure will give us a clue about what it was.

Ignoring error values in a test, or indeed anywhere in a Go program, is a pretty common mistake. It might save us a second or two now, but it'll cost us (or our successors) a lot of puzzlement later. Let's not store up any more trouble for the future than we have to.

A great way to add value and robustness to any existing Go test suite is to go through it looking for places where errors are ignored using `_` (static analysers such as `errcheck` can find such places for you). Remove the blank identifier and assign the error result to the `err` variable, then check it and fail the test with `t.Fatal` if necessary.

Unexpected errors should stop the test

How exactly *should* we fail the test if there's an unexpected error? One idea is to call `t.Error`, but is that good enough?

```
func TestCreateUser(t *testing.T) {
    ...
    got, err := CreateUser(testUser)
    if err != nil {
        t.Error(err)
    }
    ... // more testing here
}
```

No, because as we saw in an earlier chapter, `t.Error`, though it marks the test as failed, also continues. That's not the right thing to do here. We need to stop the test right away. Why?

If `err` is not `nil`, then we don't have a valid result, so we shouldn't go on to test anything about it. Indeed, even *looking* at the value of `got` could be dangerous.

Consider, for example, some function `store.Open`:

```
func Open(path string) (*Store, error) {
    ... // load data from 'path' if possible
    ... // but this could fail:
    if err != nil {
        return nil, err
    }
    return &Store{
        Data: data,
    }, nil
}
```

It's conventional among Gophers that functions like this should return a nil pointer in the error case. So any code that tries to *dereference* that pointer will panic when it's `nil`:

```
func TestOpen(t *testing.T) {
    s, err := store.Open("testdata/store.bin")
    if err != nil {
        t.Error(err)
    }
    for _, v := range s.Data { // no! panics if s is nil
        ...
```

```
    }
}
```

Suppose `store.bin` doesn't exist, so `Open` can't open it, and returns an error. The test detects this, and calls `t.Error` to report it, but then continues.

Now we're going to try to `range` over `s.Data`. But that won't work when `s` is nil, as it will be if there was an error. Dereferencing the `s` pointer will cause the test to panic, confusingly.

This isn't a total disaster, because the test will still fail, so we'll at least know that *something's* wrong. But, as we've established in previous chapters, it's important to make sure that a test fails for the *right reasons*. Otherwise, it's hard to know what it's actually telling us about the problem.

> *For the robust, an error is information; for the fragile, an error is an error.*
> —Nassim Nicholas Taleb, "The Bed of Procrustes: Philosophical and Practical Aphorisms"

The problem, it turns out, is not in `Open` at all. It's in our *test*, because we should never have touched the `s` pointer once we knew it was `nil`.

> *NIGEL: Don't touch it!*
> *MARTY: I wasn't going to touch it, I was just pointing at it.*
> *NIGEL: Well, don't **point**, even.*
> —"This is Spinal Tap"

Instead, the test should call `t.Fatal` to bail out immediately. Or, in this case, since we'd like to include some formatted data in the message, `t.Fatalf`:

```
func TestOpen(t *testing.T) {
    s, err := store.open("testdata/store.bin")
    if err != nil {
        t.Fatalf("unexpected error opening test store: %v", err)
    }
    ... // now we can safely use 's'
}
```

`t.Fatal` isn't just for unexpected errors from the system under test. We should also use it to report any problems we encounter when setting up the *context* for the test, such as our `want` value. After all, if we can't set up the necessary preconditions, we need to stop the test before it proceeds any further.

Suppose, for example, that we want to compare a function's result against the contents of some pre-prepared *golden file*. And suppose that file doesn't happen to exist for some reason, or we just got the name wrong.

For example, suppose we spell it `godlen.txt` instead of `golden.txt`. In that event, we'd want to see a test failure that tells us what's wrong, and we *wouldn't* want to continue with the test. So, again, we can use `t.Fatalf`:

```
// Can you spot the deliberate mistake?
want, err := os.ReadFile("testdata/godlen.txt")
if err != nil {
    t.Fatalf("unexpected error reading golden file: %v", err)
}
```

This is another kind of potentially confusing test bug. We can detect it by being disciplined about our error checking, and using `t.Fatal` to short-circuit the test.

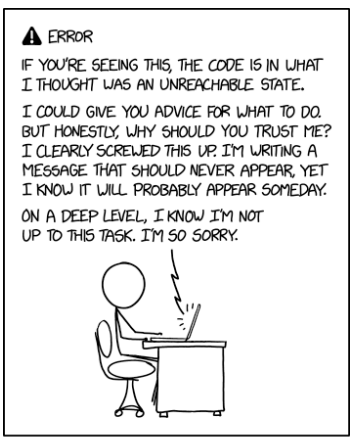
Error behaviour is part of your API

So that's how we deal with unexpected errors, but what about *expected* errors? After all, if errors are part of our public API, which they usually are, then we need to test them. Arguably, the way the system behaves when things go wrong is even *more* important than what it does when things go right.

Here's why. If the user calls our function and it just works, that's great: they'll never think any more about it. But if it *doesn't* work for some reason, now the user has a problem and they need our help.

The user needs to know, not only that this operation didn't work in general, but more importantly, what *specifically* didn't work about it, and why not. We can go further, and also suggest what they might do to solve the problem, for example. This kind of care and forethought in writing helpful, informative error messages adds a lot of value to code.

Judging by some of the error messages I've encountered, it's clear that the programmer didn't really expect anyone to ever read them. Sometimes they betray a certain amount of inner angst:



NEVER WRITE ERROR MESSAGES TIRED.

I think, on the contrary, your error messages are the most important part of your API. When things go wrong, we can't expect users to be delighted, but we can at least put a little effort into making the resulting error messages as helpful as possible.

So if we're to treat errors as a critical part of the system's behaviour, how can we test them? *What* should we test, and when?

At a minimum, our tests should check two things. One, that the system produces an error when it should. Two, that it *doesn't* produce one when it *shouldn't*.

That already sounds like two tests, doesn't it: one for the "want error" case, and another for "want no error". Let's apply this idea to testing some function `format.Data`, for example. We'll write both "want error" and "want no error" tests, and see what the typical pattern looks like for each.

Let's write "want error" first, because it's easier:

```
func TestFormatData_ErrorsOnInvalidInput(t *testing.T) {
    _, err := format.Data(invalidInput)
    if err == nil {
        t.Error("want error for invalid input")
    }
}
```

(Listing `format/1`)

And that's it! No need to go on and compare `want` with `got`, for example. We already know that `got` will be empty or invalid, and indeed we mustn't even point at it.

That's why we ignore the other return value using the blank identifier `_`. We don't plan to check that value, because the only thing this test cares about is that `err` is not `nil` under these circumstances.

For the "want no error" case, though, it makes sense to check that the function also produces the right result, provided that we bail out if it produces an error instead:

```
func TestFormatData_IsCorrectForValidInput(t *testing.T) {
    want := validInputFormatted
    got, err := format.Data(validInput)
    if err != nil {
        t.Fatal(err)
    }
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing `format/1`)

Note that even though we're not *expecting* an error in this case, we still receive it and check it, and we use `t.Fatal` to short-circuit the test if it's not `nil`. That's because what *this* test cares about is not only that the function doesn't return an error, but also that it returns the right result. Both of those things are essential to the function's public behaviour in the "valid input" case.

Checking the error here catches bugs in the function's behaviour, which is important, but that's not the only reason to do it. It's always possible that the *test* could be wrong, too. A bug in the system is bad enough, but a bug in the test is much worse. It means you can no longer reliably *detect* bugs in the system.

Simulating errors

If a function is *supposed* to return an error in certain situations, then in order to test that behaviour, we need to arrange for that error to happen.

Sometimes this is easy. For example, if the function is supposed to return an error for invalid input, as in the `format.Data` example, we can just *give* it invalid input and see what it does.

In other cases it can be a little more difficult to arrange for the specified error to occur. For example, consider a function like this:

```
func ReadAll(r io.Reader) ([]byte, error) {
    data, err := io.ReadAll(r)
    if err != nil {
        return nil, err
    }
    return data, nil
}
```

(Listing reader/1)

If the function encounters a read error, it returns it. Now, we probably wouldn't bother to test this behaviour in practice, because it's so straightforward. But let's pretend we do, just for the exercise. How could we arrange for a read error to happen in a test?

That's not so easy if we use something like a `strings.Reader`:

```
func TestReadAll_ReturnsAnyReadError(t *testing.T) {
    input := strings.NewReader("any old data")
    _, err := reader.ReadAll(input)
    if err == nil {
        t.Error("want error for broken reader, got nil")
    }
}
```

(Listing reader/1)

This will always fail, even when the function is correct, because reading from a `strings.Reader` never *does* produce an error.

How can we make the `strings.Reader` return an error when someone calls its `Read` method? We can't. But we can implement our *own* very trivial reader to do that:

```
type errReader struct{}

func (errReader) Read([]byte) (int, error) {
    return 0, io.ErrUnexpectedEOF
}
```

(Listing reader/2)

This is manifestly useless as a *reader*, since it never reads anything, always just returning a fixed error instead. But that makes it just the thing to use in our test:

```
func TestReadAll_ReturnsAnyReadError(t *testing.T) {
    input := errReader{} // always returns error
    _, err := reader.ReadAll(input)
    if err == nil {
        t.Error("want error for broken reader, got nil")
    }
}
```

(Listing reader/2)

Because `io.Reader` is such a small interface, it's very easy to implement it with whatever behaviour we need in a test. You could imagine more sophisticated kinds of fake reader, such as one that errors after a certain number of bytes have been read, or a `ReadCloser` that fails to close itself properly, and so on.

In fact, we don't even need to implement this erroneous reader ourselves. It's provided in the standard library `iotest` package as `ErrReader`, along with some other useful test readers such as `TimeoutReader` and `HalfReader`.

Testing that an error is not `nil`

There's another kind of mistake it's easy to make when testing error results, and it comes from a perfectly understandable thought process. "My function returns something and error," says the programmer, "and tests are about comparing `want` and `got`. So I should compare both results against what I expect them to be."

In other words, they'll compare the "something" result with some expected value, which is fine. And they'll also try to compare the "error" result with some expected

value, which is where they run into a problem.

Let's see an example. Remember our `store.Open` example from earlier on? It returns a `*Store`, if it can successfully open the store, or `nil` and some error if it can't.

Suppose we're writing a "want error" test for this behaviour. In other words, we'll deliberately try to open some store that can't be opened, and check that we get an error.

This is straightforward, because all we need to do is check that the `err` value is not `nil`. That's all the "contract" promises, so that's what we check:

```
func TestOpenGivesNonNilErrorForBogusFile(t *testing.T) {
    t.Parallel()
    _, err := store.Open("bogus file")
    if err == nil {
        t.Error("want error opening bogus store file")
    }
}
```

(Listing `store/1`)

As with the `format.Data` example, since we *expect* an error, we just ignore the other result value. We aren't interested in it for this test.

Indeed, if we received it and assigned it to `got`, then the compiler would complain that we don't then *do* anything with that variable:

```
got declared and not used
```

It would be easy to feel bullied by this into doing something with `got`, like comparing it against a `want` value. But *we* know that's wrong.

What the compiler is really saying, in its gnomic way, is "Hey programmer, since you don't seem to care about the value of `got`, you should ignore it using `_`." Quite right, and that's what we'll do.

But supposing we (wrongly) think that we need to test that `Open` returns some *specific* error. How would we even do that? Well, we can try the want-and-got pattern:

```
func TestOpenGivesSpecificErrorForBogusFile(t *testing.T) {
    t.Parallel()
    want := errors.New("open bogus: no such file or directory")
    _, got := store.Open("bogus")
    if got != want {
        t.Errorf("wrong error: %v", got)
    }
}
```

(Listing `store/1`)

This seems plausible, I think you'll agree. Yet, surprisingly, the test *fails*:

```
wrong error: open bogus: no such file or directory
```

Wait, what? That *is* the error we were expecting. So why are these two values not comparing equal?

Let's make the problem even more explicit, by using `errors.New` to construct both values, and *then* comparing them:

```
func TestOneErrorValueEqualsAnother(t *testing.T) {
    t.Parallel()
    want := errors.New("Go home, Go, you're drunk")
    got := errors.New("Go home, Go, you're drunk")
    if got != want {
        t.Errorf("wrong error: %v", got)
    }
}
```

(Listing `store/1`)

Surely *this* can't fail? On the contrary:

```
wrong error: Go home, Go, you're drunk
```

We might justifiably feel a little puzzled at this. We have two values constructed exactly the same way, and yet they *don't compare equal*. Why not?

This is somewhat non-obvious behaviour, I admit, but Go is doing exactly what it's supposed to. We can work out what's happening, if we think it through.

What type does `errors.New` return? We know it implements the interface type `error`, but what *concrete* type is it? Let's look at the source code of `errors.New` to see what it does:

```
func New(text string) error {
    return &errorString{text}
}
```

I bet you didn't know the unexported struct type `errorString` existed, yet you've probably used it every day of your life as a Go programmer. But that's not the important part.

The important part here is that `errors.New` returns a *pointer*. So when we construct error values by calling `errors.New`, the results we get are pointers.

Can you see now why they don't compare equal? In fact, Go pointers *never* compare equal unless they point to the same object. That is, unless they're literally the same memory address.

And that won't be the case here, because they point to two distinct instances of the `errorString` struct that merely happen to contain the same message. This is why it doesn't make sense to compare error values in Go using the `!=` operator: they'll never be equal.

What the programmer really wanted to know in this case was not "Do these two values point to the same piece of memory?", but "Do these two values represent the *same error*?"

How can we answer that question?

String matching on errors is fragile

Here's another attempt to answer the question of whether two values represent the same error:

```
func TestOpenGivesSpecificErrorStringForBogusFile(t *testing.T) {
    t.Parallel()
    want := errors.New("open bogus: no such file or directory")
    _, got := store.Open("bogus")
    if got.Error() != want.Error() {
        t.Errorf("wrong error: %v", got)
    }
}
```

(Listing `store/1`)

Instead of comparing the error values directly, we compare the *strings* produced by each error's `Error` method.

But although this works, more or less, there's something about it that doesn't feel quite right. The whole point of `error` being an interface, after all, is that we shouldn't *have* to care about what its string value is.

> *I believe you should never inspect the output of the `Error` method. The `Error` method on the `error` interface exists for humans, not code. The contents of that string belong in a log file, or displayed on screen.*
> *Comparing the string form of an error is, in my opinion, a code smell, and you should try to avoid it.*
> —Dave Cheney, "Don't just check errors, handle them gracefully"

So if we *construct* our expected error using `errors.New` and some fixed string, and compare it with the error result from the function, that's really just the same as comparing two strings.

That works great, right up until some well-meaning programmer makes a minor change to their error text, such as adding a comma. Bang! Broken tests all over the place. By doing this kind of comparison, we've made the test brittle.

There are no bad ideas, as they say, but let's keep thinking.

Sentinel errors lose useful information

If string matching errors makes tests fragile, then maybe we could define some *named* error value for `store.Open` to return.

In other words, something like this:

```
var ErrUnopenable = errors.New("can't open store file")
```

This is called a *sentinel error*, and there are several examples in the standard library, such as `io.EOF`. It's an exported identifier, so people using your package can compare your errors with that value.

This makes the test quite straightforward, because we can go back to our nice, simple scheme of comparing error values directly:

```
func TestOpenGivesErrUnopenableForBogusFile(t *testing.T) {
    t.Parallel()
    _, err := store.Open("bogus")
    if err != store.ErrUnopenable {
        t.Errorf("wrong error: %v", err)
    }
}
```

(Listing `store/2`)

This works, provided `Open` does indeed return this exact value in the error case. Well, we can arrange that:

```
func Open(path string) (*Store, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, ErrUnopenable // losing information about 'err'
    }
    // ...
}
```

(Listing `store/2`)

This *looks* okay, but it's still not ideal. Actually, we lost some information here that might have been useful: specifically, *why* we couldn't open the file. We had that information in the `err` variable, but then we threw it away and returned the fixed value `ErrUnopenable` instead.

So what do users eventually see? Just the value of `ErrUnopenable`, which is fixed:

```
can't open store file
```

This is very unhelpful. *What* store file? Why couldn't it be opened? What was the error? What action could fix the problem? The user could be forgiven for feeling somewhat let down.

> *DR EVIL: Right, okay, people, you have to tell me these things, all right? I've been frozen for thirty years, okay? Throw me a frickin' bone here! I'm the boss. Need the info.*
> —"Austin Powers: International Man of Mystery"

Actually, it would have been more helpful simply to return the `err` value directly, because it already contains everything the user needs to know:

```
open store.bin: permission denied
```

Much more informative! Unlike `ErrUnopenable`, this tells us not only which specific file couldn't be opened, but also why not.

So a sentinel error value `ErrUnopenable` makes it possible to detect that *kind* of error programmatically, but at the expense of making the error message itself nearly useless. But did we even *need* to make this trade-off in the first place?

For example, do programs that use `store.Open` really *need* to distinguish "unopenable file" errors from other kinds of errors opening a store? Or is all they care about simply that there *was* some error?

Most of the time in Go programs, all we care about is that `err` is not `nil`. In other words, that there was some error. What it is *specifically* usually doesn't matter, because the program isn't going to take different actions for different errors. It's probably just going to print the error message and then exit.

Getting back to the `store.Open` test, the user-facing behaviour that we care about is that, if the store can't be opened, `Open` returns *some* error.

And that's easy to detect, in a test or elsewhere. We can just compare it with `nil`, which is where we started:

```
func TestOpenGivesNonNilErrorForBogusFile(t *testing.T) {
    t.Parallel()
    _, err := store.Open("bogus file")
    if err == nil {
        t.Error("want error opening bogus store file")
    }
}
```

(Listing `store/1`)

In other words, if an error is intended to signal something to the *user*, then a sentinel error probably won't be that helpful, because its value is fixed. We can't use it to convey any *dynamic* information that might help the user solve the problem.

And if the system doesn't need to *know* anything about the error except that it's not `nil`, then we don't need a sentinel error at all.

Detecting sentinel errors with `errors.Is`

What about when different errors *should* result in different actions, though? Even if most of the time this isn't necessary, it's worth asking how we *should* go about constructing distinct errors when that's appropriate.

For example, suppose we're writing a function that makes HTTP requests to some API; let's call it `Request`. Now, what could go wrong?

Lots of things. We may not be able to *make* the request for some reason (its URL might be invalid, for example). And we can also get various error responses from the remote API: invalid request, not found, unauthenticated, and so on.

Is it ever necessary to distinguish between these different errors and take some specific action? Maybe it is. For example, let's say that one of the possible error responses we can get is HTTP status 429 (Too Many Requests). That status code indicates that we are being ratelimited. That is to say, we've made more requests than the server allows in a given time period, so it's telling us to back off a little.

In other words, it's not *what* we're doing, it's just that the *timing* is wrong. So maybe we can recover from this situation. Instead of reporting an error to the user and giving up, we can *retry* the request (after a polite interval).

This requires us to be able to distinguish a ratelimit error from some other error. What error value should we return, then, so that the caller can detect that rate limiting is happening?

A sentinel value would be fine here, because it doesn't matter that we can't include any context-specific information in it. We don't need to. The error value itself conveys everything the caller needs to know:

```
var ErrRateLimit = errors.New("rate limit")

func Request(URL string) error {
    resp, err := http.Get(URL)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    if resp.StatusCode == http.StatusTooManyRequests {
        return ErrRateLimit
    }
```

```
    return nil
}
```

(Listing req/1)

It doesn't matter what the actual string value of `ErrRateLimit` is here, since users won't see it. But just in case it *is* ever printed out for some reason, we can at least give it a string value that identifies it as a ratelimit error.

Now, suppose we call `Request`, maybe from a test, and we want to check if the error it returns is `ErrRateLimit`. How do we do that?

First, we'll need to create some local HTTP server that just responds to *any* request with the ratelimit status code. This is where the very useful `httptest` package comes in:

```
func newRateLimitingServer() *httptest.Server {
    return httptest.NewServer(http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            w.WriteHeader(http.StatusTooManyRequests)
        }))
}
```

(Listing req/1)

Great. Now we have a test server whose URL we can pass to `Request`, and the result should always be `ErrRateLimit`. But how can we *check* that result in a test?

Well, we already know it's safe to compare sentinel errors directly, using the == operator, and that would work. But let's try something different. Let's use the `errors.Is` function to make this comparison instead:

```
func TestRequestReturnsErrRateLimitWhenRatelimited(t *testing.T) {
    ts := newRateLimitingServer()
    defer ts.Close()
    err := req.Request(ts.URL)
    if !errors.Is(err, req.ErrRateLimit) {
        t.Errorf("wrong error: %v", err)
    }
}
```

(Listing req/1)

`errors.Is` can tell us whether some otherwise unspecified error *is* `ErrRateLimit`, which is what we want to know for this test.

If we had created this sentinel value *only* so that it could be checked by a test, that would be a smell. We generally don't want to complicate our public API just to make writing tests easier. And, as we've seen, if all that matters is that the function returns

some error in this case, we don't need a sentinel error. We can just check that `err` is not `nil`.

Here, though, we have a legitimate reason for making a sentinel error value part of our public API. It's valuable to users of our package, as well as for testing purposes.

And since we're saying it *is* an important part of our user-facing behaviour that the function returns this specific error value when it should, it needs to be tested.

So this is a legitimate application of sentinel errors, and even though we can't include any contextual information with them, they can still be useful, as in this example.

But is this the best we can do? Or is there a way we can use sentinel errors *and* add contextual information to them? Let's find out.

Wrapping sentinel errors with dynamic information

As we saw in the previous sections, a sentinel value like `ErrUnopenableFile` is pretty limited in what it can convey. It's just a fixed string, and sometimes that's fine.

But often we'd like to include some more dynamic information in the error. In our `store.Open` example, that might be the full `err` value returned from `os.Open`. How can we combine that information with a named sentinel error like `ErrUnopenableFile`?

As you probably know, we can use the `fmt.Errorf` function to construct an error value incorporating some formatted data:

```
return fmt.Errorf("user not found: %q", name)
```

That's great, but since the value we construct here will be different every time, how could we *identify* this error in code? That is, given some `err` value constructed in this way, how could the program detect that it represents a "user not found" error, as opposed to some other kind of error?

A sentinel value (`ErrUserNotFound`) wouldn't work by itself here, because it would never match the specific error. That will be different every time it's generated, because it includes a specific value of `name`. Comparing `err` against `ErrUserNotFound` with `errors.Is`, as we did before, wouldn't work here, because it's *not* the same value.

Returning a simple, fixed sentinel error also wouldn't help the user work out what's wrong. Instead, they'd like to see the specific *username* that couldn't be found: that's valuable debugging information.

And we can do that. We can take a sentinel value such as `ErrUserNotFound` and *wrap* it in a new error that contains the dynamic information, using a special format verb `%w`:

```
var ErrUserNotFound = errors.New("user not found")

func FindUser(name string) (*User, error) {
```

```
    user, ok := userDB[name]
    if !ok {
        return nil, fmt.Errorf("%q: %w", name, ErrUserNotFound)
    }
    return user, nil
}
```

(Listing user/2)

If we'd used the normal %v to interpolate the error value, that would have simply *flattened* it into a string, meaning that we could no longer compare it with ErrUserNotFound.

Instead, the %w verb tells fmt.Errorf to create a *wrapped* error: one that contains the dynamic information we give it, but also remembers what the "original" error was: ErrUserNotFound.

Wrapping errors like this is always safe, even if the code at the other end doesn't know that the error contains extra information. If you compare this error value with nil, or print it out, it acts just like a regular error.

But to those in the know, there's a way to *unwrap* the error and retrieve its original sentinel value. In fact, it's our friend errors.Is:

```
func TestFindUser_GivesErrUserNotFoundForBogusUser(t *testing.T) {
    t.Parallel()
    _, err := user.FindUser("bogus user")
    if !errors.Is(err, user.ErrUserNotFound) {
        t.Errorf("wrong error: %v", err)
    }
}
```

(Listing user/2)

Now we can see why using errors.Is beats comparing errors directly with the == operator. A direct comparison wouldn't succeed with a wrapped error. But errors.Is can inspect the error on a deeper level, to see what sentinel value is hidden *inside* the wrapping.

Can you take a wrapped error and wrap it some *more*, you might ask? Certainly you can. You can use fmt.Errorf and the %w verb as many times as you like, each time creating an extra layer of information over the original error. And errors.Is can unwrap them all.

This error-wrapping mechanism is elegant, because it allows us to include useful dynamic information in an error, while still making it easy to determine in code what *kind* of error it represents.

Custom error types and `errors.As`

Error wrapping is relatively new to Go; it was added in version 1.13. Before that, `errors.Is` wasn't available, so we had to roll our own way of combining sentinel errors and dynamic information.

We'll talk briefly about one example that you may come across, especially in older programs: the use of *custom error types*. It's no longer necessary to treat errors this way, but it'll be helpful for you to understand the idea if you should encounter it.

You already know that `error` is an interface, so you also know that any *user-defined type* can implement `error`, provided it has a suitable `Error` method.

So we could imagine creating some struct type, for example, where the *type* name conveys what *kind* of error it is, and the struct *fields* convey the extra information:

```
type ErrUserNotFound struct {
    User string
}

func (e ErrUserNotFound) Error() string {
    return fmt.Sprintf("user %q not found", e.User)
}
```

(Listing user/3)

Instantiating an error value like this is easy. We can simply return a struct literal of this type, containing the dynamic information we want to include:

```
func FindUser(name string) (*User, error) {
    user, ok := userDB[name]
    if !ok {
        return nil, ErrUserNotFound{User: name}
    }
    return user, nil
}
```

(Listing user/3)

Before the advent of the `errors` package, we would have inspected such error values using a *type assertion*, just as we would to find out the dynamic type of any interface variable in Go:

```
if _, ok := err.(ErrUserNotFound); ok {
    ... // it's a 'user not found' error
} else {
```

```
    ... // it's some other kind of error
}
```

That's okay, and it does the job. But the `errors` package now provides a better way to match the error type we're interested in. This time it's not `errors.Is`, but `errors.As`:

```
if errors.As(err, &ErrUserNotFound{}) {
    ... // this is a 'UserNotFound' error
}
```

We need to clearly distinguish these two similar-sounding functions, because they do different things. `errors.Is` tells you whether `err` is some specified error *value*, perhaps wrapped within many layers of information. On the other hand, `errors.As` tells you whether it's some specified error *type*.

If you're slightly confused about `Is` versus `As`, you're not alone. Maybe a helpful way to think about it is that `errors.Is` asks the question "**IS** it this error?", while `errors.As` asks "Is it the same type **AS** this error?"

To *test* for a custom error type using `errors.As`, for example, we could write:

```
func TestFindUser_GivesErrUserNotFoundForBogusUser(t *testing.T) {
    t.Parallel()
    _, err := user.FindUser("bogus user")
    if !errors.As(err, &user.ErrUserNotFound{}) {
        t.Errorf("wrong error: %v", err)
    }
}
```

(Listing `user/3`)

Custom error types do roughly the same job as wrapping errors with `fmt.Errorf` and the `%w` verb: they allow us to include dynamic information in errors, while still allowing comparison with a sentinel (in this case, a sentinel *type*). But they have some disadvantages.

For one thing, having to define a distinct struct type and associated `Error` method for every kind of error you need can be tedious and duplicative, especially if there are many of them.

By comparison, error wrapping takes almost zero effort. All you need to do is use the familiar `fmt.Errorf`, but change the `%v` verb to a `%w`:

```
return fmt.Errorf("file %q not found: %w", path, err)
```

Wrapping also has the advantage that, as we saw earlier, a single error can be wrapped many times, yet `errors.Is` can always tell us what sentinel value it started out as. This isn't true of custom error types.

So while it's good to understand why people used to create custom error types, there's no reason to do this in your own programs. Instead, use the standard error-wrapping mechanism provided by `fmt.Errorf` with `%w`.

Conclusions

So what have we learned in this chapter about generating, discriminating, and testing errors? There are four key points:

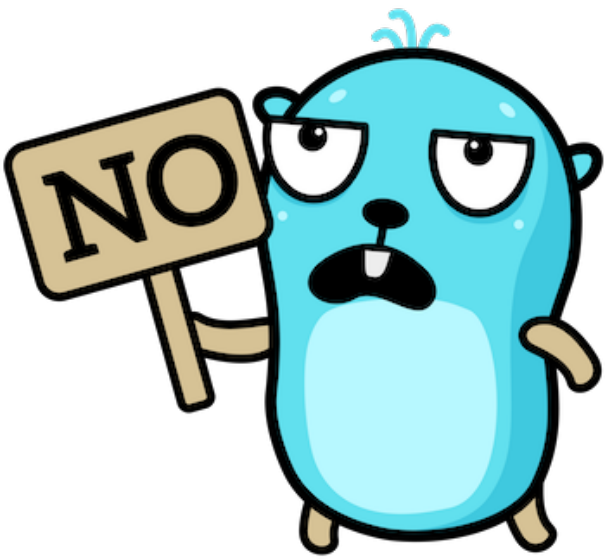
1. Always check errors in tests, whether they're expected or not.
2. It usually doesn't matter what they *are*, so long as they're not `nil`.
3. When it *does* matter what they are, use `errors.Is` to compare them with a sentinel error.
4. When you need to combine a sentinel error with some dynamic information, use `fmt.Errorf` and the `%w` verb to create a wrapped error.

The world is a messy place, and a lot of things can go wrong. Robust programs need to cope with those situations in a way that helps users.

And users themselves, as we'll see, can be an excellent source of new failing test inputs. We'll talk about this, and some of the techniques we can use to generate and organise test data, in the next chapter.

5. Users shouldn't do that

> *Ella discovered that dragging text from a text box on the screen and dropping it in another window resulted in a crash. She figured the tester would want to know about the problem, so she stopped by his cubicle and said, "Hey, let me show you something." She then demonstrated what she'd found.*
> *The tester stared at her for a few seconds and then replied. "Yeah?"*
> *"Don't you think it's a bug?"*
> *"No. Users shouldn't do that."*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"



Even the best-written tests are only as reliable as the cases they check. In this chapter, we'll talk about how to design good tests and test data, and introduce some techniques for generating them.

Constructing effective test inputs

> *Testing is, at best, sampling. Since we can't test everything, any set of real tests is some kind of sample. Fundamentally, sampling is a psychological process—and an*

> *emotional one. A sample that satisfies one person may not satisfy another in the slightest.*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

Since we usually can't test every possible input to the system, how should we decide which ones to test? Emotion and psychology should never be neglected in programming, but can we also introduce some logic to guide the sampling process?

As we've seen in previous chapters, one way to generate useful test inputs is to imagine *what could be wrong* with the system. For example, could it be a maliciously lazy implementation that simply returns hard-coded responses? To catch these, we can simply supply more inputs. But which?

Perhaps we just choose a few random values from the *input space*, that is, the set of all possible inputs that the system could take. On the other hand, maybe not all inputs are equally likely to trigger problems. For example, given a function that calculates square roots, we might test it all day long with positive numbers and never find a failure. But the moment we pass it a negative number, it might panic, or return the wrong answer.

The computer-sciency term for this is *equivalence class*. In the square root example, all positive inputs are of equivalent value for testing. If the function works for a handful of them, it will probably work for all. But negative numbers are in a different equivalence class: the result of the square root operation is not defined for them.

In other words, knowing where to usefully sample the input space depends partly on knowing how the program actually works.

> *Devising good test cases demands the same rigorous thinking as programming in general.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

When choosing our test data, then, we should try to pick a good sample of inputs from each equivalence class. *Partitioning* the input space in this way helps to reduce the number of inputs we need to get a given level of confidence in the system's correctness.

Another good place to look is the *boundary* between equivalence classes. For example, boundary values for a square root function might be 1, 0, and -1. We've all made off-by-one errors in programming. If there's such an error in the system, choosing inputs on or near boundaries will help flush it out.

A good tester thinks *adversarially* when designing test inputs. Perhaps users won't deliberately be trying to crash your software, but sometimes it can seem that way. If *you* can't think up some unlikely and bizarre inputs that crash the system, your users certainly will.

> *A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers.*
> —Brenan Keller

This especially applies to functions that validate their inputs, because the validation may well be insufficient. For example, consider this code:

```
func DoRequests(reqs int) error {
    if reqs == 0 {
        return errors.New("number of requests must be non-zero")
    }
    ... // do requests
}
```

What's wrong with this? If it's well tested, we will have already verified that it does the right thing with values of `reqs` such as 5, 10, or 100. And because the error behaviour is part of the API, we will also have tested that `DoRequests(0)` returns an error. So what are we missing?

To figure that out, think like a QA engineer ordering a beer... or, in this case, ordering -1 beers. No sane person would ever ask the function to do a *negative* number of requests, would they? What would that even mean? I don't know, but since it's *possible*, we need to make sure the system does something sensible in that case, such as returning an error.

It happens, in this example, that we can use Go's type system to enforce the requirement that `reqs` be non-negative, by declaring it as `uint` instead of `int`:

```
func DoRequests(reqs uint) error {
```

A test that tries to call `DoRequests` with a negative number, such as -1, won't even compile when we do this. Instead, we'll get a build error:

```
cannot use -1 (untyped int constant) as uint value in argument
to DoRequests (overflows)
```

This is a big advantage of *statically typed* languages such as Go: we can use the compiler to enforce a number of useful preconditions, such as non-negativity. We can think of the type system, indeed, as being a key *part* of our tests:

> *The type system is the set of tests that the language designer could think up without ever seeing your program.*
> —Michael Feathers, "Testing Patience"

So that's one way to solve the "feeble validation" problem, but maybe not the best way. The fact that we can't *compile* a test like `DoRequests(-1)` means we can't use that case as a regression test, to catch any future bugs we might introduce to the validation code.

So it may be a better idea to handle this by tightening up our `if` statement instead. It should check not just that `reqs` is not *zero*, but that it's also not *negative*:

```
func DoRequests(reqs int) error {
    if reqs < 1 {
        return errors.New("number of requests must be >= 1")
    }
    ... // do requests
}
```

When designing test inputs, then, try to think of things that nominally can't happen, or that only a crazy person would do, and test them. If you don't, then as we've seen, your users will test them for you. It's been wisely said that a good programmer is one who looks both ways before crossing a one-way street.

User testing

Another good source of crazy user inputs is actually using the software *yourself*, to the extent that it's usable at this point. If you have any kind of a runnable program yet, run it, and see what inputs you can find that cause breakage, or other weird behaviour. Mash the buttons until something interesting happens. You probably won't have to mash too long. Once you've found a bug, add a test case for it, make it pass, and go back to button-mashing.

Margaret Hamilton, a NASA programmer and one of the most distinguished software engineers of the 20th century, describes an incident that occured while she was developing the flight control software for the Apollo spacecraft:

> *I would take my six-year-old daughter to work on nights and weekends, and she would watch me testing out the software. She got very interested, and started "playing astronaut", and all of a sudden the system crashed.*
>
> *I realised she had selected P01 (a pre-launch program) during "flight". I started to worry about what if the astronauts did what she just did, and I went to the management saying "I think we need to put a change in." They said, "Won't ever happen. The astronauts are well-trained, they don't make mistakes."*
>
> *On the next mission, Apollo 8, this very thing happened.*
> —Margaret Hamilton, "The Language as a Software Engineer"

The lesson is clear: users have a blithe disregard for what they're not supposed to do, whether they're astronauts or six-year-olds. Don't listen to anyone who tells you not to worry about something because "users shouldn't do that". Expect the impossible, and test for it.

You can improve the user experience of your programs a great deal by simply *being* a user of them. What's the first thing most people do with some new program? Run it without any arguments, perhaps. Try this with one of your own programs. Does it do something helpful? Something unhelpful? Crash? Delete files without asking?

Many developers find it hard to put themselves into the mindset of a typical user, yet this is easier than you might think. After all, you and I *are* users of other people's programs, and we know how frustrating an experience that can be from time to time. We don't know what to do, so we try something apparently sensible, but it doesn't work. *We* don't know that users shouldn't do that.

When you use your own software in this way, you're bound to find all sorts of problems. Better to find them early, before it gets too expensive to fix them.

Don't rely on *actual* users to submit bug reports: most people don't do this. Instead, if your software doesn't seem to be working for them, they'll usually just find some other way to solve their problem. So an absence of user bug reports doesn't mean your program is perfect. It's just that users don't care as much about your program's correctness as you do.

Put yourself in the place of a user who wants to solve their problem, yet knows nothing about your program, and try things that they might try. Do those things work? If not, make them work. Just as writing test code will show what's wrong with your API, *using* your program will show you what's wrong with its user interface.

> *Roll up your sleeves and try to break your application. Use a destructive mindset and come up with ways to provoke issues and errors in your application. Document everything you find for later.*
> *Watch out for bugs, design issues, slow response times, missing or misleading error messages and everything else that would annoy you as a user of your software.*
> —Ham Vocke, "The Practical Test Pyramid"

Above all, don't dismiss the problems you find as mere "usability issues", to be solved by better documentation, or training. They are *bugs*. Fix them, guided by tests.

> *The developer said, "After a few crashes, they'll learn. Smart people learn fast not to do things that make the system crash."*
>
> *Developers can't control what "smart" people will learn. Perhaps users will learn to assume that the product could crash and lose data at any moment. Perhaps they will learn to avoid using this product at all. Perhaps they will learn the merits of working with a competing product.*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

And the quicker you can get the software in front of *real* users (that is, someone who's not you), the better. They will certainly find all sorts of problems you wouldn't have thought of. Fixing them now will be easier and cheaper than fixing them later.

If you can't find real users to test the system on, your fellow programmers will do nearly as well. Enlist their help:

> *A hallway usability test is where you grab the next person that passes by in the hallway and force them to try to use the code you just wrote. If you do this to five people, you will learn 95% of what there is to learn about usability problems in your code.*
> —Joel Spolsky, "The Joel Test: 12 Steps to Better Code"

If possible, be present while they try out the software, but in any case have them record their screen and voice during the test. Give them one or two typical user problems to solve using the system, and ask them to give a running commentary on what they're thinking and what they're trying to achieve at each stage. You should stay silent and resist the urge to intervene and help them. Remember, the goal is to *discover* what they can't do without your help.

You'll find this kind of feedback is invaluable, though you'll also feel a strong temptation to say to yourself, "But users shouldn't do that!"

User testing doesn't stop when the software goes to production—indeed, that's when it really starts. Make sure that all the responsible developers get detailed reports from the tech support department on every problem that users have reported. Don't let some well-meaning person filter out things like usability problems before they get to you: insist on seeing every support ticket that relates to the parts of the system you're responsible for.

You'll find lots of bugs of all kinds this way, and, naturally, you should fix them guided by tests, so that they're fixed not just for now, but forever.

Crafting bespoke bug detectors

We've discussed choosing effective test inputs in an adversarial way, picking likely boundary values, and so on. Depending on the kind of behaviour under test, there are also lots of classic "weird inputs" that professional testers know and love: long strings, accented characters, huge numbers, illegal values of all kinds. If the program asks for an IP address, try giving it `999.999.999.999` and see what happens.

We can and should attack the system with all these kinds of adversarial inputs, but we can do even more. Because we know exactly how the system is implemented, we have a good idea of what kind of bugs might be lurking in it.

If the code ranges over a slice, try passing it a `nil` slice, or one with no elements. If it takes a pointer, give it `nil`. If it writes to files, check what would happen if there is no disk space available or the filesystem is not writable (this is hard to simulate in a test, but you can check the code manually to see how it would handle the situation).

Read over the system code carefully, asking yourself at each stage "What could go wrong here?" Verify your ideas with tests. Even if it happens that the system *does* handle these problems right now, capturing them as tests ensures that no one can introduce such a bug in the future.

Table tests and subtests

Assuming we've managed to generate a bunch of effective test inputs, by various means, how should we organise them for testing in a logical, readable way?

In the first chapter we saw an example of a *table test*, one that performs the same check for each of a set of *test cases*. These are typically given as a slice of Go structs of some arbitrary type, chosen to suit the test.

We usually don't bother to name this type, since we won't *need* a name for it: it's just a convenient way of organising the data. Here's what our `ListItems` test cases look like if we rewrite them using an *anonymous struct*:

```
func TestListItems(t *testing.T) {
    t.Parallel()
    cases := []struct {
        input []string
        want  string
    }{
        {
            input: []string{},
            want:  "",
        },
        {
            input: []string{
                "a battery",
            },
            want: "You can see a battery here.",
        },
        ... // and so on
```

(Listing game/9)

For each case, we'll call `ListItems` with the `input` and check its result against `want`. If any case fails, we'll see the diff between `want` and `got`.

This is fine, but sometimes we'd like to be more explicit about precisely *which* test case is failing. In other words, as well as showing how `want` differed from `got`, we'd like to say something about which `input` caused this problem.

We could, of course, include `input` in the message generated by `t.Errorf` or `t.Fatalf`. But since there might be several places where the test can fail, it's tedious to include the input in each message, and for large `input` slices, it'll also obscure the message.

Even reporting the input that caused the failure might not be enough information. To be as helpful as possible, we'd like to give a descriptive *name* to each test case. One way to do this is to set out our test cases in the form of a *map*, keyed by suitably informative strings:

```
func TestListItems(t *testing.T) {
    t.Parallel()
    cases := map[string]struct {
        input []string
        want  string
    }{
        "no items": {
```

```
            input: []string{},
            want:  "",
        },
        "one item": {
            input: []string{
                "a battery",
            },
            want: "You can see a battery here.",
        },
        ... // and so on
```

(Listing game/10)

This is already an improvement, because we could simply loop over these as we did before and print out the *name* of the case for each failure. But we can do better, by using the `t.Run` method:

```
for name, tc := range cases {
    t.Run(name, func(t *testing.T) {
        got := game.ListItems(tc.input)
        if tc.want != got {
            t.Error(cmp.Diff(tc.want, got))
        }
    })
}
```

(Listing game/10)

Note first that in looping over a map with `range` we get two values, the `name`, and the `tc` struct it identifies. Inside this loop, we then call `t.Run` with two arguments: the name we just obtained, and a function literal of this form:

```
func(t *testing.T) { ... }
```

This function is called a *subtest* in Go. When any subtest fails, its name is printed out automatically as a prefix to the failure message:

```
--- FAIL: TestListItems/two_items (0.00s)
    game_test.go:46:   strings.Join({
        "You can see here a battery",
    +   ",",
        " and a key.",
        }, "")
```

In this example, we can see that the parent test is `TestListItems`, and this particular failing case—this subtest—is named `two items`. In the failure output, any spaces in the name of a subtest are replaced with underscores, so this is printed as 'two_items'.

The subtest name often describes the inputs, but we can use it to add any labelling information that would be helpful. For example, if this case reproduces a reported bug, we could use the bug ID as the subtest name. Or if the inputs are chosen to trigger a specific problem, we could name the subtest after that problem (for example, "invalid input data").

An interesting side effect of organising our test cases as a Go map is that the `range` loop will iterate over them in some unspecified order. After all, map keys *have* no inherent ordering, and that's useful in this context.

Suppose there were some hidden bug in the system that caused our test cases to pass only when run in a certain order. Looping over a map of cases would be likely to expose this, since the iteration order is unstable and is likely to change from one run to the next.

Table tests group together similar cases

Table tests are a powerful idea, but sometimes people get a bit too attached to them, and start trying to turn *everything* into a table test. For example, they might try to mix valid and invalid input cases in the same test:

```
tcs := []struct {
    {
        input: -1,
        wantError: true,
    },
    {
        input: 0,
        wantError: false,
    },
    ... // and so on
}
```

This can lead to some complicated logic in the subtest body, because now there are two kinds of possible cases: ones with valid input that are expected to succeed, and ones with invalid input that are expected to fail. In other words, the subtest would need to look something like this:

```
if tc.wantError {
    ... // test as an invalid input case
} else {
    ... // test as a valid input case
}
```

That's complicated both to write and to read, and also unnecessary. We can simplify the whole thing by just splitting it into *two* tests, one for valid input and one for invalid input. Each can have subtests, but now the subtest body itself is shorter and simpler, because it knows exactly what kind of case it has to deal with.

Watch out for this kind of smell in your own tests. Any time we have some `if` statement in the test body that selects different test logic based on *something about the case*, we should probably refactor into multiple tests, one for each *kind* of case we have.

Conversely, we can say that any time we're testing *the exact same thing* about a set of different inputs, we have a good candidate for a table test.

Just a note of warning: many IDEs have built-in facilities for generating table tests automatically. Indeed, the only *kind* of test they seem to know about is a table test. For example, if you put the cursor on some function in VS Code and run the *Go: Generate Unit Tests for Function* command, it generates a table test with a slice of test case structs, inviting you to fill in the missing data.

I don't recommend you use this feature. It can pressure you into trying to test the *function*, not its *behaviour*, by jamming different kinds of cases into a single test where they don't really belong. And, as we've seen, named subtests are better organised as a map than as a slice.

Using dummy names to mark irrelevant test inputs

In the examples we've seen so far, the test inputs and expectations have been pretty small, and we've simply quoted them directly in the test as literals. What kinds of data does it make sense to use in this situation?

Actually, we should distinguish between two kinds of test data: *relevant* to the test, and *irrelevant*. For example:

```
func TestGreetingReturnsCorrectGreetingForLanguage(t *testing.T) {
    t.Parallel()
    u := user.New(" 刘慈欣")
    u.Language = "Chinese"
    want := " 你好"
    got := u.Greeting()
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing user/4)

In this test, we create some user named 刘慈欣, set his preferred language to Chinese, and check that the system's generated greeting for him is "你好", which should make all

civilised persons feel right at home.

We have two pieces of test input data here: the user's name, and his language setting. Which of these is relevant to the test, and which is irrelevant? It's not instantly clear to the reader.

In fact, it turns out that the user's *name* has no effect whatosever on the greeting generated by the system. That is entirely determined by the language setting.

So the user's name is irrelevant: changing it wouldn't affect the test. We could signal that by using an obviously fake name as the data:

```
u := user.New("Fake User")
```

That makes it really clear that the value passed to `CreateUser` can't affect the test, because we'd be unlikely to write a system that has some special behaviour when the username is "Fake User". If we had, that would be a mistake, because we can't safely assume that no real person has that name.

On the other hand, the language value "Chinese" *is* important to this test. If we changed it to "Spanish", we might instead expect a greeting like "*¿Que tal?*"

If the language were *not* relevant, on the other hand, we could indicate that by using an obviously fake value here:

```
u.Language = "Bogus Language"
```

And the fact that we *haven't* done that tells the reader to pay attention to the `Language` value. By using the name of a real language, we're signalling that it's relevant to the behaviour under test.

Another example: suppose we have some `CallAPI` function that takes an authentication token. And suppose the test looks like this:

```
token := "eyJhbGciOiJIUzI1N"
got, err := CallAPI(req, token)
```

Is the value of `token` relevant to the test here? We can't tell. It *looks* like it could be a real token. On the other hand, if we used a different token value, then maybe it would affect the test, and maybe it wouldn't.

Instead, let's send a clear signal to the reader, by choosing a value that's *obviously* fake:

```
got, err := CallAPI(req, "dummy token")
```

Now the reader can easily infer that `CallAPI` doesn't care about the token, at least under the conditions of this test. It has to be there syntactically, and an empty string wouldn't have been a useful substitute: the reader might well wonder if it's important that it be empty.

By using the string "dummy token", we spell out the fact that it doesn't matter. Small signals like this, consistently applied, can make it much easier to readers to understand the purpose of a test.

> *Use data that makes the tests easy to read and follow. You are writing tests to an audience. Don't scatter data values around just to be scattering data values around. If there is a difference in the data, then it should be meaningful.*
> —Kent Beck, "Test-Driven Development by Example"

Outsourcing test data to variables or functions

Sometimes the data can be quite large, and putting it in the test code can be distracting and create clutter. What could we do instead?

One answer is to use *variables* for the data. This has the advantage that we can give the variable an informative name (`validInput`), and by moving it outside the test, up to package level, we keep the test free of visual clutter.

For example:

```
func TestParseHandlesValidInput(t *testing.T) {
    got, err := parse.Parse(validInput)
    ... // check the results
}

var validInput = `TGlzdGVuaW5nIGlzIGxvdmUuCuKAlEplYW4gS2xlaW4=`
```

This is helpful when it doesn't matter what the data actually *is*. In other words, the exact contents of the string `validInput` are irrelevant to the test. What the reader needs to know is simply that it is an example of *valid input*.

Clarity in code is often as much a matter of what you *don't* say as of what you do. By "not saying" the value of the input data, we avoid distracting the reader, who might otherwise wonder "Do I need to read this string? Are its contents significant in some way?" No, and no, our test says.

It often happens that many tests can use the same pieces of data. For example, there might be several tests that can use the `validInput` variable, and by doing so we helpfully declutter all those tests as well as reducing duplication.

Note that we gave the variable definition *after* the test that uses it. In general, it's a good idea to put your test functions *first* in any test file, and then follow them with all the variables, helpers, and so on. The ordering makes no difference to Go, but it makes a big difference to readers, who expect to see the most important things first.

Sometimes what's relevant about a piece of data is not its actual contents or validity, but just that it's very large. For example, to test that a function doesn't blow up when given a string of length 10,000, we'd like to have such a string defined as a variable.

It would be annoying to have to type out such a string, and wasteful to store it in the source code, so a better idea is just to use some code to generate it on the fly:

```
var longInput = strings.Repeat("a", 10_000)
```

If generating the data is a little more complicated, perhaps involving several steps, we can call a function literal to construct it instead:

```
var complicatedInput = func() string {
    ... // lots of steps here
    return result
}()
```

Note the trailing parentheses `()`: the value of `complicatedInput` will be the *result* of this function, not the function itself. Tests that use this input data can reference the variable directly:

```
func TestParseHandlesComplicatedInput(t *testing.T) {
    got, err := parse.Parse(complicatedInput)
    ... // check the results
}
```

Because the `var` statement is executed just once, when the program starts, we don't need to wastefully re-compute that complicated input value every time a test needs it.

A danger lurks here, though. A global variable is a good way to share data between tests, when it *can* safely be shared. On the other hand, some types of data are *not* safe to share.

For example, maps and slices are *reference types*: even if you create copies of the value, they all refer to the same underlying piece of memory.

Suppose we need some map of `string` to `int` in multiple tests, and we define it as a global variable in the test package, like this:

```
var ageData = map[string]int {
        "sam": 18,
        "ashley": 72,
        "chandra": 38,
}
```

Even parallel tests can safely use this `ageData` variable, so long as they don't try to modify it. But if they do, or if the system code they're calling happens to modify its input, that will change the data stored in `ageData`. Now other tests will be using the modified data, which can lead to very confusing results.

Instead, we need to create a new, independent `ageData` map value for every test that wants to use it. We can do this by writing a function that returns the map:

```
func makeAgeData() map[string]int {
    return map[string]int{
```

```
        "sam":     18,
        "ashley":  72,
        "chandra": 38,
    }
}
```

It's worth noting that, unlike in the previous `complicatedInput` example, we're not *calling* the function here, merely defining it. That's because we don't want to create a single map value that will be re-used by every test.

Instead, we want each test to have a way to create its own independent map value, by calling the `makeAgeData` function:

```
func TestTotalReturnsTotalOfSuppliedAges(t *testing.T) {
    t.Parallel()
    want := 128
    got := ages.Total(makeAgeData())
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

The test (and the function) can now do whatever they like with the `ageData`, including modifying it, without worrying about corrupting the data used by other tests.

Use functions to safely construct any kind of test input that contains maps and slices, including structs whose *fields* are maps or slices.

Loading test data from files

Using a global variable is a good way to move bulky data out of the test function itself, but sometimes that data can become very large (say, more than a few dozen lines), or there can be many pieces of it. In this case, to avoid cluttering the source code, it's a good idea to put the test data in a *file* instead.

Conventionally, we put these files in a folder named `testdata`, since the Go tools will ignore any folder with this name when looking for packages. To open and load these files, we can use plain ordinary Go code:

```
data, err := os.Open("testdata/hugefile.txt")
if err != nil {
    t.Fatal(err)
}
defer data.Close()
... // read from 'data'
```

Note that if there's any error opening `hugefile.txt`, we need to stop the test with `t.Fatal` right away, since we won't be able to use the file as input.

Once we've successfully opened the file, we `defer` closing it, so as not to hold open any unnecessary file handles during the test run.

Since file operations tend to be at least an order of magnitude slower than reading the same data from memory, it's a good idea to avoid having tests open files unless it's really necessary.

In particular, avoid opening a file just to create an `io.Reader`, if that's what the function needs to take. Use `strings.NewReader` instead to create a reader from static data:

```
input := strings.NewReader("hello world")
// 'input' implements 'io.Reader'
got, err := parse.ParseReader(input)
...
```

Similarly, don't create a file just to get an `io.Writer` value. If the function under test takes a writer, and we don't care what goes into it, we don't even need a `strings.Reader`. Instead, we can just use the predeclared `io.Discard`, which is a writer to nowhere:

```
// writes to 'io.Discard' are ignored
tps.WriteReportTo(io.Discard)
```

On the other hand, if the test needs to look at the writer afterwards, to see what the function actually *wrote* to it, then we can use a `*bytes.Buffer`:

```
buf := &bytes.Buffer{}
tps.WriteReportTo(buf)
want := "PC Load Letter"
got := buf.String()
```

Readers and writers versus files

If you find yourself having to create or open a file because the *function under test* expects a `*os.File`, this might be a smell. Does the function really need a *file*, or does it just need an `io.Reader` or `io.Writer`?

It depends. If the function just wants to read bytes from somewhere, you can pass it an `io.Reader`. If it needs to write bytes somewhere (for example, printing text), use an `io.Writer`. This is convenient for users, since lots of things implement `io.Reader/Writer`. And it's handy for testing: a `*bytes.Buffer` implements both interfaces.

On the other hand, if the function is really about operating on *files*, not just streams of bytes, then we can't use those interfaces. For example, if it needs to call methods on its argument that are specific to `*os.File`, such as `Name`, or `Stat`, then we have to pass it a file.

A good place for such files is the `testdata` folder, and it's helpful to name them after the *kind* of input data they represent. For example, we might create two input files, `valid.txt`, and `invalid.txt`.

The filesystem abstraction

Another useful interface for functions to accept, especially when their job is to deal with *trees* of files on disk, is `fs.FS`—usually referred to as a *filesystem*.

I've covered Go filesystems in detail in The Power of Go: Tools, though if you haven't read that book yet, the relevant chapter is also available as a blog post titled Walking with Filesystems: Go's new `fs.FS` interface.

Suffice it to say, for our purposes, that `fs.FS` is a standard library interface that usually represents a tree of files on disk.

We use a filesystem when we want to recursively visit each file or directory in the tree it represents. For example:

```
fsys := os.DirFS("/home/john/go")
results := find.GoFiles(fsys)
```

In this example, `fsys` represents a real disk filesystem, but `fs.FS` can also be easily implemented by some in-memory structure. Since all I/O operations are much, much slower than accessing memory, and physical spinning disks are especially slow, we'd prefer not to do any unnecessary disk access in our tests.

A convenient in-memory filesystem is provided by `fstest.MapFS`:

```
fsys := fstest.MapFS{
    "file.go":                {},
    "subfolder/subfolder.go": {},
    "subfolder2/another.go":  {},
    "subfolder2/file.go":     {},
}
results := find.GoFiles(fsys)
```

The *function* doesn't care what's under the hood of `fsys`: for all it knows, these are real files on disk. But our test is much faster, because `fsys` isn't having to do any disk I/O: it's using a Go map instead, meaning that all its operations happen in memory.

Using `t.TempDir` for test output with cleanup

When you need to pass a function a pathname to *write* data to, the best choice is usually to call `t.TempDir` to get the name of a per-test temporary directory. As we've seen earlier in this book, the directory created by `TempDir` is automatically deleted at the end of the test, together with its contents.

Even though the file will be created in a temporary directory, it's a good idea to name the file itself after the test or subtest that creates it. That way, if the file ends up in the wrong place, or gets left lying around by mistake, you'll at least know where it came from. For example:

```
image.Generate(t.TempDir()+"/test_generate.png")
```

A useful method here is `t.Name`, which returns the name of the current test or subtest. You can use this instead of an arbitrary name:

```
image.Generate(t.TempDir()+"/"+t.Name()+".png")
```

Managing golden files

One common use of files in tests is as *golden files*. As we saw in an earlier chapter, a golden file contains the expected output from some process, and we can compare it against the *actual* output to see whether it's correct.

Ideally, we would get this golden data straight from some *test oracle*, such as the legacy system we're replacing. The test is then simply a matter of checking how close this is to the current output from the function:

```
func TestWriteReportFile_ProducesCorrectOutputFile(t *testing.T) {
    t.Parallel()
    output := t.TempDir() + "/" + t.Name()
    tps.WriteReportFile(output)
    want, err := os.ReadFile("testdata/output.golden")
    if err != nil {
        t.Fatal(err)
    }
    got, err := os.ReadFile(output)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
```

```
    }
}
```

(Listing tps/1)

This test reads both files into memory for simplicity, but this obviously limits the size of the data it can handle. If your program generates *huge* output files, you'll need to compare them in a more efficient way, such as computing their hash sums, or comparing them a chunk at a time. For example, you could use `io.ReadFull` to read a chunk of bytes at a time from each reader, and then `bytes.Equal` to compare them.

You'll sometimes see advice to implement an `-update` flag in your test package, to automatically update your golden files. The effect of this flag is to assume that the current output of the function under test is correct, and to use it to overwrite the contents of the golden file.

I don't think this is a good idea, partly because it adds complication and magic to your tests, and you'd really need a separate flag for each golden file you use. But the most serious objection to an `-update` flag is that it provides a very easy and automated way to render your tests useless.

Suppose, for example, that you make a minor change to some file generation routine that creates megabytes of output data. And suppose that you rigorously scrutinise every byte of that output to make sure it's correct, and finally you run `go test -update` to overwrite your golden file. Well, from now on your test will always pass, since you've *defined* the required behaviour as whatever the function currently does.

You might be very diligent in checking the output before committing an updated golden file, but someone else might not be so careful. Sooner or later, someone will make a mistake. And from that point on, we'll be in a very uncomfortable situation, because the test will say that everything's okay when, in fact, the system is incorrect.

Worse, if someone should change the system to have the *correct* behaviour, the test will immediately fail! It takes considerable strength of mind to overrule a failing test, and in any case a good deal of time will probably be wasted figuring out what's gone wrong.

The best golden file, then, is *real data*, taken from some external oracle or system. The next best is *hand-rolled data*, carefully constructed by someone who knows exactly what the output should look like. When this format needs to change, change the golden file manually.

The very last resort, and something we should avoid if at all possible, is to use the *system itself* to generate the golden data, for the reasons we've discussed. If this ever has to be done, the results should be checked rigorously, and then manually updated as the requirements evolve.

We don't want to make it easy to change this critical data, on which the correctness of the whole system depends. Rather, we *want* changing the golden data to be a big deal, so while an `-update` flag is a neat idea, I think the risks probably outweigh the benefits.

Dealing with cross-platform line endings

If you're managing your source code using Git, as you most likely are, there's another point to watch out for when using golden files. Git has a configuration setting named `autocrlf`, which auto-translates linebreak characters to suit the convention of the operating system you're using.

As you may know, Windows uses a different line-ending convention from other operating systems. Consequently, Git tries to help you out by *auto-translating* the line endings to something appropriate for your system.

This means that your golden files, if they contain text, will look different on different operating systems. When compared directly, they will produce a diff showing a missing `\r` on each line:

```
-    "Hello world\r\n",
+    "Hello world\n",
```

This can be puzzling if you're not aware of the line ending issue, since the tests will pass on non-Windows systems.

One solution to this is to add a `.gitattributes` file to the root directory of your repo, with the following contents:

```
* -text
```

This tells Git to treat all files (`*`) in the repo as *not* text (`-text`). In other words, to treat all files as binary, with no line ending translation. This should fix the problem, and users with up-to-date versions of Windows and text editors such as Notepad will have no problem editing the files if they want to.

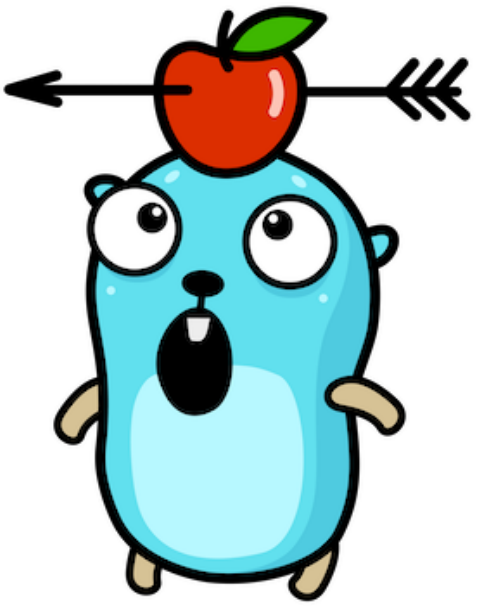
What about the inputs you didn't think of?

In this chapter we've discussed various ways of picking and organising *specific* input values, chosen in advance for their effectiveness at flushing out bugs in the system. And most of the tests we write will be like this, but for a sufficiently non-trivial problem, there will always be inputs we can't imagine or just don't happen to think of.

To supplement our hand-picked test inputs, then, it's also often useful to generate inputs *randomly* in various ways. We'll learn more about this process in the next chapter.

6. Fuzzy thinking

> *I shot an arrow into the air,*
> *It fell to earth, I knew not where.*
> —Henry Wadsworth Longfellow, "The Arrow and the Song"



In the previous chapter, we explored various strategies for manually choosing effective test inputs, but we also noted that there are certain limitations to this approach. There may be certain inputs that trigger a bug, but if we just don't happen to think of them, we won't find the bug.

> *By its nature, testing is never complete. As the influential computer scientist Edsger Dijkstra put it, "Testing shows the presence, not the absence of bugs." No quantity of tests can ever prove a package free of bugs. At best, they increase our confidence that the package works well in a wide range of important scenarios.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

Generating random test inputs

Sometimes, then, it can be a good idea to generate test inputs *randomly*. For example, here's a test that generates a random value and sends it on a *round trip* through two functions, `Encode` and `Decode`:

```
func TestEncodeFollowedByDecodeGivesStartingValue(t *testing.T) {
    t.Parallel()
    input := rand.Intn(10)
    encoded := codec.Encode(input)
    t.Logf("encoded value: %#v", encoded)
    // after the round trip, we should get what we started with
    want := input
    got := codec.Decode(encoded)
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing codec/1)

You might worry that, if the value is *truly* random, then we could end up with a flaky test. If there's a bug in `Encode` or `Decode` that's only triggered by certain inputs, then won't a test like this sometimes pass and sometimes fail?

That's a valid concern, but remember that Go's `math/rand` package uses a default *seed* (in fact, it's 1). This particular random generator produces a sequence of numbers that *appear* unpredictable, but in fact depend predictably on the initial seed value.

Such sequences are certainly not safe for use in cryptography, precisely *because* they're predictable (use `crypto/rand/` for crypto). But they're usually good enough for games, for example. They can be implemented with astonishingly little code.

The sequence of numbers produced by `rand.Intn` will always be the same for a given seed. That is to say, a program like this will always produce the same output (try it):

```
fmt.Println(rand.Intn(10))
fmt.Println(rand.Intn(10))
fmt.Println(rand.Intn(10))
```

Even though the sequence it generates will *look* pretty random, it will always be the same sequence, because it comes from the same seed (namely, 1).

In other words, the test is *deterministic*: if it passes today, it'll pass tomorrow, and every day thereafter. But the downside of that is that the sequence never changes. In order to get different sequences, we'd have to seed the random generator with some value that changes every time we run the program.

A common way to produce that ever-changing seed value is to get the current wall-clock time, by calling `time.Now`:

```
rand.Seed(time.Now().UnixNano())
input := rand.Intn(10)
```

Now `input` will have a different value (at least, one of 10 different values) every time we run the test.

One advantage of this is that if there *is* some value in the input space that triggers a bug, we should eventually find it. The downside is that it might take a while. The bigger the input space, the longer it takes to explore it by picking values at random.

And we usually prefer our tests to be deterministic (that is, non-flaky). So, rather than use changing random seeds, a better idea is to stick with the default random sequence, or a fixed seed value, making the outcome of the tests predictable.

To increase our coverage of the input space, and thus our chances of finding breaking test inputs, we can simply generate *lots* of values, and try them all.

Randomly permuting a set of known inputs

A nice way to use randomness without causing flaky tests is to *permute* a set of inputs—that is, rearrange them—in some random order.

For example, the following code generates a slice containing the integers from 0 to 99, ordered randomly:

```
inputs := rand.Perm(100)
for _, n := range inputs {
    ... // test with input n
}
```

The sequence of 100 integers generated by `rand.Perm(100)` is not random *itself*, since each value from 0 to 99 will be represented exactly once. That wouldn't be true of randomly *chosen* numbers, where some values would occur many times and others not at all.

Instead, this sequence is randomly *permuted* (that is, randomly ordered). You can decide for yourself whether you'd like to seed the random generator with a non-fixed value (for example, using `time.Now`). If so, each time you run the test it will try the inputs in a different order, which may be useful for finding ordering bugs in the system.

On the other hand, if you don't seed the random generator, then the sequence of inputs will always be the same, which is more conducive to reproducible builds.

Property-based testing

Randomness can be a good way of finding interesting new test cases that you might not come up with by yourself. For example, if your function takes integers, you might have tried testing it with 1, 5, 7, and so on. You might not have thought of trying *zero* as an input, but the random generator would likely produce that at some point. And if your function breaks when given zero, that's something you'd certainly want to know.

Good testers, in general, are good at suggesting inputs that break the program because the *programmer* didn't think of them. Randomness can help with that process.

One problem with randomised test inputs that may already have occurred to you is this: if we don't know in advance what input we're going to use, we can't know what answer to *expect* as the result.

For example:

```
func TestSquareGivesCorrectResult(t *testing.T) {
    t.Parallel()
    input := rand.Intn(100)
    got := square.Square(input)
    want := ... // uh-oh, what to put here?
```

What's our `want` value here? We don't know, because we don't know what the value of `input` will be when the test runs, and thus what its square would be. If we knew that `input` would be 10, for example, then we could have the test expect the answer 100, but we *don't* know that. We're stuck.

And we don't want to try to *compute* the expected result in the test, because clearly we could get that computation wrong. In the most pernicious case, we might end up using the same code path to compute it in the test as we do in the *system*, so we'd end up testing nothing at all.

If we can't predict what the *exact* result of `Square` will be for a given input, is there anything we can say about it in general terms?

Actually, there is something we can say: it shouldn't be negative! No matter what the input, if `Square` returns a negative result, something's gone wrong. So although we can't predict the *exact* result if the system is correct, we can still identify some *properties* it should have.

So we could write a test that calls `Square` with lots of different inputs, and checks that the result is never negative:

```
func TestSquareResultIsAlwaysNonNegative(t *testing.T) {
    t.Parallel()
    inputs := rand.Perm(100)
    for _, n := range inputs {
```

```
        t.Run(strconv.Itoa(n), func(t *testing.T) {
            got := square.Square(n)
            if got < 0 {
                t.Errorf("Square(%d) is negative: %d", n, got)
            }
        })
    }
}
```

(Listing square/1)

This approach is sometimes called *property-based testing*, to distinguish it from what we've been doing up to now, which we might call *example-based testing*.

Another way to think about it is that property-based tests describe the behaviour of the system, not in terms of exact values, but in terms of *invariants*: things that don't change about the result, no matter what the input is. In the case of `Square`, for example, its result should invariably be positive.

Randomised, property-based testing helps to fix the problem that maybe the function only works for the specific examples we thought of. And although we *generate* the inputs randomly, once we find some value that triggers a bug, it should then simply become part of our conventional example-based tests.

We could do this by manually adding such values to the set of inputs in a table test, for example, but there's no need. Go provides an automated way to turn randomly-generated breaking inputs into static test cases for us, using what's called *fuzz testing*.

Fuzz testing

> *Fuzz testing works by throwing everything at the wall and seeing what sticks.*
> —Xuanyi Chew, "Property-Based Testing"

Nobody's perfect, especially programmers, and while we try to be reasonably careful when writing code, we're bound to overlook things from time to time. Remember, in our adventure game example from the first chapter, how early versions of the `ListItems` function panicked when given slices of length zero or one?

That's a good demonstration of how bugs can arise when the programmer doesn't envision certain kinds of inputs, and thus doesn't test that the system handles them correctly. In other words, you can only find bugs you can *think* of.

Slice overruns are a classic example of the kind of bug that busy programmers can easily overlook:

```
results := SearchProducts(productID)
return results[0].Name // panics if there are no results
```

Indeed, this code might run perfectly well in production for months or even years, and then suddenly break when someone searches for a product that doesn't exist. It wasn't caught by any example-based tests because nobody *thought* about that situation.

But there's another kind of testing that can help: *fuzz testing*. The "fuzz" here refers not to data that's blurry, or imprecise, but means simply "randomly generated". Tools for fuzz testing have been around for decades, but built-in support for it was added to Go relatively recently, in version 1.18.

Could we boost our chances of finding lurking bugs in a Go function by *fuzzing* it—that is, by randomly generating lots of different inputs, and seeing what happens?

To find out, let's deliberately write a function that will panic when given a certain secret trigger value:

```
func Guess(n int) {
    if n == 21 {
        panic("blackjack!")
    }
}
```

(Listing guess/1)

Now, we'll pretend we don't know the special value, but we merely *suspect* that such a bug lurks within the function. We'll try to find it using randomised inputs. How could we do that?

We need to generate a large number of random values that decently cover the space of likely inputs, and call the function with each of them in turn. If one of them causes a panic, then we have found our bug.

It's not that we couldn't write this by hand, as in the previous examples, but instead let's try writing a *fuzz test*, using Go's standard `testing` package.

Here's what that looks like:

```
func FuzzGuess(f *testing.F) {
    f.Fuzz(func(t *testing.T, input int) {
        guess.Guess(input)
    })
}
```

(Listing guess/1)

The names of fuzz tests in Go start with the word `Fuzz`, just as regular tests start with the word `Test`. And the parameter they take is a `*testing.F`, which is similar to our friend the `*testing.T`, and has many of the same methods.

The fuzz target

What do we do with this `f` parameter, then? We call its `Fuzz` method, passing it a certain function. Passing functions to functions is very Go-like, and though it can be confusing when you're new to the idea, it's also very powerful once you grasp it.

So what's this function within a function? In fact, it's the function that the fuzz testing machinery (the *fuzzer*, for short) will repeatedly call with the different values it generates.

We call this function the *fuzz target*, and here is its signature:

```
func(t *testing.T, input int)
```

Remember, this isn't the *test* function, it's the function we *pass* to `f.Fuzz`. And although it must take a `*testing.T` as its first argument, it can also take any number of other arguments. In this example, it takes a single `int` argument that we'll call `input`.

These extra arguments to the fuzz target represent your inputs, which the fuzzer is going to randomly generate. In other words, the fuzzer's job is to call *this* function—the fuzz target—with lots of different values for `input`, and see what happens.

And here's the fuzz target we'll be using in this example:

```
func(t *testing.T, input int) {
    guess.Guess(input)
}
```

This very simple fuzz target does nothing but call `Guess` with the generated `input`. We *could* do more here; indeed, we could do anything that a regular test can do using the `t` parameter.

But to keep things simple for now, we'll just *call* the function under test, without actually checking anything about the result. So the only way this fuzz target can fail is if the function either panics, or takes so long to return that the test times out.

Running tests in fuzzing mode

To start fuzzing, we use the `go test` command, adding the `-fuzz` flag:

```
go test -fuzz .
```

Note the dot after the `-fuzz`, which is significant. Just as the `-run` flag takes a regular expression to specify which tests to run, the `-fuzz` flag does the same. In this case, we're using the regular expression "`.`", which matches all fuzz tests.

Here's the result:

```
warning: starting with empty corpus
fuzz: elapsed: 0s, execs: 0 (0/sec), new interesting: 0 (total: 0)
```

```
--- FAIL: FuzzGuess (0.05s)
    --- FAIL: FuzzGuess (0.00s)
        testing.go:1356: panic: blackjack!
        [stack trace omitted]

Failing input written to testdata/fuzz/FuzzGuess/xxx
    To re-run:
    go test -run=FuzzGuess/xxx
```

The first thing to note about this output is that the test failed. Specifically, it failed with this panic:

```
panic: blackjack!
```

This is the panic message that we deliberately hid inside the `Guess` function. It didn't take too long for the fuzzer to find the "secret" input that triggers the panic. But what was it?

This isn't shown as part of the output, but there's a clue:

```
Failing input written to testdata/fuzz/FuzzGuess/xxx
```

Instead of `xxx`, the actual filename is some long hexadecimal string that I won't weary you by reproducing in full. We don't normally need to look at the contents of these files, but just for fun, let's take a peek:

```
go test fuzz v1
int(21)
```

The first line identifies what *kind* of file this is: it's generated by `v1` of the Go fuzzer. Including a version identifier like this is a good idea when generating any kind of machine-readable file, by the way. If you decide you'd like to change the file format later, you can alter the version number to maintain compatibility with older versions of the reading program.

The second line contains the information we're really interested in:

```
int(21)
```

This gives the specific input value represented by this test case. It's an `int`, which is no surprise, since that's what our fuzz target takes, and the value that triggered a failure is the magic 21. Winner, winner, chicken dinner!

Failing inputs become static test cases

> *Fuzz testing is great at discovering unexpected inputs that expose weird behaviour.*
> —Jay Conrod, "Internals of Go's new fuzzing system"

Why has the fuzzer generated this `testdata` file, then? Why not just report the failing input as part of the test results, for example?

Well, suppose we grow bored of fuzzing for the time being, and we decide to just run our plain old tests again, without the `-fuzz` flag:

```
go test
```

```
--- FAIL: FuzzGuess (0.00s)
    --- FAIL: FuzzGuess/xxx (0.00s)
panic: blackjack! [recovered]
```

Wait, what? Why is it running the fuzz test, if we didn't say `-fuzz`?

The answer, as you may have guessed, is that fuzz tests can also act as regular tests. When run in non-fuzzing mode (that is, without the `-fuzz` flag), they don't generate any random inputs. What they do instead is call the fuzz target with any and all inputs that have *previously* been found to trigger a failure. These are the inputs stored in the special files under `testdata/fuzz`.

As it happens, there's only one such input at the moment (21), but it's enough to fail the test. A good way to think about the fuzzer is that its job is to *generate failing test cases*, which then become part of your normal example-based tests.

In other words, once some randomly-generated failing input has been found by the fuzzer, running `go test` will test your function with that input ever afterwards, unless you choose to remove the `testdata` file manually.

Let's fix the function so that it no longer panics on 21, or indeed does anything at all:

```
func Guess(n int) {}
```

(Listing guess/2)

Let's use `go test` with the `-v` flag to show that the 21 test case now passes:

```
go test -v
```

```
=== RUN   FuzzGuess
=== RUN   FuzzGuess/xxx
--- PASS: FuzzGuess (0.00s)
    --- PASS: FuzzGuess/xxx (0.00s)
```

This is a neat feature, as it means that the fuzzer automatically creates *regression tests* for us. If we should ever write a bug in the future that would cause the function to fail given this same input, `go test` will catch it. Every time a new failing test case is found by the fuzzer, it becomes part of our regular test suite.

Fuzzing a risky function

Let's take a more complicated example. First, to set the scene, a word or two about strings, bytes, runes, and characters in Go.

As you may know, text characters in Go are called *runes*. In many languages characters are represented by single-byte values, and therefore characters and bytes are equivalent, but that's not the case in Go.

You probably also know that Go can represent not merely the 256 possible single-byte characters specified by ASCII, but also the 150,000-odd Unicode characters. This means that your Go programs can deal with the alphabet of just about every human language, and many other characters besides, such as emojis.

This does mean, though, that it can take more than one byte to represent a Go rune, since a single byte only gives 256 possible values. So it's not safe to write Go code that assumes each *byte* of a string represents a single *character*. Such code will produce the wrong result when given non-ASCII input, such as accented letters, or Chinese characters.

And we can demonstrate this quite easily. Suppose, for example, we want to write some function `FirstRune` that takes a string and returns its first rune. Given the string `"Hello"`, let's say, it should return the rune `'H'`.

In a programming language where runes are just bytes, this would be straightforward. We could simply treat the string as a `[]byte`, and return its first element.

We know that won't work in Go, unless we can guarantee that the input is restricted to ASCII text. But it's the kind of mistake that's easy to make, especially if you're new to Go.

Let's see if we can use fuzz testing to catch this kind of error, then. We'll use the fuzzer to generate random inputs, and see if any of them produce the wrong answer from `FirstRune`.

Here's an example of such a test:

```
func FuzzFirstRune(f *testing.F) {
    f.Add("Hello")
    f.Add("world")
    f.Fuzz(func(t *testing.T, s string) {
        got := runes.FirstRune(s)
        want, _ := utf8.DecodeRuneInString(s)
        if want == utf8.RuneError {
            t.Skip() // don't bother testing invalid runes
        }
        if want != got {
            t.Errorf("given %q (0x%[1]x): want '%c' (0x%[2]x)",
                s, want)
            t.Errorf("got '%c' (0x%[1]x)", got)
        }
```

```
    })
}
```

(Listing runes/1)

It might look a little alarming at first, but we'll break it down, line by line.

The general idea is to call `FirstRune` with some string (randomly generated by the fuzzer), and compare the result against what the "oracle" (in this case, the standard `utf8.DecodeRuneInString` function) says it should be.

If our `FirstRune` function works correctly, it should give the same rune result as `DecodeRuneInString` for all inputs. But if there's a bug in the function, its result won't match the oracle's, and the fuzz target will mark this as a failing test case.

Adding training data with `f.Add`

While, as we've seen, the fuzzer is quite capable of generating random values from scratch, it can be helpful to give it some starting inputs as hints. Remember the mysterious warning it gave us when we ran the fuzz test for the `Guess` function?

```
warning: starting with empty corpus
```

The fuzzer's *corpus* is the set of training data we give it, if you like. It can still fuzz even with an empty corpus, as we saw, but it helps if we can provide a few example values for it to start with. These will be supplied to our fuzz target and tested. The fuzzer will then generate more inputs from them by mutating them randomly.

Here's how we add training data to the corpus:

```
f.Add("Hello")
f.Add("world")
```

(Listing runes/1)

We can call `f.Add` to add more example values as many times as we like. The more training data we can give the fuzzer to suggest possible lines of attack, the better.

The corpus also includes any previously-generated failing test cases stored in the `testdata/fuzz` folder, like the "21" case in our previous example. So over time we'll build up a bigger and bigger corpus, which gives the fuzzer more to work with.

A more sophisticated fuzz target

Next comes the call to `f.Fuzz` with our fuzz target:

```
f.Fuzz(func(t *testing.T, s string) {
    got := runes.FirstRune(s)
    want, _ := utf8.DecodeRuneInString(s)
```

```
    if want == utf8.RuneError {
        t.Skip() // don't bother testing invalid runes
    }
    if want != got {
        ... // fail
    }
})
```

(Listing runes/1)

First, we'll call the function and save its result to a variable `got`, so we can compare it with the expected result. What *is* the expected result, though, since we don't know in advance what `s` is?

Well, we can't predict it, but we *can* ask the oracle. `DecodeRuneInString` will give us the right answer, and we'll store it in the `want` variable.

Now, we know that the fuzzer is going to generate lots of random strings for us to use as input, but are all possible strings actually *valid* input to `FirstRune`?

No, because not all strings encode valid UTF-8 runes. For example, the empty string doesn't. And there are many sequences of arbitrary bytes that don't happen to be valid UTF-8 data. So we need to do a little pre-filtering on our input, before checking whether `FirstRune` actually handled it correctly.

Suppose the fuzzer calls our fuzz target with some string that doesn't contain valid runes. What will our oracle function `DecodeRuneInString` return in this case?

The answer is the constant `utf8.RuneError`. If we get that result, then this particular input value is invalid, and we should just skip it. That's what the call to `t.Skip` does:

```
if want == utf8.RuneError {
    t.Skip() // don't bother testing invalid runes
}
```

(Listing runes/1)

A skipped test neither fails nor succeeds; the fuzzer just ignores it and moves on to the next randomly-generated input.

Why don't we skip the test *before* calling `FirstRune` with the input? Because even though we don't require `FirstRune` to give the right *answer* for invalid strings, we *do* require that it doesn't panic. So if any fuzz input makes `FirstRune` panic, we want to find that out before we skip.

Finally, just like in any test, we compare `want` and `got`, and fail the test if they're not equal. The failure message is a little more complicated than any we've written so far:

```
t.Errorf("given %q (0x%[1]x): want '%c' (0x%[2]x)", s, want)
t.Errorf("got '%c' (0x%[1]x)", got)
```

(Listing runes/1)

Let's break it down. In the event of a failure, what information would we like to print? Certainly the input string that caused the failure, so that's the first piece of data:

```
given %q
```

Even though this string is guaranteed to be valid UTF-8, we can't guarantee that all its characters will be *printable*, or have a corresponding glyph in the display font we're using.

So just in case the string contains some non-printing runes, we'd also like to print out its raw *bytes*, as numbers. How can we do that?

As you probably know, the `fmt` verb `%x` will print a piece of data as a hexadecimal value. And we can give the reader a clue that what follows is a hex value by prefixing it with "0x", which denotes a hex literal in Go.

So a format string like this would do the job:

```
t.Errorf("given %q (0x%x) ... ", s, s)
```

For example, if `s` were the string "hello", and the test failed, this would print:

```
given "hello" (0x68656c6c6f) ...
```

But notice that we had to supply the argument `s` *twice*, which seems wasteful. As you know, every `%` verb in the format string has to correspond to a piece of data, supplied as a subsequent argument to `t.Errorf`.

So if there are two verbs (`%q` and `%x`), then we need two data *arguments*, even though they're actually the same value. Is there a way to avoid repeating `s` in this case, then?

Yes. We can use what's called an *explicit argument index*, which is a number in square brackets that goes between the `%` and the `x` in `%x`:

```
%[1]x
```

This is interpreted by the formatting machinery as meaning "print the Nth data value". In this case, that'll be number 1, the first data value, which is `s`.

We can use the explicit-index trick to repeat a given argument any number of times in our format string:

```
fmt.Printf("%s %[1]s %[1]s %[1]s", "hello")
// Output:
// hello hello hello hello
```

In the fuzzing example, we need to use `s` twice, once to print it out as a quoted string (`%q`), and again as a hex value (`%x`). So here's how we do that:

```
t.Errorf("given %q (0x%[1]x) ...", s, ...)
```

So far, so good. What about the next part of the format string? Since both `want` and `got` in this case are characters, we'll need to use the verb `%c` to display them.

And, again, we'll use the explicit argument index (in this case, index 2) to re-use the same data, but this time printing it as a hex value (`%x`).

```
t.Errorf("... want '%c' (0x%[2]x)", ..., want)
```

Here's what that would look like if the test failed:

```
... want 'w' (0x77)
```

The final part of the failure message prints `got` in the same way:

```
t.Errorf("got '%c' (0x%[1]x)", got)
```

And the result:

```
got 'v' (0x76)
```

Using the fuzzer to detect a panic

So we're ready to write a null implementation of `FirstRune` and try out our test. We know it won't pass, and that's the point. As you'll recall, if we're trying to build a bug detector, we need to see it detect a bug.

Let's write one:

```
func FirstRune(s string) rune {
    return 0
}
```

(Listing runes/1)

This is definitely wrong, so let's see if the test agrees:

```
go test -fuzz .
```

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/2 completed
failure while testing seed corpus entry: FuzzFirstRune/seed#1
fuzz: elapsed: 0s, gathering baseline coverage: 0/2 completed
--- FAIL: FuzzFirstRune (0.02s)
    --- FAIL: FuzzFirstRune (0.00s)
        runes_test.go:22: given "world" (0x776f726c64): want 'w'
        (0x77)
        runes_test.go:24: got '' (0x0)
```

That's encouraging, so let's see what our carefully-constructed failure message actually reports:

```
given "world" (0x776f726c64): want 'w' (0x77)
got '' (0x0)
```

So the failing input is "world", which was one of the cases we manually added to the corpus using `f.Add`. What caused the failure, then?

The oracle thinks that the first rune of `"world"` is `w`, and I agree. But what `FirstRune` *actually* returned in this case was the rune value `0x0`, which corresponds to a non-printing character.

This is exactly what we expected to happen, since we deliberately wrote `FirstRune` so that it *always* returns zero, no matter what. But now that we've validated the bug detector, we can go ahead and write a slightly more plausible version of `FirstRune`:

```
func FirstRune(s string) rune {
    return rune(s[0])
}
```

(Listing runes/2)

We use the square bracket slice-index notation (`s[0]`) to extract the first byte of `s`, and we return it having converted it explicitly to the `rune` type.

Though plausible, this is still incorrect, and you can see why, can't you? This is the kind of code that someone might write, for example, if they didn't know about the distinction between bytes and runes in Go that we talked about earlier. It might even work some of the time, but only by accident, and so long as we only ever get input strings containing single-byte characters.

Let's see how far we get this time:

```
go test -fuzz .
```

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/2 completed
fuzz: elapsed: 0s, gathering baseline coverage: 2/2 completed,
    now fuzzing with 8 workers
fuzz: minimizing 27-byte failing input file
fuzz: elapsed: 0s, minimizing
--- FAIL: FuzzFirstRune (0.16s)
    --- FAIL: FuzzFirstRune (0.00s)
        testing.go:1356: panic: runtime error: index out of range
        [0] with length 0
        [stack trace omitted]
```

It seems that the two seed corpus examples have now been successfully checked (`2/2 completed`), and the fuzzer announces `now fuzzing with 8 workers`, meaning that all eight CPU cores are concurrently generating inputs for the fuzz target. The more cores we have available, the more fuzzing we can do in parallel.

When we're reporting a bug we found manually, it's always worth taking a little time to find the *minimal* input that still reproduces the bug. This will help the maintainers, because they can ignore the irrelevant parts of the input and focus on the part that triggers the problem. By this *shrinking* process, we eliminate any part of the input that doesn't affect the outcome.

The fuzzer does the same thing. If it finds some input that fails the test, it tries smaller and smaller versions of it, looking for the minimal failing input:

```
fuzz: minimizing 27-byte failing input file
fuzz: elapsed: 0s, minimizing
```

In this example, we can see that it found a 27-byte input string that caused the fuzz target to fail. It then tries smaller and smaller versions of the input to see if they still fail. This process continues until it has found the minimal input that causes the fuzz target to fail.

And here it is:

```
--- FAIL: FuzzFirstRune (0.16s)
    --- FAIL: FuzzFirstRune (0.00s)
        testing.go:1356: panic: runtime error: index out of range
        [0] with length 0
```

It seems that, in this case, the fuzzer managed to shrink the failing input to a completely empty string (`length 0`). This is rather puzzling. What's gone on here?

In fact, what's probably happened is that the fuzzer generated a random 27-byte string whose first rune was multi-byte, causing the function to return the wrong answer. As the fuzzer tried shorter and shorter versions of the string, the same "wrong result" failure kept happening.

Finally, when it reduced the string to length *zero*, it got a different failure: an "index out of range" panic. It so happens that `FirstRune` doesn't cope well with empty input strings:

```
func FirstRune(s string) rune {
    return rune(s[0]) // oops, slice overrun
}
```

(Listing runes/2)

As I'm sure you spotted right away, the slice index expression `s[0]` will panic when `s` is empty, and indeed that's what we're seeing. Let's fix that:

```
func FirstRune(s string) rune {
    if s == "" {
        return utf8.RuneError
    }
    return rune(s[0])
}
```

(Listing runes/3)

It seems reasonable to follow the example set by the standard library, which is to return `utf8.RuneError` in this case. Now the function won't panic when given an empty

input.

As you'll recall from our earlier example, when the fuzzer finds a failing input, it automatically saves it to the `testdata` folder. This becomes part of the corpus, and even when we run `go test` without the `-fuzz` flag, this case will still be checked.

Let's try that now, using the `-v` flag to `go test` to see which tests are actually being run, and whether our fuzz-generated cases are among them:

```
go test -v
```

```
=== RUN   FuzzFirstRune
=== RUN   FuzzFirstRune/seed#0
=== RUN   FuzzFirstRune/seed#1
=== RUN   FuzzFirstRune/xxx
    fuzz_test.go:61:
--- PASS: FuzzFirstRune (0.00s)
    --- PASS: FuzzFirstRune/seed#0 (0.00s)
    --- PASS: FuzzFirstRune/seed#1 (0.00s)
    --- SKIP: FuzzFirstRune/xxx (0.00s)
```

Again, `xxx` is actually a much longer name, but I've spared you that. We can see, though, that the two manually-added seeds are being checked (`seed#0` and `seed#1`), and then we come to the generated case `xxx`, which we know is the empty string.

The result of that is neither pass nor fail, but `SKIP`. Why? Remember when we first wrote the test, we said that we'll skip inputs for which `DecodeRuneInString` returns `RuneError`. That's true for the empty string, which is why this case shows as skipped.

Detecting more subtle bugs

The fuzzer has already proved its worth, by detecting a panic in the `FirstRune` function when it's called with the empty string as input. But are there still more bugs lurking?

In the previous example, we saw that the fuzzer found a 27-byte string that caused a test failure. That can't be due to the slice overrun bug, so there must still be another bug remaining in `FirstRune`.

Let's see if we can find it:

```
go test -fuzz .
```

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/3 completed
fuzz: elapsed: 0s, gathering baseline coverage: 3/3 completed, now
fuzzing with 8 workers
fuzz: minimizing 32-byte failing input file
fuzz: elapsed: 0s, minimizing
--- FAIL: FuzzFirstRune (0.13s)
    --- FAIL: FuzzFirstRune (0.00s)
```

```
fuzz_test.go:22: given "⍃" (0xcaad): want '⍃' (0x2ad),
got 'Ê' (0xca)
```

Again, interpreting this output, we can see that the fuzzer found a 32-byte failing input, which it then managed to shrink to just 2 bytes:

```
given "⍃" (0xcaad)
```

That is, this input string consists of the rune ⍃, which in UTF-8 is encoded as two bytes, `0xca` and `0xad`. If `FirstRune` had *correctly* decoded this string, it would have returned the rune ⍃, the same result as the oracle.

Instead, it mistakenly interpreted the first byte of the string (`0xca`) as a single-byte rune, in this case the letter Ê.

A victory for fuzzing! Even if we didn't know about the distinction between runes and bytes in Go, and even if we might not have thought of using a string like "⍃" as an example input, we were still able to find the bug.

So fuzz testing is very useful in situations where there is a wide range of possible inputs, and where the function may return correct results for some inputs, but not others.

This is usually true whenever the system accepts *user* input, for example. Users input all kinds of crazy things to our systems, even though they shouldn't do that. So any program that needs to parse or transform user input, or arbitrary data of any kind, is a good candidate for fuzz testing.

Even when we can't check the exact results of a function in a fuzz test, we can still catch panics and hangs (timeouts), and we may also be able to test some *properties* of the result, as in our earlier `Square` example.

What to do with fuzz-generated test cases

As we've seen, the aim of fuzz testing is to generate new failing test cases. So what should we do with the results?

We saw in the previous chapter that it's not a good idea to automatically update golden files; that is, to derive *passing* test cases from the current behaviour of the system. But it's absolutely a good idea to do the opposite: to automatically create test cases that cause the current system to *fail*.

The reason is simple: a failing test case blocks deployment and grabs your attention. A *false positive* bug detection—that is, a test failure that doesn't correspond to a real bug—is much less serious than a *false negative*, where there *is* a bug but you don't know about it.

Because fuzzing can create new test data, you'll need to remember to check these changes into your version control system. That's why it really only makes sense to run the fuzzer manually, for example on your development machine, rather than

automatically, in a CI pipeline or GitHub Action. The CI system would have to be set up to detect any changes and commit them to version control, and that's usually not straightforward.

Fuzzing can take a while, too, so we wouldn't want to do it every time we run the tests. A good time to try fuzzing is when you think you've covered the system pretty well with example-based testing, but you suspect there may still be bugs lurking. The more critical the reliability and correctness of the system, the more fuzz testing you'll want to do.

The examples we've tested in this chapter have all resulted in failing cases being found fairly quickly, but what if that doesn't happen? How long will the fuzzer keep trying, before deciding there are no failures to be found?

The answer is that it will keep trying forever! A test run in fuzz mode, if no failures are found, will never terminate. And that makes sense: it's unlikely that we could ever completely exhaust the space of possible inputs. There are always more bugs to be found.

> *It is up to you to decide how long to run fuzzing. It is very possible that an execution of fuzzing could run indefinitely if it doesn't find any errors.*
> —Katie Hockman, "Go Fuzzing"

You'll probably find that on your first few fuzz runs, there are plenty of failures to keep you occupied. Once you've found and fixed the bugs responsible, the fuzzer may run for a while without any results; perhaps for several minutes.

If you like, you can leave it to run for several hours, or even overnight. You never know, it *might* find something. Once you decide there's not much more to be gained by further fuzzing, though, or if your CPUs are starting to melt, you can terminate the test run by typing Ctrl-C in the terminal.

Alternatively, you can run the fuzzer for a fixed time period, using the `-fuzztime` flag. For example:

```
go test -fuzz . -fuzztime=10m
```

Fixing the implementation

How *should* we have implemented `FirstRune` correctly, then? Well, that's not really important for the discussion of fuzz testing, but I don't like to leave anyone in suspense. Let's see if we can fix this bug.

There are a variety of ways we could write `FirstRune` correctly, including the rather dull one of simply calling `utf8.DecodeRuneInString`, as the test does.

But here's a slightly more interesting version, just for fun:

```
func FirstRune(s string) rune {
    for _, r := range s {
        return r
```

```
    }
    return utf8.RuneError
}
```

(Listing runes/4)

Why does this work? As you probably know, the `range` operator iterates over a string by runes, *not* by bytes. So, on each successive execution of this loop, `r` will be the next rune in the string, starting with the first.

But on this very first iteration, we hit the `return` statement inside the loop body, and we return the first rune straight away. After all, that's the right answer!

In the case where `s` is empty, the `for` loop will execute zero times; that is to say, not at all. We'll go straight to the end of the function and return the value `0`.

I think this is correct, but let's see if the fuzzer agrees. Or, rather, we'll give the fuzzer a chance to find some counterexamples, and see if it succeeds:

```
go test -fuzz .
```

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/4 completed
fuzz: elapsed: 0s, gathering baseline coverage: 4/4 completed, now
fuzzing with 8 workers
fuzz: elapsed: 3s, execs: 130517 (43494/sec), new interesting: 5
(total: 13)
fuzz: elapsed: 6s, execs: 268452 (45962/sec), new interesting: 5
(total: 13)
fuzz: elapsed: 9s, execs: 396881 (42797/sec), new interesting: 5
(total: 13)
...
```

And, in fact, I left this to run for a minute or two, but it didn't find any more failing inputs. That doesn't guarantee that there aren't any, of course. Even the fuzzer can't guarantee that.

> *We can't prove anything in science. But the best that scientists can do is fail to disprove things while pointing to how hard they tried.*
> —Richard Dawkins, "The Greatest Show on Earth"

The aim of fuzz testing, then, is to fail to disprove the correctness of a function such as `FirstRune` for some reasonable period of time, while pointing to how hard we tried.

One interesting thing you might have noticed about this output is the word "interesting". What does that mean?

```
fuzz: elapsed: 3s, execs: 130517 (43494/sec), new interesting: 5
(total: 13)
```

It turns out that the fuzzer does more than just generate random inputs. It also looks at *which code paths are executed* by those inputs. It considers an input "interesting" if it causes a new code path to be executed in the system under test.

After all, if there's a code path that's executed by no other test case, bugs could definitely be lurking there. So the fuzzer tries to discover inputs that cause new code paths to be executed. If it finds any, it uses them as a basis to generate similar inputs, hoping to find one that triggers a failure.

A question that might occur to you is *how* the fuzzer knows which code paths in your system are executed by any given test case. The answer is that it uses the same runtime instrumentation with which Go calculates your *test coverage*, and we'll talk more about that in the next chapter.

7. Wandering mutants

> *Just because something is easy to measure doesn't mean it's important.*
> —Seth Godin, "Who vs. how many"

We saw in the previous chapter that Go's fuzz testing facilities are a great way to find new "interesting" test inputs. That is to say, test cases that cover hitherto-untested parts of our code.

But what does it mean to "cover" some piece of code, exactly? And how useful a measure *is* that, when thinking about test quality? In this chapter we'll explore what test coverage means, and how it works. We'll also try out a technique that can help us cover the system's behaviour in greater depth: *mutation testing*.

What is test coverage?

> *Test coverage is a term that describes how much of a package's code is exercised by running the package's tests. If executing the test suite causes 80% of the package's source statements to be run, we say that the test coverage is 80%.*

—Rob Pike, "The cover story"

We've emphasised in this book that the best way to think about how testing verifies your system is in terms of its *behaviours*, not its code. In other words, the goal of the tests is not merely to execute as many of your Go statements as possible, but to find out *what those statements do*.

However, it's quite common for one reason or another to end up with tests that don't actually cover all of your system code. Of the remaining statements that aren't executed by any test, we can ask: does it matter?

In other words, given some un-covered statement, is this statement important for the user-facing behaviour of the system? If so, we should probably add a test for it. But if not, we can remove it. Either way, that's a win for code quality.

We'll see later in this chapter that the test coverage in real programs almost never reaches 100%, and that's okay. Indeed, gaming the system to inflate this number artificially is a waste of time, and can lead to a false sense of security about the tests.

But while the absolute coverage number isn't a useful *target*, it can still be a valuable *signal*. If it's very low, it's likely telling you either that some important parts of your system aren't tested, or that you have some unnecessary code. Both are handy things to know, so let's see how to use the Go tools to check test coverage.

Coverage profiling with the `go` tool

To get a simple percentage figure for the current package, run:

```
go test -cover
```

```
PASS
coverage: 85.7% of statements
```

Not bad, but it looks like some statements aren't covered by any test. That might be okay, but we'd like to know *which* statements aren't covered, so we can check them.

To get that information, we need to generate a *coverage profile*:

```
go test -coverprofile=coverage.out
```

We see the same output as before, but now we can use the captured coverage data to generate an instrumented view of the source code:

```
go tool cover -html=coverage.out
```

This will open the default web browser with an HTML page showing the code highlighted in different colours. Covered code is shown in green, while uncovered code is shown in red. Some source lines just aren't relevant to test coverage, such as imports, function names, comments, and so on. These are coloured grey, and you can ignore them.

The interesting part of this output will be the code highlighted in red, indicating that it's not executed by any test.

Look at the red statements, and decide whether or not they *need* testing. Is the uncovered section so short and simple that you can *see* it's correct? If so, don't worry about testing it.

For example:

```
if err != nil {
    return err
}
```

We may well find a `return err` statement like this among the 14.3% of our code that's never executed by a test. And that's okay. We don't need to test Go's `return` statement: we know it works. There's no benefit in writing a test just to cover this one line. The closer we get to 100% coverage, the more rapidly the value of the additional tests diminishes.

Tests aren't free: they take time and energy to write, and every line of code we write is a kind of technical debt, tests included. We'll have to live with them, read them, run them, and maintain them, perhaps for years to come.

If we're going to invest in writing and owning a test, then, it needs to deliver enough value to repay that investment. The kind of tests needed to get us from, say, 85% coverage to 100% usually aren't worth it.

> *Testing is fundamentally a pragmatic endeavor, a trade-off between the cost of writing tests and the cost of failures that could have been prevented by tests.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

But in some cases of untested code, maybe there *is* some logic there that could be wrong. Almost anything *could* be wrong, indeed. You may want to add a test to exercise those specific lines, or to extend an existing test so that it triggers that code path.

Again, we're not blindly trying to raise the *number*, just for the sake of having a big number. Instead, we're using it to help us find areas of the system which may need improvement.

How is this number calculated, then? In other words, how does Go tell whether or not a particular line is executed when we run the tests?

The answer is that when running in coverage mode, the Go compiler inserts some extra instructions into the compiled program, before and after each statement in our Go source file, to record whether or not it's executed.

That's the data used to calculate the coverage profile, and also, as we saw in the previous chapter, to detect "interesting" code paths triggered by fuzz tests.

If you use an IDE such as Visual Studio Code (with the Go extension and its supporting tools) or GoLand, you can see test coverage directly in the editor, without having to run any terminal commands.

For example, in VS Code, use the *Go: Toggle Test Coverage in Current Package* command. This will run the tests and highlight the code to show what's covered and what's not. Once you've had a look around, the same command will turn off the highlighting again.

Coverage is a signal, not a target

There's a well-known saying to the effect that whatever you measure is what will be maximised. So you'd better be careful what you measure.

And when, in turn, a measure becomes a *target*, it ceases to be a good measure. That's Goodhart's Law. People will tend to *game* that measure, doing unproductive things simply to maximise the number they're being rewarded for.

You can understand why some organisations get a bit too excited about test coverage. It seems to make it possible to boil down the nebulous business of measuring code quality to a single number. And if we could do that, maybe we could make every program 100% good. How wonderful!

Some unwise engineering managers will require their teams to target a specific coverage percentage for exactly this reason. A bigger number is better, they reason, and so programmers should be forced to meet minimum coverage quotas.

But this is counterproductive, because not every change that increases test coverage is actually an improvement. For example, if you write a 100-line function that does basically nothing, and call it in a test, that will automatically increase your coverage percentage, but few people would agree that this actually improves the *system*.

It's the Cobra Effect: perverse incentives lead to unintended consequences. If you put a bounty on venomous snakes, some enterprising folks, rather than killing snakes, will start *breeding* them instead. And that's probably not what you wanted. Some well-intended "bug bounty" schemes for software have had similar results.

Test coverage, though, isn't even really a good measure of anything much. Let's see why that's the case.

Suppose, for example, that we have some function `Add` to compute the sum of two numbers. And suppose we write an obviously useless test for it, like this:

```
func TestAdd(t *testing.T) {
    t.Parallel()
    add.Add(2, 2)
}
```

(Listing add/1)

Now, this test *can't* fail, since it doesn't call `t.Error` or `t.Fatal`. And in any case it doesn't test anything *about* the `Add` function. It merely calls it.

So it's very easy to write an implementation of `Add` that's wronger than wrong, but still passes the test:

```
func Add(x, y int) int {
    return 42
}
```

(Listing add/1)

This useless test doesn't just pass. Worse, it also provides 100% coverage:

```
go test -cover
```

```
PASS
coverage: 100.0% of statements
```

If it weren't already clear, this shows the critical difference between test *coverage* and test *quality*. If we were focused solely on test coverage, we'd be happy with this test, and with the system.

All that said, though, it doesn't mean test coverage is worthless. It's worth *something*. The chief value of test coverage, I think, is as an occasional check on whether important parts of your system behaviour remain uncovered. And this can sometimes happen even when we're building the system guided by tests.

Also, coverage can give you new and useful information about how your code actually behaves, as opposed to how you *think* it behaves:

> *I like to do a test run with coverage on and then quickly glance at all the significant code chunks, looking at the green and red sidebars. Every time I do this I get surprises, usually in the form of some file where I thought my unit tests were clever but there are huge gaps in the coverage.*
> *This doesn't just make me want to improve the testing, it teaches me something I didn't know about how my code is reacting to inputs.*
> —Tim Bray, "Testing in the Twenties"

Finally, the *relative* changes in your test coverage over time can also be a valuable signal. By running a coverage check automatically, you can arrange to be warned when

some commit would reduce it, and perhaps even disallow such commits. As Tim Bray puts it in the same piece:

> *Here's a policy I've seen applied successfully: no check-in is allowed to make the coverage numbers go down.*

And this makes sense, doesn't it? If you're trying to check in a change that decreases your coverage percentage, then either you added some untested code, or you removed some tests you shouldn't have. Applying the "coverage ratchet" rule can help prevent mistakes like this.

And when the coverage on a project is already unacceptably low, the rule can at least prevent it from getting worse until you can spare the time or resources to pay down some of this technical debt.

So don't neglect test coverage, but don't obsess over it either. The key thing to keep in mind about tests in general is this:

> *Just because a statement is executed does not mean it is bug-free.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

In other words, it's possible to have a test that *executes* some piece of code, but still doesn't tell us anything useful about its *correctness*.

Assuming the test passes, that tells us at least that the code doesn't panic, which I suppose is better than nothing, but not by much. What we care about is, firstly, that it actually *does* something, and even more importantly, that it does the *right* thing.

How can we find that out?

Using "bebugging" to discover feeble tests

We saw in the previous section that, while test coverage can be useful for detecting sections of our system that we accidentally omitted to test, it's not a valuable goal in itself. This is because, while it shows us how *broadly* our code is covered by the tests, it doesn't tell us anything about how *deep* that coverage goes.

> *Most people–and most code coverage tools–think about unit tests in terms of breadth of coverage–how many lines of code do your tests touch? But depth of coverage is just as important.*
> *My definition of depth is: how **meaningfully** are you touching a line? As you might guess, that is hard to programmatically measure. But I know it when I see it.*
> —Michael Sorens, "Go Unit Tests: Tips from the Trenches"

In other words, what are we really testing here? In our `Add` example, the answer is nothing at all—or, at most, that `Add` doesn't panic. You wouldn't write a test like that, I know, but what if you find yourself in the position of evaluating someone else's codebase?

Perhaps you've been tasked with fixing a bug in an old system, for example, or even adding a new feature, and you're (wisely) not that confident in the tests. What could you do?

You could read through the tests carefully, asking yourself in each case "What are we really testing here?", and making sure that the test actually tests what it *claims* to test.

This kind of deep code reading is a very good idea in general, and I recommend it highly. This single activity will probably add more value to the tests, and to the codebase as a whole, than anything else you could be doing in your first few weeks on a new assignment.

But can we do even more? What other techniques might we use to find, for example, important behaviours that aren't covered by tests? While the test coverage tools can be useful, as we've seen in this chapter, they don't tell us everything. Importantly, they don't tell us where our tests are *feeble*.

As you'll recall from previous chapters, a feeble test is one that doesn't test as much as it should. It may well *cover* some piece of code, but still not *test* it adequately. For example, it might call a function, but test nothing useful about its behaviour.

One way to detect such tests might be to deliberately *add* bugs to the system, and see if any test detects them. If not, then that part of the system is, by definition, not adequately covered by tests.

> *Sometimes, you can gain quantitative estimates of how many problems might remain in software by seeding (or "bebugging"). Insert known bugs without telling the testers, then estimate the number of remaining unknown bugs by the percentage of known bugs they find.*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

And it's not hard to do this. It's hard to write code *without* bugs, if not impossible, but it's correspondingly easy to create them.

For example, suppose we see some line of system code like this:

```
if err != nil {
    return err
}
```

Let's enable "bebugging mode" and see what happens. Suppose we were to flip that `!=` operator to a == instead, for example:

```
if err == nil {
    return err
}
```

That should *definitely* break the program in some way, shouldn't it? If we run the tests now, will they detect the problem?

If this wanton act of bebugging doesn't cause some test to fail, then we have a missing or feeble test. We can now go ahead and improve the test so that it *does* catch this problem. (Don't forget to remove the problem, and also check that the test now passes.)

Detecting unnecessary or unreachable code

Bebugging is good at identifying feeble tests, but that's not the only reason it's useful. It's quite possible that we could insert some deliberate bug and find that no test fails, *and that's okay*.

It may be, in some cases, that even though the tests can't detect such a bug, it's not because they're feeble. Maybe they *do* capture everything important about the user-visible behaviour of the system. What can we deduce, then, from the fact that the bebugged code didn't make any of the tests fail?

We can deduce that the affected code simply *isn't necessary*. After all, it apparently makes no difference to the program's behaviour whether it works or not. In that case, we should be able to remove it altogether.

Suppose we have the following code in some part of our system, for one reason or another:

```
if false {
    return 1 // unreachable code
}
```

And suppose we change the `return 1` here to `return 0`, or just remove the line altogether. Will that change make some test fail? It seems unlikely, since this `return` statement can never be executed, either by a test or otherwise.

This isn't telling us that our tests are feeble. It's just telling us that this code isn't needed, which is true.

While we can see by inspection that code like this is unreachable, the same kind of bug can lurk in a much less obvious way. For example:

```
if s.TLSEnabled {
    if err := s.initiateTLS(); err != nil {
        return err
    }
}
```

Will changing `if err != nil` to `if err == nil` break some test? We don't know. It all depends on whether some test ever sets `s.TLSEnabled`.

If we bebug this code, and that doesn't cause any test to fail, we can infer that `s.TLSEnabled` is never true, and hence this code can never be reached—at least when using the system from tests.

Why aren't we testing with `s.TLSEnabled`, then? Isn't that an important user-facing behaviour? If it is, we should definitely test it. And we'll know when we've written a good, non-feeble test, because now flipping the `!=` to `==` will cause a failure.

On the other hand, maybe we're not testing `s.TLSEnabled` because users don't care about it, and don't use it in production. Even better! If that's the case, we can remove that setting and all its associated code.

If we can insert a bug that doesn't cause a test failure, it may be showing us where our tests are feeble, or it may be helping us to discover behaviours of the system that simply aren't needed. Either way, bebugging guides us to valuable improvements.

Automated bebugging is called *mutation testing*, and it's a very useful technique, though not many people are aware of it. It's definitely worth trying out at least once on any project. Let's see how it works.

Automated mutation testing

A mutation tester does automatically what we've just been doing manually: inserting bugs into the program. But it does this in a very systematic way. It first of all parses the program into a syntax tree, to identify assignment statements, conditional expressions, and so on.

It then *mutates* one of those code lines in a similar way to our manual bebugging. For example, it might change an `==` to a `!=`. And it then runs the tests. If any test fails, well, then the mutation tester concludes that this code is adequately covered, and moves on to the next.

On the other hand, if it bebugs a line and *no* test fails, then it has potentially found a real problem. It will report the program line and the change it made that didn't trigger a failure, and continue mutating the rest of the program.

The end result, then, is a set of *diffs*: changes that it's possible to make to the program which don't fail any test. It's up to you to interpret these results and decide what to do about them. As we've seen, sometimes the right response is to beef up a feeble test, or even add a new one. Other times we may able to remove the offending code, as apparently it's not that important.

There will also be plenty of *false positives*: cases where a mutation doesn't trigger a failure, but we're okay with that. For example, the mutation tester might change a line like this:

```
fmt.Println("Hello, world")
```

to this:

```
_, _ = fmt.Println("Hello, world")
```

It turns out `fmt.Println` actually returns something (who knew?) But we usually don't care about its result values, so we just don't assign them to anything. By adding the

blank identifier to ignore these values, the mutation tester has certainly changed the code, but it hasn't changed the user-visible *behaviour* of the code, so we can ignore it.

Finding a bug with mutation testing

Let's write a function and see if mutation testing can discover any interesting bugs or feeble tests. We'll keep it simple, for demonstration purposes.

Suppose we want to write some function `IsEven`, that (you guessed it) determines whether a given number is even.

We'll start with a test:

```
func TestIsEven_IsTrueForEvenNumbers(t *testing.T) {
    t.Parallel()
    for i := 0; i < 100; i += 2 {
        t.Run(strconv.Itoa(i), func(t *testing.T) {
            if !even.IsEven(i) {
                t.Error(false)
            }
        })
    }
}
```

(Listing even/1)

This test's name clearly states what it intends to test: that `IsEven` is `true` for even numbers. And not just one even number; we'll test dozens of them. In fact, we'll test the numbers 0, 2, 4, 6... in succession, all the way up to 98. It's hard (though not impossible) to imagine a bug in `IsEven` that would make it return the correct results for all these numbers, but not others.

Sidebar: notice that the subtest name we pass as the first argument to `t.Run` is just the generated value of the input `i`. There's nothing more we need to say about it: the value of `i` completely describes this test case.

Because you're an intelligent and attentive reader, though, you'll no doubt be jumping up and down right now and yelling "We need to test the *odd* number behaviour too!"

And you make a very good point. A malicious implementer could easily pass this test by just having `IsEven` always return `true`, no matter what the input.

So let's test the other, equally important behaviour of `IsEven`: that it's `false` for odd numbers.

```
func TestIsEven_IsFalseForOddNumbers(t *testing.T) {
    t.Parallel()
```

```
    for i := 1; i < 100; i += 2 {
        t.Run(strconv.Itoa(i), func(t *testing.T) {
            if even.IsEven(i) {
                t.Error(true)
            }
        })
    }
}
```

(Listing even/1)

Now, it shouldn't be too hard to write a simple implementation of `IsEven` that passes these tests. Let's try:

```
func IsEven(n int) bool {
    if n%2 == 0 {
        return true
    }
    return false
}
```

(Listing even/1)

Perhaps you think we should simplify this to just `return n%2 == 0`, and I agree with you. I really do. But that won't give the mutation tester quite enough to work with, so let's stick with this version for the moment.

We don't *think* there are any bugs lurking here, and the tests *look* adequate. But let's unleash mutation testing on it and see what happens.

Running `go-mutesting`

For this example we'll use the `go-mutesting` tool, which you can install by running this command:

```
go install github.com/avito-tech/go-mutesting/cmd/go-mutesting@latest
```

One very important caveat before we start:

Make a backup of your entire project, including the `.git` directory, before running `go-mutesting`.

It's not hard to see how a tool like this, whose *job* is to mess with your source code, could cause damage to your project files. So you'll want to either create a temporary copy to run the mutation tester on, or back up all your code, or both.

Having installed the `go-mutesting` tool, and made our backups, all we need to do to start testing is to run the command:

```
go-mutesting .
```

```
PASS "/var/folders/q3/xxx/T/go-mutesting-704821216/even.go.0" with
checksum xxx
The mutation score is 1.000000 (1 passed, 0 failed, 0 duplicated,
0 skipped, total is 1)
```

That didn't take long. And it looks like we've passed easily. We can see from the tool's output (`1 passed`) that it made just one mutation, and it "passed" (that is, the change caused some test to fail, so everything's fine).

Overall, we have a score of 1.0, which is a perfect score. In other words, the code survived all the mutations the tester was able to make to it, where "survived" means that any bug-inducing mutations were detected and stopped.

If you want to see what the survivable mutations *were*, just for fun, there's a switch for that. During the testing, each modified version of the source file is saved to a temporary folder, but these are normally deleted after the run is complete. To preserve them, use the `--do-not-remove-tmp-folder` switch to `go-mutesting`. You can then look at the various mutated versions of the function by viewing these temporary files.

In this case, the mutation it made was to remove the first `return` statement altogether, leaving us with this:

```
func IsEven(n int) bool {
    if n%2 == 0 {
    }
    return false
}
```

We feel this code shouldn't pass the tests, and indeed that's the case, so `go-mutesting` doesn't alert us about this mutation. It's apparently survivable, because our built-in bug detector catches it and prevents this change from going to "production".

On the other hand, if it had made some mutation that *didn't* cause a test failure, that would have lowered our score, and we'd have seen the problem reported.

Introducing a deliberate bug

Let's make our `IsEven` function a bit more complicated now, and see if we can introduce a bug that the mutation tester will be able to find.

Suppose we decide to speed up the function by *caching* each result after it's calculated. Once we know that a given number is even (or not), we can save that data in memory for future retrieval. Next time we're asked about the same number, we won't need to calculate the answer: we'll have it to hand.

This technique is called *memoization*, by the way: another good comp-sci word. Here's one attempt at adding memoization to `IsEven`:

```
var cache = map[int]bool{}

func IsEven(n int) (even bool) {
    even = n%2 == 0
    if _, ok := cache[n]; !ok {
        cache[n] = n%2 == 0
        even = n%2 == 0
    }
    return even
}
```

(Listing even/2)

First, we calculate whether n is even. Then we look up n in the cache to see if we already have that result stored. If so, we return it. If not, we store it in the cache, *then* return it.

Now, you may already have some private reservations about this code. Me too. But let's see if it even passes the tests, first of all.

```
go test
```

```
fatal error: concurrent map read and map write
```

Whoops. One does not simply read and write to a map from multiple goroutines, such as the ones created by `t.Parallel()`. That's a data race.

One way out of it would be to protect the map with a *mutex*. Let's try that:

```
var (
    m     = sync.Mutex{}
    cache = map[int]bool{}
)

func IsEven(n int) (even bool) {
    m.Lock()
    defer m.Unlock()
    even = n%2 == 0
    if _, ok := cache[n]; !ok {
        cache[n] = n%2 == 0
        even = n%2 == 0
    }
    return even
}
```

(Listing even/3)

Okay, this passes the tests, though I still don't recommend it as a shining example of Go code. Remember, we *need* some bugs in order to show off the awesome power of mutation testing, and I have a feeling we won't be disappointed.

Let's see what the mutation tester makes of it:

```
go-mutesting .
```

```
[details omitted]
...
The mutation score is 0.200000 (1 passed, 4 failed, 0 duplicated,
0 skipped, total is 5)
```

So it looks like we've found a few things worthy of our attention. Let's look at the details of this result and figure out what they can tell us.

Interpreting `go-mutesting` results

According to the `go-mutesting` output, it's detected a problem. In fact, it found *four* unsurvivable mutations.

Let's see what the first one was. Here's the relevant part of the output:

```
         defer m.Unlock()
         even = n%2 == 0
         if _, ok := cache[n]; !ok {
-                cache[n] = n%2 == 0
-                even = n%2 == 0
+                _, _, _, _, _ = cache, n, n, even, n
+
         }
         return even
 }
```

It can take a bit of practice to read these diffs, so let's break it down. The mutation here was to change these lines, inside the `if` block:

```
cache[n] = n%2 == 0
even = n%2 == 0
```

to just:

```
_, _, _, _, _ = cache, n, n, even, n
```

Weird flex, but OK. The change, however wild it looks, still results in a valid program. And the tests still pass, which is bad. This is exactly the kind of problem that

`go-mutesting` is supposed to find.

Why do the tests still pass, though? The effect of this change is that now we never save the computed value in the cache, so our hard-won memoization is actually disabled entirely. And apparently that's just fine, since none of the tests fail as a result.

The second failing mutation does more or less the same thing, and it's easy to see from this that although we're testing the results of `IsEven` correctly, we're *not* testing the *caching*. More useful information discovered by the mutation tester.

Revealing a subtle test feebleness

Let's look at the remaining diffs from this run of `go-mutesting` and see if there's anything else useful they can tell us about the state of the system.

Here's one of them:

```
 func IsEven(n int) (even bool) {
        m.Lock()
        defer m.Unlock()
-       even = n%2 == 0
+       _, _ = even, n
+
        if _, ok := cache[n]; !ok {
                cache[n] = n%2 == 0
                even = n%2 == 0
```

The change here comes before the `if` block, changing:

```
even = n%2 == 0
```

to:

```
_, _ = even, n
```

Since this change doesn't cause a test to fail, which is why we've been alerted to it, apparently this whole statement doesn't affect the tests in any way. Why not?

This takes a little thinking about, but suppose we call `IsEven` with some even number, say 10, and suppose that result *isn't* in the cache. In that case, we will compute the result, store it in the cache, and return it. Result, test pass.

And now suppose we call `IsEven` with some *odd* number, say 9. Exactly the same thing will happen: compute the result, store it, return it. Result, test pass. Either way, setting `even` before the cache check has no effect on the test status.

Can you see the problem now? If not, try going for a short walk (I always find this an invaluable thinking aid).

It *looks* as though the mutation tester is telling us we can safely remove this line. But can we? Well, suppose we call `IsEven` with some number that *is* in the cache. In that case, the `if` condition will be false, and we'll proceed straight to the `return even` statement at the end of the function.

Since the default value of `even`, like any `bool` variable, is `false`, that means `IsEven` will *always* return `false` for any cached number. Is that okay? Clearly not. So the problem must be in our *tests*.

Once you know that, it's not hard to see that no test ever calls `IsEven` with the same value *twice*. If it did, it would fail when this mutant code is present.

How disappointing! We went to some trouble to generate a wide range of possible test inputs, thinking that this would increase the chances of finding bugs. It does, but for all our efforts, there's one kind of input value we didn't think to supply: a *repeated* one.

So, while we can't easily test that the cache behaviour is correct, we've found at least one way that it could be *incorrect*, which is definitely a good thing. How could we improve our tests in the light of this discovery?

One way would be to have them check that `IsEven` returns the same result when called repeatedly with the same input, which it certainly doesn't at the moment. Incidentally, this would be a *property-based* test of the kind that we discussed in the previous chapter.

Here's the final failing diff that `go-mutesting` found:

```
        even = n%2 == 0
        if _, ok := cache[n]; !ok {
                cache[n] = n%2 == 0
-               even = n%2 == 0
+               _, _ = even, n
+
        }
        return even
 }
```

What is this saying? It's saying that if we remove the *other* `even = n%2 == 0` statement, inside the `if` block, then that *also* has no effect on the tests. Which makes sense: they do exactly the same thing. It seems like we should need one or the other of them, but not both.

In fact, it's the *first* one that we need, since removing it would cause incorrect behaviour for cached inputs. The second assignment just overwrites `even` with whatever its value already was. So we can get rid of it, making the program shorter and simpler, but without affecting its important behaviour. Thanks, mutation tester!

Fixing up the function

Here's a new version of the `IsEven` function, with some improvements to mitigate the various problems found by the mutation tester:

```
func IsEven(n int) (even bool) {
    m.Lock()
    defer m.Unlock()
    even = n%2 == 0
    if _, ok := cache[n]; !ok {
        cache[n] = n%2 == 0
    }
    return even
}
```

As perhaps you've noticed, though, even this isn't very good. What's the point of the cache code if, as it turns out, we always compute `n%2 == 0` anyway, whether it's cached or not?

Here's something marginally better, just for the sake of completeness:

```
func IsEven(n int) (even bool) {
    m.Lock()
    defer m.Unlock()
    even, ok := cache[n]
    if !ok {
        even = n%2 == 0
        cache[n] = even
    }
    return even
}
```

(Listing even/4)

As it turns out, in fact, no one really needs this program anyway. Go already has a built-in integer remainder operator, so instead of calling some `IsEven` function, we can just write, directly:

```
if n%2 == 0 {
```

And even if we did write such a function like `IsEven`, we certainly wouldn't bother trying to speed it up with memoization. Indeed, checking whether a number is even or odd only requires looking at its lowest-order bit, which should be a pretty fast operation on modern hardware.

Never mind, though. It's best to start off your exploration of mutation testing with some

pretty simple code, to get the hang of it, and that's what we've done. Once you feel confident that you understand what the mutation tester is telling you, and what to do about it, you're ready to run it on a real project.

Mutation testing is worth your while

Running a tool like `go-mutesting` over a large and complicated program can produce a *lot* of results, most of which aren't useful, and it can take some time and effort to go through them all carefully. But I can almost guarantee you that you'll also find one or two genuine bugs, feeble tests, or unnecessary code paths.

You might also like to try out a new mutation testing tool that's currently under development, `gremlins`:

- https://github.com/go-gremlins/gremlins

There's no need to run mutation testing every day, or even every week. Think of it more like a regular health check for your project, one that you'll maybe run every couple of months, or after making any major changes to the codebase.

It's also useful to record your score (that overall summary value between 0 and 1) each time. If you make changes to the project and your mutation testing score goes *down* as a result, then apparently you've introduced some unnecessary or feebly-tested code. You may want to review it and see if it can be removed or improved.

We've seen in this chapter what bebugging and mutation testing can do to help us improve existing tests, or to identify important behaviours that aren't tested. That's great, and it's a valuable addition to our testing toolbox.

But what about the tests that don't exist in the first place, because we simply can't figure out how to write them? Why do some things seem harder to test than others? What *makes* things hard to test? And how can we go about testing the apparently untestable?

Let's talk about this in the next chapter.

8. Testing the untestable

> *Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous.*
> —Saying

Most of the testing we've done so far in this book has been of the "compare want and got" kind. We call some function with prepared inputs, and compare its results to what we expected. This is straightforward, as most of these functions have been *pure*, meaning that their results depend only on their inputs, and they have no side effects.

It's not always that easy, though. Some functions and programs can be challenging to test, at least at first sight. In the next few chapters, we'll look at a few techniques for testing the so-called untestable, and we'll see that many apparent difficulties melt away if we can approach the problem in the right frame of mind.

First, how do we even *start*? When we begin a new project, we may not even have a clear idea of exactly what it should do and how it should behave. We need to do some prototyping, so let's talk about that process.

Building a "walking skeleton"

When starting a new project, to avoid getting bogged down in too much detail, it's a great idea to get something *running* as soon as you can. This needn't be feature-

complete, or anything like it. Indeed, it's better to focus on running the simplest possible program that does *anything* useful at all.

This has been aptly called a *walking skeleton*:

> *A walking skeleton is an implementation of the thinnest possible slice of real functionality that we can build, deploy, and test end-to-end.*
>
> *We keep the skeleton's application functionality so simple that it's obvious and uninteresting, leaving us free to concentrate on the infrastructure.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

Software engineering is, it turns out, more about the *connections* between components than the internals of the components themselves. It will probably take you longer to design and debug these connections than to write any individual component, so it's wise to start this process as early as possible.

Ideally, you'd just hook up a bunch of mostly dummy components in what looks like the right way, *run* it, and see what happens. You can iterate very fast on a walking skeleton, precisely because it has so little actual functionality.

Once we've got a walking—or rather, running—skeleton, our first priority is to move all its important behaviours into a package, covered by tests. This provides a solid foundation on which we can build each new increment of functionality, until we eventually have the complete system.

But we shouldn't abandon "runnability" at this point. Indeed, we should try never to get too far away from a runnable program. That means, naturally enough, that any changes we make must be small and incremental, which we already know is a good thing from a design point of view. So runnability helps keep our development process from going too far off the rails.

Once we've discarded our skeletal prototype, and started rebuilding it as an importable package, guided by tests, the next priority should be to deploy *this* to production as soon as possible.

> *There's one piece of advice that I find myself giving consistently during the very early days: get the simplest version you can think of into prod, as soon as you can. Focus solely on what's needed to get a "Hello, world" version of your new system up and running in a real production environment. No more, no less. I call this a Hello, Production release.*
> —Pete Hodgson, "Hello, Production"

Really? Production? Not a test environment?

Yup. If we want it to work in the *real* environment, then we'd better try it there. After all, we implicitly test in production all the time. We may as well accept that fact, and start doing it a little more rigorously. However much we might test the system in isolation, that only tells us how it behaves *in isolation*, which is of limited value.

> *Once you deploy, you aren't testing code anymore, you're testing **systems**—complex systems made up of users, code, environment, infrastructure, and a point in time. These systems have unpredictable interactions, lack any sane ordering, and develop emergent properties which perpetually and eternally defy your ability to deterministically test.*
>
> *The phrase "I don't always test, but when I do, I test in production" seems to insinuate that you can only do one or the other: test before production or test in production. But that's a false dichotomy. All responsible teams perform both kinds of tests.*
> —Charity Majors, "I test in prod"

The "hello, production" release will not only catch many show-stopping issues that could affect the design of the system, but it also means that, because we have the full pipeline working, we *could* ship at almost any time if we had to. It's hard to make predictions, especially, as the saying goes, about the future. There might not always be quite as much time to get things finished as we expect.

A very *bad* time to find major design issues, on the other hand, is when we're close to the deadline, worn out, frazzled, and under tremendous pressure to ship. If you've ever wondered why some software seems a bit half-finished, well, now you know.

The remedy is to get the system half-finished as soon as possible, and to maintain it always in a mostly-runnable state. So how do we go from a walking skeleton to a runnable, but half-finished system, guided by tests?

The first test

The first test of any system, or even any user story, is usually the hardest to write, and that's no surprise. Any non-trivial system has lots of behaviours. When we look at it from the outside, considering the user-facing behaviour, it can seem so complicated as to defy testing altogether.

The trick is not to solve that problem. Instead, pick some much smaller and simpler version of the problem, and solve *that*. Then iterate.

> *Make the problem easy, then solve the easy problem (warning: making the problem easy may be hard).*
> Kent Beck, "Is TDD Dead?"

For example, suppose you need to test some "buy product" HTTP endpoint that takes some form data supplied by a user, processes their payment details, and creates an order in the database. That's horrendous! No one could write a test for that kind of thing straight off. And if we're smart, we won't even try.

Instead, make the problem simpler. Cut out the payment handling, and don't worry about creating orders. Just have the handler ignore the form data altogether, and respond with a simple "hello, world" message. Is that useful? No. But could we test it?

Yes. And could we then incrementally build up the behaviour we need, guided by the test? Yes, indeed.

Solving big problems

Solving big problems is hard, so we don't do that. Instead, we eat the elephant one bite at a time. We say "if I had this component, and that component, then I could solve the problem by plugging them together". If there's a big complex behaviour, like "Be a CRM system", we break it down into lots of little behaviours, like "Store customer's name and address", or "Generate invoices".

Keep on breaking down behaviours into smaller units until you get to something that you could actually imagine implementing. This isn't just shuffling paper: by analysing complex behaviours into simpler ones, you're *defining* them more precisely, adding more and more detail to the overall design as you go.

> *The secret to building large apps is NEVER build large apps. Break up your applications into small pieces. Then assemble those testable, bite-sized pieces into your big application.*
> —Justin Meyer, "Organizing A jQuery Application"

Good software systems tend to be made up of lots of small behaviours, each of which has a well-defined responsibility and so is easy to understand and change. It stands to reason that if we build software by breaking up complex *behaviours* into small, well-specified units, the structure of our *code* will end up reflecting that modular organisation too. And that's something we'll appreciate as we come to maintain and extend it over time.

Designing with tests

Once you know what the behaviour of the system should be, the next challenge is to design a good *API* for it. In other words, how should users interact with this component? What should it be called, what information do they need to pass to it, and what should they expect in return?

It's hard, in general, to come up with design ideas from nothing, and the ideas we come up with usually don't work out perfectly in practice. Rather than try to design a function from scratch, in isolation, it's better to design its API by actually *using* it.

Indeed, a useful way to think about what we're doing is not so much writing a function as creating a *language* in which users can elegantly express their programs. The functions we design as part of our system are merely words in that language.

> *The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.*
> —Edsger W. Dijkstra, "The Humble Programmer"

This implies that we should think about our APIs from the user's point of view first, not the implementer's.

For example, suppose we're designing the constructor API for some `LoadTester` object. It has a lot of fields:

```
type LoadTester struct {
    URL                string
    Output, ErrOutput  io.Writer
    HTTPClient         *http.Client
    Fanout             int
    UserAgent          string
}
```

(Listing load/1)

How should we write `NewLoadTester`, then? We might start with something like the following:

```
func NewLoadTester(URL string, stdout, stderr io.Writer,
    httpClient *http.Client, fanout int, userAgent string)
    *LoadTester {
    ...
}
```

(Listing load/1)

But this is annoying to use. Look how much information we have to pass at the site of the function call:

```
got := load.NewLoadTester("https://example.com", os.Stdout,
    os.Stderr, http.DefaultClient, 20, "loadtest")
```

(Listing load/1)

Not only are there too many arguments, most of them are usually unnecessary, since they have sensible default values. For example, users will usually want the load tester to write to `os.Stdout` and `os.Stderr`, so there's no point making them specify the obvious values every time.

The ideal API would be one where if you *need* to specify some parameter, you can, but if you don't, you can omit it. A popular pattern for this is to use *functional options* (see The Power of Go: Tools for more about these).

With a functional-option API, if users don't need to customise anything, they can write a very concise function call:

```
got := load.NewLoadTester("https://example.com")
```

On the other hand, customisation is available when it's needed:

```
got := load.NewLoadTester("https://example.com",
    load.WithOutput(buf),
    load.WithErrOutput(buf),
    load.WithHTTPClient(&http.Client{Timeout: time.Second}),
    load.WithConcurrentRequests(20),
    load.WithUserAgent("loadtest"),
)
```

(Listing `load/2`)

Everyone's happy.

> *Simple things should be simple, complex things should be possible.*
> —Alan Kay, "The Wiki Way"

See Listing `load/2` if you're interested in how these functional options are implemented. It's not complicated, but it does require a bit of extra boilerplate. However, this extra effort is well worthwhile if it eliminates a large number of mandatory parameters from your API.

Unexported functions

Throughout this book, and others, I've emphasised that we should think about testing *user-visible behaviours*, not functions. That leads us naturally to so-called *external tests*: that is, we write the tests in a different package from the one we're testing.

> *Your test files should always be in a different package. Why? Because it forces you to test behavior rather than implementation.*
>
> *If you are in a different package, you only have access to the API of the package; you do not have access to internal state. So you are forced to treat the package as a black box, which is a good thing. That way, you are free to change internals without breaking tests as long as the API does not change.*
> —Michael Sorens, "Go Unit Tests: Tips from the Trenches"

This sometimes creates a puzzle when we want to test some *unexported* function. As you probably know, in Go a function whose name begins with a lowercase letter can only be called *internally*: that is, from within the same package.

How can we call it from a test in a different package, then?

We can't, but that's okay. By *making* the function unexported, we're saying it's not something users need to know about. Therefore, it doesn't need to be tested—at least, not directly.

Instead, we'll end up testing these functions *indirectly*, by calling the exported functions that use them. In other words, it's the user-facing behaviour that matters. Users don't care how that's actually implemented, and nor should our tests.

> *We've adopted a habit of putting tests in a different package from the code they're exercising. We want to make sure we're driving the code through its public interfaces, like any other client, rather than opening up a package-scoped back door for testing.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

Tests are a special case of users, but not *that* special: they should go through the "front door" like everybody else. After all, that's the door we really care about.

It does sometimes make sense to write a few internal tests. For example, we discovered in the previous chapter that it's difficult to test hidden behaviour such as caching when you test through the public API. The most straightforward way to check that the cache is being correctly updated is to write an internal test, but these should be kept separate from the others:

> *If you do need to test some internals, create another file with `_internal_test.go` as the suffix. Internal tests will necessarily be more brittle than your [external] tests, but they're a great way to ensure internal components are behaving.*
> —Mat Ryer, "5 simple tips and tricks for writing tests in Go"

Internal tests will necessarily be more brittle than external ones, because by definition they care about implementation details. If you change the implementation, even in a way that doesn't change the public behaviour, you're likely to break internal tests that rely on it—but that's a price we sometimes have to accept.

Concurrency

Another kind of function can be tricky to test, not because we can't *call* it, but because its execution is *concurrent*. For example, a function that uses a go statement to initiate a goroutine, possibly returning before the goroutine has completed.

How, then, can we test that the function has done its job, if that job may not actually complete for some unknown time? Writing concurrent programs is hard in general, and *testing* is one of the things that's hard about it.

The best way to solve the problem of testing a concurrent API is *not to write a concurrent API*. As we've seen before, a difficulty that surfaces in testing can point to an underlying design issue.

> *The tests are a canary in a coal mine revealing by their distress the presence of evil design vapors.*
> —Kent Beck, "Test-Driven Development by Example"

Users find concurrent APIs hard to understand and call correctly, too. The only way for an asynchronous function to communicate with its caller is via channels, and channels are awkward for users to consume. In essence, by providing a concurrent API, we force users to write concurrent programs in order to call it. This introduces a lot of complexity and many opportunities for obscure and difficult bugs.

> *It's much easier to test an API which is synchronous: that is, it doesn't export channels. There's a lot of cognitive overhead: who owns the channel, what is the life cycle of the channel, who closes it, is it buffered or unbuffered?*
> *If you write your test in a synchronous way it communicates the intent much better, and you don't have to worry about all of those additional semantics.*
> —Alan Braithwaite, "Advanced Testing Techniques"

Instead, a better plan is to *leave concurrency to the caller*. Dave Cheney advances the following two examples in his Practical Go:

```
// ListDirectory returns the contents of dir.
func ListDirectory(dir string) ([]string, error)

// ListDirectory returns a channel over which
// directory entries will be published. When the list
// of entries is exhausted, the channel will be closed.
func ListDirectory(dir string) chan string
```

The first version, which returns a `[]string`, is clearly synchronous. It doesn't return until the work is done, at which point it returns the complete set of results all at once. This might take a while, though, and consume arbitrary amounts of memory, and meanwhile the user's program must block.

The second version is equally clearly *asynchronous*, because it returns a channel. The work will proceed concurrently, with results being sent over this channel as they are generated. The responsibility is on the user to figure out how to consume this channel, and to know when the work is done.

What's the most likely way for them to do that? Surely something like this:

```
var entries []string
entriesChan := ListDirectory(dir)
for entry := range entriesChan {
    entries = append(entries, entry)
}
```

The `for ... range` loop continues to execute until the channel is closed, at which point we have built up a slice containing all the results in `entries`.

But this is precisely what the synchronous version of the function returns! The following code is much simpler from the *caller's* point of view:

```
entries := ListDirectory(dir)
```

Indeed, the concurrency is of no value after all, since the user's program ends up blocking until all the results are known. It's effectively synchronous anyway.

So the synchronous version wastes the user's time, while the asynchronous version wastes their effort instead. Is there a better way? Cheney suggests using a *callback*. That

is, passing a user-supplied function as an argument:

```
func ListDirectory(dir string, fn func(string))
```

In other words, the user calls `ListDirectory` and provides a function that will be called repeatedly—the callback—each time receiving the latest result as an argument. This is the pattern used by standard library functions such as `fs.WalkDir`, for example:

```
fs.WalkDir(fsys, ".", func(p string, d fs.DirEntry, err error)
    error {
    if filepath.Ext(p) == ".go" {
        count++
    }
    return nil
})
```

The function literal here receives each successive file path discovered by `fs.WalkDir` as a string, and can do whatever it needs to do with it. In this case, it increments a counter when the file path matches a certain extension. (See The Power of Go: Tools for more about `fs.WalkDir` and this example.)

It's now up to the user to decide whether this work should be concurrent. If so, they can call this function in a goroutine of their own, and get on with other work while it runs. The point is that concurrency isn't *required* as part of the API: it's optional.

But supposing that, as testers, we don't *have* that option, what should we do? For example, how should we test a pre-existing asynchronous API? Well, we'll need to write an asynchronous test, using the *wait for success* pattern:

> *An asynchronous test must wait for success and use timeouts to detect failure. This implies that every tested activity must affect the system so that its observable state becomes different. This sounds obvious but it drives how we think about writing asynchronous tests.*
> *If an activity has no observable effect, there is nothing the test can wait for, and therefore no way for the test to synchronize with the system it is testing.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

Let's take the common case of starting some kind of network server (for example, an HTTP server). Suppose its API is something like this:

```
// returns immediately
serve.ListenAsync(addr)
```

Let's first of all deal with an issue we'll encounter testing any kind of network server: which *address* to have it listen on. Sometimes the server *has* to listen on a specific port to be useful: for example, a DNS server must listen on port 53, because that's the DNS port.

For other servers the port number may not matter, but in order to test servers *in general* we'll need to be able to specify an arbitrary port. The reason is straightforward: each test needs to be able to start its own independent server, concurrently with other tests.

We could just assign arbitrary port numbers to each test, but that's fragile: we might accidentally end up with two tests using the same port, or worse still, one of the ports we assigned might already be in use by some other program.

Let's solve both problems by constructing a test helper that finds a *random* port number that happens to be free. How can we do that? There's a trick.

If we start a network listener on port *zero*, that has the effect of asking the kernel to assign us some random free port. We can then ask the resulting listener what its address *really* is. That's the address we'll use in the test:

```
func randomLocalAddr(t *testing.T) string {
    t.Helper()
    l, err := net.Listen("tcp", "localhost:0")
    if err != nil {
        t.Fatal(err)
    }
    defer l.Close()
    return l.Addr().String()
}
```

(Listing serve/1)

The actual address returned will probably be something like:

```
127.0.0.1:60606
```

So now we know how to get a unique, available network address for each test, we can pass this to `ListenAsync` and proceed to test the server by trying to connect to it.

We might start with something like this:

```
func TestListenAsyncEagerly(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    serve.ListenAsync(addr)
    _, err := net.Dial("tcp", addr)
    if err != nil {
        t.Fatal(err)
    }
}
```

(Listing serve/1)

Seems promising, but unfortunately it doesn't work:

```
serve_test.go:16: dial tcp 127.0.0.1:60733: connect: connection
refused
```

And a little thought will make it clear why not. Presumably, since `ListenAsync` returns immediately, it uses a goroutine to start the server concurrently.

But our test, having called `ListenAsync`, then *instantly* tries to make the connection with `net.Dial`. This is unlikely to succeed, because the server goroutine won't have had a chance to run yet.

We're being too eager. Concurrency is not parallelism, as Rob Pike has observed, and while starting a goroutine is usually fast, it's not *instant*. Setting up the server and listener itself may also take a while.

We need to wait a little. Should we add, for example, some fixed delay to the test? Would, say, ten milliseconds be enough?

```
func TestListenAsyncWithSleep(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    serve.ListenAsync(addr)
    time.Sleep(10 * time.Millisecond)
    _, err := net.Dial("tcp", addr)
    if err != nil {
        t.Fatal(err)
    }
}
```

(Listing serve/2)

Well, maybe. But also maybe not. The test is now *flaky*. If we run it ten times in a row, it will sometimes fail, sometimes pass:

```
go test -count 10 -run ListenAsyncWithSleep
```

```
--- FAIL: TestListenAsyncWithSleep (0.02s)
    serve_test.go:18: dial tcp :61707: connect: connection refused
--- FAIL: TestListenAsyncWithSleep (0.01s)
    serve_test.go:18: dial tcp :61710: connect: connection refused
```

The test's goroutine is *racing* with the server goroutine: sometimes the server starts first, sometimes it doesn't. The shorter the fixed sleep, the more likely the test is to flake, or "flicker":

> *A test can fail intermittently if its timeout is too close to the time the tested behavior normally takes to run. An occasional flickering test might not cause problems, but as the test suite grows, it becomes increasingly difficult to get a test run in which none of the flickering tests fail.*

> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

But making the fixed sleep longer also makes the test suite slower overall. There's no right length for this interval. If it's long enough that the test passes 95% of the time, then we'll waste a lot of time overall. Are we compelled to choose between flaky tests or slow tests, or is there a better option? Maybe there is.

Just *waiting* isn't good enough by itself, it turns out. As Freeman & Pryce observe, the trick is to wait *for success*. In other words, the test needs to try the connection, understanding that it may fail, and keep retrying until either it succeeds, or the test times out:

```
func TestListenAsyncPatiently(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    serve.ListenAsync(addr)
    _, err := net.Dial("tcp", addr)
    for err != nil {
        t.Log("retrying")
        time.Sleep(time.Millisecond)
        _, err = net.Dial("tcp", addr)
    }
}
```

(Listing serve/3)

Note that we still call `time.Sleep`, but now it's for a much shorter time, because we call it inside a loop. It's possible, though unlikely, that the very first `net.Dial` will succeed, in which case we never enter the loop at all.

If it fails, though, we log a message to say that we're retrying, sleep a little, and call `net.Dial` again. Let's see what happens when we run this test with logging enabled:

```
go test -run ListenAsyncPatiently -v
```

```
=== RUN   TestListenAsyncPatiently
=== PAUSE TestListenAsyncPatiently
=== CONT  TestListenAsyncPatiently
    server_test.go:17: retrying
    server_test.go:17: retrying
    server_test.go:17: retrying
    server_test.go:17: retrying
    server_test.go:17: retrying
--- PASS: TestListenAsyncPatiently (0.02s)
```

It looks as though the server is taking around five retries to become ready, amounting to about five milliseconds in total. The exact length of time isn't the point here, though:

the point is that the test waits as long as it takes the server to start, but never much longer than that.

One question may already have occurred to you, though: what if the server never starts up at all? Will the test simply wait forever?

No, because all tests time out after a certain period anyway (by default, 10 minutes, but we can control this with the `-timeout` flag to `go test`). However, we may not want to wait this long to find out that the server isn't starting.

Instead, we can add some timeout code to the test itself, using the ever-useful `time.Timer`:

```
func TestListenAsyncWithTimeout(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    serve.ListenAsync(addr)
    timeout := time.NewTimer(100 * time.Millisecond)
    _, err := net.Dial("tcp", addr)
    for err != nil {
        select {
        case <-timeout.C:
            t.Fatal("timed out")
        default:
            t.Log("retrying: ", err)
            time.Sleep(time.Millisecond)
            _, err = net.Dial("tcp", addr)
        }
    }
}
```

(Listing serve/4)

As before, if the first connection attempt fails, we enter the retry loop. But now we have a `select` statement that attempts to receive from the timer's channel (`timeout.C`). If this receive succeeds, we have timed out, so we fail the test.

On the other hand, if there is no message for us on the timeout channel, meaning that there's still some time left, we sleep and retry the connection as before.

One question that might occur to you is why don't we just start our own goroutine in the test that sleeps for a fixed duration and then calls `t.Fatal`? Something like this, for example:

```
go func() {
    time.Sleep(100 * time.Millisecond)
```

```
    t.Fatal("timeout") // has no effect
}()
```

Alas, that's no good. `t.Fatal` only works when it's called from the test's own goroutine, which is also why we can't use `time.AfterFunc`, for example. A `select` statement does the trick, though.

In practice, since we're likely to need this "wait for server" code in several tests, let's outsource it to another helper function:

```
func waitForServer(t *testing.T, addr string) {
    t.Helper()
    timeout := time.NewTimer(100 * time.Millisecond)
    _, err := net.Dial("tcp", addr)
    for err != nil {
        select {
        case <-timeout.C:
            t.Fatal("timed out")
        default:
            t.Log("retrying: ", err)
            time.Sleep(time.Millisecond)
            _, err = net.Dial("tcp", addr)
        }
    }
}
```

(Listing serve/5)

The `waitForServer` helper cleans up the actual tests considerably, leaving us free to express what we're really testing here:

```
func TestListenAsyncWithHelper(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    serve.ListenAsync(addr)
    waitForServer(t, addr)
    // test whatever we're really testing
}
```

(Listing serve/5)

Indeed, we could move *all* this setup code into a synchronous helper function (`startTestServer`) that starts the server, waits for it to be ready, and then returns. But you get the point: when testing asynchronous APIs, *wait for success* to avoid flakiness.

Concurrency safety

As we've seen, it's wise to avoid exposing concurrency in your API unless its *purpose* is to be concurrent. Synchronous APIs are easier to understand, easier to use, and easier to test.

But even with an API that doesn't use concurrency itself, it's also wise to assume that users will *call* it concurrently: this is Go, after all. That means that we *always* need to be concerned about avoiding data races, even when we're not using concurrency ourselves.

It's common to see Go packages written to use some kind of global mutable state, like this:

```
package kvstore

var data = map[string]string{} // no! data race!

func Set(k, v string) {
    data[k] = v
}

func Get(k string) string {
    return data[k]
}
```

(Listing kvstore/1)

How could there be a data race here? It doesn't look like the package even uses concurrency. Well, no, it doesn't. But what if the *user's* program, or a test, does?

For example, consider a test like this:

```
func TestSmokeKVStore(t *testing.T) {
    t.Parallel()
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        for i := 0; i < 1000; i++ {
            kvstore.Set("foo", strconv.Itoa(i))
        }
        wg.Done()
    }()
    for i := 0; i < 1000; i++ {
```

```
            _ = kvstore.Get("foo")
            runtime.Gosched()
        }
        wg.Wait()
}
```

(Listing kvstore/1)

We start one goroutine doing a thousand `Set`s on the map, and in the test's own goroutine we concurrently do a thousand `Get`s. This should be enough to trigger a crash if (as we suspect) the `kvstore` package is not concurrency safe.

And indeed, that's what we see:

```
go test
```

```
fatal error: concurrent map read and map write
```

There's a data race, because the `data` map is shared between multiple goroutines, at least one of those goroutines writes to it, and there is no synchronisation to ensure that accesses to the variable happen in a deterministic order.

The author of `kvstore` evidently wasn't thinking about concurrency, because they weren't *using* concurrency. But in Go, we always need to think about concurrency! At least, since any function *can* be called concurrently, we need to assume that it *will* be, and prepare accordingly.

One way to check concurrency safety is to call the package from a *smoke test*, such as the one in our example, that repeatedly calls the function from multiple goroutines, hoping to trigger a data race. And in this case we succeeded.

But this is a matter of luck: a fatal error like this may not actually happen. The order in which goroutines execute is unpredictable: that's why concurrency safety is an issue in the first place. We don't want flaky tests, and we can't rely on luck to detect a bug like this. But we have a secret weapon: Go's *race detector*.

By running the race detector on this test, using the `-race` flag, we get the benefit of its ability to detect even many *potential* data races:

```
go test -race
```

```
==================
WARNING: DATA RACE
Write at 0x00c000016360 by goroutine 8:
  runtime.mapassign_faststr()
      /usr/local/go/src/runtime/map_faststr.go:203 +0x0
  kvstore.Set()
      .../kvstore/1/kvstore.go:6 +0x91
  kvstore_test.TestSmokeKVStore.func1()
      .../kvstore/1/kvstore_test.go:18 +0x66
```

```
Previous read at 0x00c000016360 by goroutine 7:
  runtime.mapaccess1_faststr()
      /usr/local/go/src/runtime/map_faststr.go:13 +0x0
  kvstore.Get()
      .../kvstore/1/kvstore.go:10 +0x105
  kvstore_test.TestSmokeKVStore()
      .../kvstore/1/kvstore_test.go:23 +0xda
  ...
==================
--- FAIL: TestSmokeKVStore (0.00s)
    testing.go:1319: race detected during execution of test
```

This is telling us, formally, what we already knew, informally: calling `Set` and `Get` concurrently is not safe. The problem is with the basic design of `kvstore`: it has global mutable state, so it can't safely be called concurrently.

Let's see how to fix this, by using a mutex to synchronise access to the map, and eliminating the global `data` variable. We'll create a new `Store` type to hold the data and the mutex that protects it:

```
type Store struct {
    m    *sync.Mutex
    data map[string]string
}

func NewStore() *Store {
    return &Store{
        m:    new(sync.Mutex),
        data: map[string]string{},
    }
}

func (s *Store) Set(k, v string) {
    s.m.Lock()
    defer s.m.Unlock()
    s.data[k] = v
}

func (s *Store) Get(k string) string {
    s.m.Lock()
    defer s.m.Unlock()
    return s.data[k]
}
```

(Listing kvstore/2)

Now we can have *multiple* stores instead of just one, which is useful, and each of those stores is in turn concurrency safe, because its `Set` and `Get` methods use the mutex to (briefly) get exclusive access to the map.

We haven't created too much extra paperwork for users. The API now looks like this:

```
s := kvstore.NewStore()
s.Set("a", "b")
...
```

Let's try our smoke test, with the race detector enabled:

```
go test -race
```

```
PASS
```

While the race detector is a wonderful tool, it can't detect every *possible* data race in the program. It can only inspect the code paths that happen to be executed by a given test. A smoke test can help, but we can't guarantee to catch everything.

This is a limitation of testing in general. We can't just bang out some unsafe code and assume the tests will catch any problems. Tests are not a substitute for thinking:

> *We say, "I can make a change because I have tests." Who does that? Who drives their car around banging into the guard rails?*
> —Rich Hickey, "Simple Made Easy"

And however careful our smoke testing, we may not be able to put enough load on the system to actually trigger a race. Indeed, data races are most likely to cause crashes in *production*, at times of peak demand. In other words, exactly when you don't want them.

Long-running tasks

One common application for concurrency in Go is to enable *long-running tasks* to proceed independently. For example, any function that calls an external network service, such as a remote API, could in principle take a long time. Or we might need to execute some lengthy batch job, such as a backup, in a goroutine so that we can get other work done while we're waiting.

In cases like these, as with any other concurrent job, we may need a way to *cancel* the job, either because some other event has rendered it unnecessary, or perhaps because we simply got tired of waiting.

The `context` package is the standard way to cancel or time out concurrent tasks in Go, and it works like this. The caller creates a `context.Context` object and passes it to the potentially long-running function. When the caller gets bored, it can *cancel* the context, signalling the job that it should terminate.

We saw earlier in this chapter how to time out slow operations in a test, using `time.Timer`, for example. When testing functions that take a context, though, we have a built-in timeout facility, in the `context` package itself.

Suppose we need to test some function `RunBatchJob`, for example. Under test conditions, it should return fairly quickly, but it *might* take a long time, or even hang. We want to detect that without slowing down our tests too much.

We could write something like this:

```
func TestRunBatchJob(t *testing.T) {
    t.Parallel()
    ctx, cancel := context.WithTimeout(context.Background(),
        10*time.Millisecond)
    defer cancel()
    go func() {
        batch.RunBatchJob(ctx)
        cancel()
    }()
    <-ctx.Done()
    if errors.Is(ctx.Err(), context.DeadlineExceeded) {
        t.Fatal("timed out")
    }
}
```

(Listing batch/1)

First, we create a context with a pre-set timeout of 10ms. We defer calling its `cancel` function, to make sure that however the test exits, the operation will be cancelled, to avoid wasting time or resources. Next, we start the operation itself, asynchronously, by calling `RunBatchJob` in a little goroutine and passing it the context.

Now, we almost certainly need to wait a *bit*, so we block on receiving from the context's `Done` channel, to give the job a chance to run.

As you probably know, a blocked channel receive statement is unblocked when the channel is closed, and a so-called *close-only channel* is a common idiom in Go programs. No messages will actually be sent on such a channel: the closing of the channel *is* the message.

So the test will wait for something to happen, and only proceed once the context's `Done` channel is closed. This will be for one of two reasons: either the job completed normally, or it ran too long and the context timed out.

Let's see how that works. Suppose `RunBatchJob` is behaving well, and returns within the specified time. In this case, the little goroutine calls `cancel()` itself and exits. This in turn unblocks the receive from `ctx.Done()` in the test's main goroutine, allowing

the test to proceed.

At this point, we know the context is done, one way or the other: either the job has completed normally, or the timeout has fired. In order to find out which it was, we now check the value of `ctx.Err()`.

In the happy path, the job completed on time, and so the context was cancelled by us. In this case, the value returned by `ctx.Err()` will be `context.Cancelled`. This is as expected, so that's the end of the test: pass.

On the other hand, what happens if `RunBatchJob` takes too long? Now the 10ms timer will fire, and the `Done` channel will be closed, allowing the test to proceed. But when we check the reason (by looking at `ctx.Err()`), we will find that it is `context.DeadlineExceeded`, meaning that the context timed out. That's a problem, so we fail the test with `t.Fatal`.

The exact details of this example don't matter, but the main idea is that, when testing a potentially long-running operation that takes a `context.Context`, we can pass it a context with a timeout. We start the operation asynchronously, and wait on the context's `Done` channel. When it's done, we check to see if the context timed out.

User interaction

Some simple-looking functions can be quite tricky to test in Go, because they interact directly with the user's terminal. Here's an example:

```
func Greet() {
    fmt.Print("Your name? ")
    var name string
    fmt.Scanln(&name)
    fmt.Printf("Hello, %s!\n", name)
}
```

(Listing greet/1)

The resulting interaction might go something like this:

```
Your name? John
Hello, John!
```

Very friendly. Now let's try to write a test for the `Greet` function, starting with *calling* it:

```
func TestGreet(t *testing.T) {
    t.Parallel()
    greet.Greet()
}
```

(Listing greet/1)

We run into a problem straight away, though: how and when should this test *fail*?

`Greet` doesn't return anything, so there's no result value to check. Let's run the test anyway and see what happens:

```
go test
```

```
Your name? Hello, !
PASS
ok      greet   0.193s
```

So we have a useless test, that tests nothing and can't fail, and it also prints junk to the terminal, cluttering up the test results. Not good. Should we just give up, and reject this function as untestable?

Maybe. But maybe there's something we *can* do. The first question to ask, as usual, is "What are we really testing here?"

The behaviour we want is something like this: `Greet` prints a prompt, reads a string as input, then prints a greeting incorporating that string.

When you put it that way, it becomes clear that `Greet` interacts with the user in two separate ways: "printing", that is, writing to some `io.Writer`, and "reading", that is, reading from some `io.Reader`. This insight is the key to writing a useful test.

Let's start with the first problem: how to test that `Greet` prints a prompt. We saw in the earlier chapter on test data that a `*bytes.Buffer` implements the `io.Writer` interface, so it's something we can print to. And because we can then interrogate the contents of the buffer, we can find out exactly *what* was printed.

This sounds like a potential way forward, but there's still a problem. Here's the code in `Greet` that prints the prompt message:

```
func Greet() {
    fmt.Print("Your name? ")
    // ...
}
```

This code calls `fmt.Print`, which doesn't *use* some `io.Writer` variable; it just prints to the terminal.

But how does `fmt.Print` know that it should print to the terminal anyway? Is that just hard-wired somehow? Let's take a look at the source code of the `fmt` package and see:

```
func Print(a ...any) (n int, err error) {
    return Fprint(os.Stdout, a...)
}
```

It turns out that `fmt.Print` is just a thin wrapper around another function `fmt.Fprint`, supplying `os.Stdout` as its first argument, along with whatever

values the user passed to `Print`. So the more general function is `Fprint`, which can print to any writer.

So how can we make the `Greet` function print somewhere else, given that it uses `Print` rather than the more flexible `Fprint`? For example, how could we make it print to a buffer created in a test?

Our first idea might be to assign that buffer to the `os.Stdout` variable, but this is a bad idea. Firstly, if this happens in parallel tests we would have a data race. Secondly, we don't have exclusive use of `os.Stdout`: Go needs it too, for example when printing the test results.

A function that's hard-wired to print to `os.Stdout` is hard to test in general. So we don't test that function. We test *this* function instead, which takes an `io.Writer`, and prints to it:

```
func Greet(out io.Writer) {
    fmt.Fprint(out, "Your name? ")
    var name string
    fmt.Scanln(&name)
    fmt.Fprintf(out, "Hello, %s!\n", name)
}
```

(Listing greet/2)

We've pulled off another magician's switch, replacing a tricky problem (how to safely replace `os.Stdout`) with an easy one: how to create an `io.Writer` in the test, and pass it to the `Greet` function.

This is straightforward to solve:

```
func TestGreet(t *testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    greet.Greet(buf)
    want := "Your name? Hello, !\n"
    got := buf.String()
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing greet/2)

And, of course, if we'd written this test *first*, this is exactly the kind of idea we would have come up with. Knowing that we needed to pass an `io.Writer` to the function,

we would have designed it to take one as a parameter. We simply can't *write* untestable functions when the test comes first.

Fine, but what if this is a legacy system? What if the `Greet` function has already been written, without being guided by tests, and now the job of adding a test after the fact has landed in our unwilling lap? Is there anything we can do?

Indeed there is: we can refactor the function to accept an `io.Writer`, as we've done here. Now the test becomes easy to write, so we write it. Legacy systems that weren't designed to be testable can usually be refactored in a fairly simple way to *make* them testable, as in this example.

However, this test is still too feeble. It tests that `Greet` prints something, successfully, but it doesn't test that it then *reads* anything.

Let's turn to the second problem, then: reading "user input". How does `fmt.Scanln` know where to read input from, anyway? Could it be simply a convenience wrapper around some other function, as we found previously with `fmt.Print`?

Let's take a look:

```
func Scanln(a ...any) (n int, err error) {
    return Fscanln(os.Stdin, a...)
}
```

Just as before, we've found a more general API (`Fscanln`) concealed behind a more convenient one (`Scanln`). And, just as before, we can refactor our function to call `fmt.Fscanln` directly, passing it some `io.Reader` to read from:

```
func Greet(in io.Reader, out io.Writer) {
    fmt.Fprint(out, "Your name? ")
    var name string
    fmt.Fscanln(in, &name)
    fmt.Fprintf(out, "Hello, %s!\n", name)
}
```

(Listing greet/3)

Now all we need in the test is something that satisfies `io.Reader`, and that we can preload with some string representing what the "user" types at the prompt.

Here's a simple way to construct just such a reader, using `strings.NewReader`:

```
func TestGreet(t *testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    input := strings.NewReader("fakename\n")
    greet.Greet(input, buf)
    want := "Your name? Hello, fakename!\n"
```

```
    got := buf.String()
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing greet/3)

By supplying the input and the output streams as parameters to the function, we make it far easier to test. But what about the real program? When we call `Greet` from `main`, what values should we pass it for `in` and `out` so that it will read and write to the user's terminal?

Well, we already know the answer to that:

```
func main() {
    greet.Greet(os.Stdin, os.Stdout)
}
```

So now that we've succeeded in crafting a test for `Greet`, we can ask: is it a *good* test? Are there any bugs in `Greet` that a more demanding test (or simply a more imaginative choice of test data) might uncover?

My friend Mary Jo suggested this modification to the test, for example:

```
func TestGreet(t *testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    input := strings.NewReader("Mary Jo\n")
    greet.Greet(input, buf)
    want := "Your name? Hello, Mary Jo!\n"
    got := buf.String()
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing greet/4)

Let's see if her choice of test input behaves differently to mine:

```
--- FAIL: TestGreet (0.00s)
    -   "Your name? Hello, Mary Jo!\n",
    +   "Your name? Hello, Mary!\n",
```

What's going on? Have we accidentally written a sexist function that doesn't listen to women?

Or is there a more subtle, if no less serious, problem? Actually, the clue lies in a closer reading of the documentation for `fmt.Fscan`:

```
// Fscan scans text read from r, storing successive
// space-separated values into successive arguments...
```

The key phrase here is "space-separated". In other words, `Fscan` treats spaces in the input as *delimiters*. When we ask it to read a string value, as in this example, it stops when it sees the first space, and returns everything it's read up to that point. The remainder of the input is simply lost. That's bad news for Mary Jo, and everybody else who happens to have a name containing spaces.

By being a little more imaginative in our choice of test inputs, or perhaps by having a more diverse team of contributors, we successfully uncovered an important bug. And this is the kind of situation where fuzz tests, which we discussed in a previous chapter, could also be very helpful in generating new test cases.

So, just out of interest, what should we have used instead of `fmt.Fscan`?

Whenever we're interested in reading *lines* of input, as opposed to bytes, for instance, then a `bufio.Scanner` is a good choice:

```
func Greet(in io.Reader, out io.Writer) {
    fmt.Fprint(out, "Your name? ")
    scanner := bufio.NewScanner(in)
    if !scanner.Scan() {
        return
    }
    fmt.Fprintf(out, "Hello, %s!\n", scanner.Text())
}
```

(Listing greet/5)

`bufio.NewScanner` takes an input stream—that is, some `io.Reader`—and returns a scanner object that will read from the stream. We can then call its `Scan` method to ask it to do some scanning. If it successfully scanned a line, the method returns `true`, and we can then get the line by calling `Text()`.

One issue here is what to do if `Scan` returns false, meaning that no input was read. Well, we can't return an error from `Greet` without changing its signature, and we don't want everyone who calls this function to have to check for some error which will rarely happen in the real program: that would be irritating.

It also doesn't really make sense to choose a *default* name if we can't read one from the user. The best thing in this situation is probably just to do nothing and return.

Command-line interfaces

We've discussed in previous chapters how important it is to make sure that our system is always runnable, or at least not too far away from being runnable. One reason for this is to avoid the embarrassing situation where all your tests pass, but the program doesn't actually work when the user runs it.

If the program is well-designed, then as much of its important behaviour as possible will be implemented as an importable package. And if we've *built* that package guided by tests, then we should feel reasonably confident that it's correct.

But there's one important function that we can't easily test: `main`. And if that doesn't work, nothing else will. So it's worth asking: what could be *wrong* with the `main` function?

Well, quite a lot, actually. It could do nothing at all, or if it calls into the package, it could do it the wrong way, or it could ignore the user's command-line arguments and options, or misinterpret them. It could crash or panic if arguments are missing or invalid.

As long as all this logic is in `main`, it's difficult to test without compiling and invoking the binary. Sometimes that's an appropriate thing to do, and we'll talk about a convenient way to do it in the next chapter. But for now, let's think about how we could get that logic *out* of the `main` package and put it somewhere we can directly test it.

There's a bunch of stuff that almost all command-line tools have in common, regardless of what they actually do. They usually take arguments and flags, they may reference environment variables or configuration files, they might read from their standard input, and they usually write to their standard output or standard error.

Let's refer to these aspects of the program's behaviour as its *command-line interface* (CLI), as distinct from whatever it is that the program actually *does*.

So how can we test the command-line interface, specifically? For example, if the program takes a command-line argument, how can we test that supplying this argument actually has the desired effect?

For example, suppose we're writing a timer program whose job is simply to wait for a specified period, then exit. Users might interact with it by running the binary with a command-line argument, like this:

```
timer 10s
```

```
Starting a timer for 10s...
[time passes]
...done.
```

Since specifying the timer period is an important part of the user interface, we'd like to test that the program interprets that argument correctly. It will presumably do so by looking at the value of `os.Args`.

How could we supply arbitrary arguments to the program from a *test*, then?

One effective, but rather laborious, way would be to *build* it, by executing `go build` as a subprocess in our test, and then executing the resulting binary. That's doable, but it's an awful lot of work just to get some arbitrary values into `os.Args`.

One way around this problem, suggested by Mat Ryer and others, is to have `main` do nothing but call some other function and *pass* it `os.Args`:

```
func main() {
    timer.Main(os.Args)
}
```

You might recognise this as the same refactoring pattern we used in the previous section, on testing user interaction. In that earlier case, we passed I/O streams to the function. Here, we pass its command-line arguments instead.

Now it's easy to test the program's argument-handling behaviour by passing arbitrary arguments to `timer.Main`:

```
func TestTimer(t *testing.T) {
    t.Parallel()
    args := []string{"program", "arg1", "arg2", "arg3"}
    timer.Main(args)
    ... // test something
}
```

That takes care of *invoking* the program with arbitrary arguments, but it's still not clear how to test what it actually *does* as a result.

And maybe we don't need to. After all, what are we really testing here? Essentially, that the program correctly interprets its command-line arguments.

All the test needs to show is that, given the argument `10ms`, for example, the program understands that it *should* sleep for 10ms. The sleeping behaviour itself is not part of the *interface* testing. We can presume this is tested separately, in isolation from the CLI.

So how can we check *only* that the program correctly parses its arguments, without having to test the sleeping as well?

A simple way to do this is to break up its behaviour into two functions, `ParseArgs` and `Sleep`:

```
func main() {
    interval := timer.ParseArgs(os.Args)
    timer.Sleep(interval)
}
```

(Listing `timer/1`)

Assuming `Sleep` is independently tested, all we need to worry about here is testing `ParseArgs`. And because that's a pure function, this is now quite straightforward:

```
func TestParseArgs(t *testing.T) {
    t.Parallel()
    want := 10 * time.Millisecond
    got := timer.ParseArgs([]string{"timer", "10ms"})
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing timer/1)

With a program this simple, that's just about all we need. But you can imagine that if there were two or three configuration parameters, this API would start to become annoying to use. `ParseArgs` would have to return two or three values, and it's hard to scale much beyond that.

When that's the case, we can group together all the necessary config settings into a single struct. Since this object represents a timer that's configured in a specific way, let's call it simply a `Timer`:

```
type Timer struct {
    Interval time.Duration
}
```

(Listing timer/2)

It makes sense that `Sleep` now becomes a *method* on the timer, since it needs access to its configuration:

```
func (t Timer) Sleep() {
    time.Sleep(t.Interval)
}
```

(Listing timer/2)

The `main` function still looks reasonable, but now the configuration can scale to as many parameters as we want, without creating burdensome paperwork:

```
func main() {
    t := timer.NewTimerFromArgs(os.Args)
    t.Sleep()
}
```

(Listing timer/2)

Testing the command-line interface is now a matter of calling `NewTimerFromArgs` with some arbitrary arguments, and checking that we get back the expected value of `Timer`:

```
func TestNewTimerFromArgs(t *testing.T) {
    t.Parallel()
    args := []string{"program", "10s"}
    want := timer.Timer{
        Interval: 10 * time.Second,
    }
    got := timer.NewTimerFromArgs(args)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing `timer/2`)

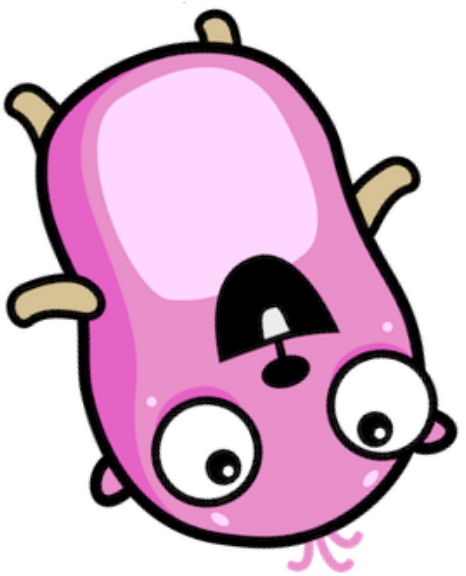
One answer to the vexed question of how to test CLI tools, then, is to decouple the command-line interface from the rest of the program, and test them both separately. A decoupled design is a better design, as we've seen throughout this book.

But sometimes there's no substitute for actually running the program directly and seeing what it does, especially when we want to test what output and error messages the *user* should see in different circumstances.

In the next chapter, we'll look at a neat way to do this, using *test scripts*.

9. Flipping the script

> *By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems.*
> —Alfred North Whitehead, "An Introduction to Mathematics"



Testing CLI tools in Go by running the binary directly can get complicated, as we mentioned in the previous chapter. We'd have to execute the Go tool to build the binary, then execute the *binary*, passing it arbitrary arguments, and then we'd have to check the exit status, parse the output, and so on.

That sounds like hard work, so let's not do that. Instead, let's flip the script.

In the real world, we'll probably end up running these tools from some *shell*, using a command line. So what if we could write Go tests for our tools using a simplified kind of shell script?

The `testscript` package lets us do exactly that. Let's try it out.

Introducing `testscript`

`testscript` defines a little "language" for writing test scripts, which we store as separate files in our project, along with our Go source code.

Let's see a simple example of such a script:

```
exec echo 'hello world'
stdout 'hello world\n'
```

(Listing `script/1`)

Pretty straightforward! You can probably guess what this does right away. The `exec` line runs `echo 'hello world'`, just as though we'd typed this command line at the terminal.

On the next line, `stdout` asserts something about the expected *output* of the program we just ran: that it contains at least the string `hello world`, followed by a newline.

That's all very well, but what does it have to do with Go? How, in other words, can we actually run such a script from a Go test, and connect its results with that of the test? `testscript` takes care of this too.

Let's take our simple `hello world` script, then, and write a Go test that runs it:

```
package script_test

import (
    "testing"

    "github.com/rogpeppe/go-internal/testscript"
)

func TestScript(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir: "testdata/script",
    })
}
```

(Listing `script/1`)

There's nothing special about the name `TestScript`, by the way; we could have called it `TestAnything`. This is just an ordinary Go test function, so far. The interesting part for us here is the call to `testscript.Run`. What's that about?

You'll notice that we pass our test's `t` as an argument, suggesting that `testscript.Run` could use it to fail the test. That makes sense, since `Run`'s job is to run a bunch of test scripts, each of which acts as a (parallel) subtest of `TestScript`.

The other thing we pass to `testscript.Run` is a `Params` struct which, as you probably guessed, supplies configuration parameters. The `Dir` parameter gives the path to the directory where our script files will live.

In our example, we've set it to `testdata/script`, though again the `script` here has no special meaning. We can put test scripts in any convenient subfolder of `testdata`.

Let's go ahead and create the `testdata/script` folder, and add a file in it named `hello.txtar` containing the text of our little example script:

```
exec echo 'hello world'
stdout 'hello world\n'
```

(Listing `script/1`)

We'll see what the `txtar` extension means shortly, but first, let's give this a try.

Provided that we've set everything up correctly, running our Go tests now will tell `testscript` to run this script, and the parent `TestScript` test will pass or fail depending on its result. Let's see what happens:

```
go test
```

```
PASS
```

Encouraging!

Running programs with `exec`

So what does the test script actually do? To put it another way, what are we really testing here?

Here's the first line again:

```
exec echo 'hello world'
```

(Listing `script/1`)

This tells `testscript` to execute the program `echo`, with the single command-line argument `hello world`. The effect of this is just as though the "user" had typed `echo 'hello world'` at the terminal.

This `exec` statement, though, is more than just a way of running programs; it's also an assertion. The program we execute must *succeed*: that is, its exit status must be zero. Otherwise, the `exec` assertion fails.

If this, or any other assertion in the script, fails, then the script terminates at that point. In other words, it acts like `t.Fatal` in a Go test. It marks the subtest as failed and exits, skipping the rest of the script.

So the simplest way we can test our programs using `testscript` is to use `exec` to run them directly from a test script, and see if they succeed. Each script runs in its own

unique temporary *work directory*, which will be automatically cleaned up after the script exits.

The work directory starts empty, which helps to make tests reproducible. If the program panics when some file or directory it relies on is not present, for example, we'd detect that as soon as we tried to run it from a script.

This is the kind of problem that's typically very difficult to catch in ordinary tests, because it only happens when you actually *run* the program.

> *What's true of every bug found in the field? It passed all the tests!*
> —Rich Hickey, "Simple Made Easy"

So test scripts go a long way to closing that "works on my laptop" loophole. If we could only run programs with certain arguments and assert their exit status, that would still be useful in itself, but we can do more.

Here's the second line of the script again:

```
stdout 'hello world\n'
```

(Listing `script/1`)

This is another assertion, this time about the contents of the standard output from the previous `exec` statement (we'll see how to match standard error shortly).

`stdout` asserts that the output will match some regular expression, in this case the string `hello world` followed by a newline.

The program can output *more* than that, but this script asserts that it produces *at least* `hello world\n` somewhere among its output. If it doesn't, the `stdout` assertion will fail, and this subtest will fail as a result.

We can see what that looks like by changing the `stdout` assertion to one that definitely *won't* succeed:

```
exec echo 'hello world'
stdout 'goodbye world\n'
```

We think this should fail the test, so let's see what happens.

Interpreting `testscript` output

We deliberately set up this test script to make it fail, and here's the result:

```
go test
```

```
--- FAIL: TestScript (0.00s)
    --- FAIL: TestScript/hello (0.00s)
        testscript.go:397:
            > exec echo 'hello world'
            [stdout]
```

```
            hello world
            > stdout 'goodbye world\n'
            FAIL: testdata/script/hello.txtar:2: no match for
            `goodbye world\n` found in stdout
```

There are a few interesting things about this output to note. First, the failing parent test is `TestScript`, and the name of the failing subtest represented by the script is `hello`. That's derived from the filename of the script, `hello.txtar`, but without the `.txtar` extension:

```
--- FAIL: TestScript/hello (0.00s)
```

Each line of the script is shown in the test output as it's executed, prefixed by a > character:

```
> exec echo 'hello world'
```

The standard output of the program run by `exec` is shown next in the test output:

```
[stdout]
hello world
```

And here's the line that actually triggered the test failure:

```
> stdout 'goodbye world\n'
FAIL: testdata/script/hello.txtar:2: no match for
`goodbye world\n` found in stdout
```

The `stdout` assertion is failing, because the program's output *didn't* match the given expression, so the test fails. Notice that as well as printing the failing line itself, the message also gives the script filename and line number where we can find it:

```
testdata/script/hello.txtar:2
```

Remember, all of this is happening because we invoked the script in our Go test named `TestScript`:

```
func TestScript(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir: "testdata/script",
    })
}
```

(Listing `script/1`)

It's important to note that the `TestScript` function, even though it's a Go test, doesn't test anything directly by *itself*.

Its job here is merely to invoke `testscript.Run`, delegating the actual testing to the scripts stored in `testdata/script`. At the moment we have only one script file in that directory, but if we were to add more, they would all be run as parallel subtests of `TestScript`.

How should we organise our test scripts, then? We could put all the scripts for the whole project in a single directory, and run them from a single test using `testscript.Run`. That's fine, but if there are many scripts, you may prefer to put them in separate directories and run them from distinct tests. Most projects, though, seem to get by just fine with a single directory of test scripts.

By the way, if we want to run just one specific script, we can do that too. Because each script is a subtest, we can run it in the same way as we would any individual test or subtest: by giving its name along with the `-run` flag to `go test`:

```
go test -run TestScript/hello
```

It's a good idea to keep each script fairly small and focused on one or two related behaviours. For example, you might have one script for checking the various kinds of "invalid input" behaviour, and another for the happy path. A script that does too much is difficult to read and understand, just like a test that does too much.

The `testscript` language

Test scripts are not shell scripts, though they look quite similar (we'll talk about some of the important differences later in this chapter). Neither are they Go code. Instead, they're written in `testscript`'s own *domain-specific language* (DSL), dedicated purely to running programs and asserting things about their behaviour.

The point of the `testscript` DSL is not to be a general-purpose programming language, or even to replace all tests written in Go. Like all the best tools, it does one thing well: it provides an elegant notation for writing automated tests of command-line programs.

Its keywords include the ones we've already seen, `exec` and `stdout`, plus a few others that we'll talk about later in this chapter. And its syntax is about as simple as it could possibly be, but no simpler.

A good way to think about the `testscript` DSL is as a kind of restricted, single-purpose version of the Unix shell. Restricted, but by no means restrictive: we can do some pretty sophisticated things with it, as we'll see in the rest of this chapter.

> *A really good language should be both clean and dirty: cleanly designed, with a small core of well understood and highly orthogonal operators, but dirty in the sense that it lets hackers have their way with it.*
> —Paul Graham, "Hackers & Painters: Big Ideas from the Computer Age"

Negating assertions with the `!` prefix

We've seen that we can use `stdout` to assert that a program's standard output matches some regular expression. What else can we assert in a script?

Interestingly, we can also assert something that the program's output *mustn't* match, by prefixing the `stdout` line with a `!` character.

```
exec echo 'hello world'
! stdout 'hello world\n'
```

The `!` is the equivalent of Go's "not" operator, so we can think of it as meaning "not" here, too. We're asserting that the standard output of the program run by the previous `exec` should not match the expression `hello world\n`.

What's the result of running this script? Well, the `exec` command line certainly *does* produce this output, so we'd expect our `! stdout` assertion to fail. Indeed, that's the case:

```
> ! stdout 'hello world\n'
FAIL: testdata/script/hello.txtar:2: unexpected match for
`hello world\n` found in stdout: hello world
```

We can also use the `!` operator to negate other assertions: for example, `exec`.

As we saw earlier, an `exec` by itself asserts that the given program succeeds: that is, returns zero exit status. What would `! exec` assert, then?

As you might expect, the effect of `! exec` is to assert that the program *fails*: in other words, that its exit status is *not* zero.

Let's try using it with the `cat` program, which among other things will print the contents of any file it's given.

If we try to use `cat` to print some file that doesn't exist in the work directory, then, it should return a non-zero exit status. We can write that as an assertion using `! exec`:

```
! exec cat doesntexist
```

(Listing `script/1`)

Why would we want to assert that running a certain command *fails*? Well, that's the expected behaviour of a command-line tool when the user specifies some invalid flag or argument, for instance. It's conventional to print an error message and exit with a non-zero exit status in this case.

For example, the public API of the `cat` program says that it should fail when given a non-existent file as argument. Our `! exec` example asserts exactly that behaviour.

What about the error message behaviour? Could we test that too? Conventionally, programs print error messages on the *standard error* stream, rather than standard output. In practice, both streams usually go to the same place (the user's terminal, for example), but they're independent in principle, and we can test them independently.

We're already familiar with `stdout`, so it's not surprising that the corresponding assertion for matching standard error is named `stderr`:

```
! exec cat doesntexist
stderr 'cat: doesntexist: No such file or directory'
```

(Listing script/1)

This asserts, not only that the given `cat` command fails, but also that it produces the expected message on its standard error stream.

In fact, we can go even further. Because, in this case, we don't expect `cat` to print anything to its standard output at all, we can assert that too, using `! stdout`:

```
! exec cat doesntexist
stderr 'cat: doesntexist: No such file or directory'
! stdout .
```

(Listing script/1)

Recall that the argument to `stdout` or `stderr` is a regular expression. Since the regular expression `.` matches any text, the effect of the third line here is to assert that the program produces *no* output on `stdout`.

The combined effect of these three lines, then, is to assert that:

- `cat doesntexist` fails
- it prints the expected message to `stderr`, and
- it prints nothing to `stdout`

What if we want to test that a program invoked with *valid* arguments prints standard output, but no standard error? In that case, we can write, for example:

```
exec echo success
stdout 'success'
! stderr .
```

(Listing script/1)

This script asserts that:

- `echo success` succeeds
- it prints the expected message to `stdout`, and
- it prints nothing to `stderr`

Passing arguments to programs

As we've seen in many of the examples so far, we can pass *arguments* to the program run by `exec`, by putting them after the program name:

```
# Execute 'echo', with the argument 'success'
exec echo success
```

It's important to know exactly *how* these arguments are passed to the program, though. Each space-separated word is treated as a distinct argument, unless those words are grouped together inside single quotes. This can make a big difference to how the program interprets them.

For example, suppose we want to give a program some argument that contains spaces, such as the filename `data file.txt`. This wouldn't work:

```
exec cat data file.txt
```

Because `data` and `file.txt` are passed as distinct arguments to `cat`, it thinks we're referring to two distinct files. Instead, we need to quote the filename, with single quotes:

```
exec cat 'data file.txt'
```

And, since the single quote character has this special effect of grouping space-separated arguments, you might be wondering how we can write a *literal* single quote character when we need to.

To do that, we write two *consecutive* single quote characters, as in this example:

```
exec echo 'Here''s how to escape single quotes'
```

This prints:

```
Here's how to escape single quotes
```

Importantly, and unlike in shell scripts, *double* quote characters have no quoting effect in test scripts. They're simply treated as literal double quote characters:

```
exec echo "This will print with literal double quotes"
```

Unlike the single quote, the double quote character has no special meaning in test scripts, so it will simply be passed on to the program as part of its arguments. This gives the following output, including the double quotes:

```
"This will print with literal double quotes"
```

Watch out for this, as it's easy to accidentally use double quotes when you meant to use single quotes.

Testing CLI tools with `testscript`

If all we could do with `testscript` were to run existing programs with certain arguments, and assert that they succeed (or fail), and produce certain outputs (or not), that would still be pretty useful.

But we're not limited to running *existing* programs. If we want to test our own binary, for example, we don't have to go through all the labour of compiling and installing it first, in order to execute it in a script. `testscript` can save us the trouble. Let's see how.

Suppose we're writing a program named `hello`, for example, whose job is simply to print a "hello world" message on the terminal. As it's an executable binary, it will need a `main` function. Something like this would do:

```
func main() {
    fmt.Println("hello world")
}
```

How can we test this? The `main` function can't be called directly from a Go test; it's invoked automatically when we run the compiled binary. We can't test it directly.

So we'll instead delegate its duties to some other function that we *can* call from a test: `hello.Main`, let's say.

We'll do nothing in the real `main` function except call `hello.Main` to do the printing, and then exit with whatever status value it returns:

```
func main() {
    status := hello.Main()
    os.Exit(status)
}
```

(Listing `hello/1`)

Fine. So we can deduce from this what the signature of `hello.Main` needs to be. It takes no arguments, and it returns an `int` value representing the program's exit status. Let's write it:

```
package hello

import "fmt"

func Main() int {
    fmt.Println("hello world")
    return 0
}
```

(Listing `hello/1`)

This is a very simple program, of course, so it doesn't *have* any reason to report a non-zero exit status, but it could if it needed to. Here, we just explicitly return `0`, so the program will always succeed.

Now that we've delegated all the real functionality of the program to this `hello.Main` function, we *could* call it from a Go test if we wanted to. But that wouldn't invoke it as a *binary*, only as a regular Go function call, which isn't really what we want here. For example, we probably want different values of `os.Args` for each invocation of the program.

What we need to do instead is to tell `testscript` to make our program available to scripts as a binary named `hello`. Here's what that looks like in our Go test code:

```
func TestMain(m *testing.M) {
    os.Exit(testscript.RunMain(m, map[string]func() int{
        "hello": hello.Main,
    }))
}
```

(Listing hello/1)

The function `TestMain` is special to Go: it doesn't test anything itself, but its job is usually to set something up in advance, before any test actually runs.

What this `TestMain` is doing is calling `testscript.RunMain`. What does *that* do? Well, it runs all our Go tests, but before it does that, it also sets up any custom programs that we want to use in scripts.

To do that, we pass this map to `RunMain`, connecting the name of our desired binary (`hello`) with its delegate `main` function (`hello.Main`):

```
map[string]func() int{
    "hello": hello.Main,
}
```

This tells `testscript` to create an executable binary named `hello`, whose `main` function will call `hello.Main`. This binary will be installed in a temporary directory (not the script's work directory), and that directory will be added to the `$PATH` variable in the environment of all our scripts.

If the magic works the way it should, then, we'll be able to use `exec` in a script to run the `hello` program, just as if we'd compiled and installed it manually. After the tests have finished, the binary and its temporary directory will be deleted automatically.

Let's give it a try. We'll create a script with the following contents:

```
exec hello
stdout 'hello world\n'
```

(Listing hello/1)

Now we'll add a test that runs this script using `testscript.Run`, as before:

```
func TestHello(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir: "testdata/script",
    })
}
```

(Listing hello/1)

Here's the result of running `go test`:

```
PASS
```

It worked! But are we *really* executing the binary implemented by `hello.Main`, or does there just happen to be some unrelated program named `hello` somewhere on our system? You can't be too careful these days.

To find out, let's change the `hello.Main` function to print something slightly different, and see if that makes the test fail. This ought to prove that `testscript` is really running the program we think it is:

```
func HelloMain() int {
    fmt.Println("goodbye world")
    return 0
}
```

Here's the result:

```
> exec hello
[stdout]
goodbye world
> stdout 'hello world\n'
FAIL: testdata/script/hello.txtar:2: no match for `hello world\n`
found in stdout
```

Proof positive that we're executing the right `hello` program, I think you'll agree. Let's also check that returning anything other than 0 from `hello.Main` causes the `exec` assertion to fail, as we would expect:

```
func HelloMain() int {
    fmt.Println("hello world")
    return 1
}
```

Here's the result:

```
> exec hello
[stdout]
hello world
[exit status 1]
FAIL: testdata/script/hello.txt:1: unexpected command failure
```

One thing to be careful of when defining custom commands in this way is to *remember to call `os.Exit`* with the result of `testscript.RunMain`. For example, suppose we were to write a `TestMain` like this:

```
func TestMain(m *testing.M) {
```

```
    testscript.RunMain(m, map[string]func() int{
        "hello": hello.Main,
    })
    // oops, forgot to use 'status'
}
```

This *looks* reasonable, but the status value returned by `RunMain` (which is the exit status of our custom command) is ignored. Implicitly, we exit with a zero exit status, meaning that the `hello` command would always appear to "succeed", regardless of what `hello.Main` actually returns.

So if you find that your custom command always succeeds, even when it's supposed to fail, check that you have the necessary call to `os.Exit` in `TestMain`.

Great. Now we can test that our program succeeds and fails when it should. What about more complicated behaviours, such as those involving command-line arguments?

For example, let's extend our `hello` program to take a command-line argument, and fail if it's not provided. Since all the real work is done in `hello.Main`, that's where we need to make this change:

```
func Main() int {
    if len(os.Args[1:]) < 1 {
        fmt.Fprintln(os.Stderr, "usage: hello NAME")
        return 1
    }
    fmt.Println("Hello to you,", os.Args[1])
    return 0
}
```

(Listing `hello/2`)

This program now has *two* behaviours. When given an argument, it should print a greeting using that argument, and succeed. On the other hand, when the argument is missing, it should print an error message and fail.

Let's test both behaviours in the same script:

```
# With no arguments, fail and print a usage message
! exec hello
! stdout .
stderr 'usage: hello NAME'

# With an argument, print a greeting using that value
exec hello Joumana
stdout 'Hello to you, Joumana'
```

```
! stderr .
```

(Listing hello/2)

The ability to define and run custom programs in this way is the key to using `testscript` to test command-line tools. We can invoke the program with whatever arguments, environment variables, and supporting files are required to test a given behaviour. In this way we can test even quite complex behaviours with a minimum of code.

And that's no surprise, because `testscript` is derived directly from the code used to test the Go tool itself, which is probably as complex a command-line tool as any. It's part of the very handy `go-internal` repository:

https://github.com/rogpeppe/go-internal

Checking the test coverage of scripts

One especially neat feature of `testscript` is that it can even provide us with *coverage* information when testing our binary. That's something we'd find hard to do if we built and executed the binary ourselves, but `testscript` makes it work seemingly by magic:

```
go test -coverprofile=cover.out
```

```
PASS
coverage: 0.0% of statements
total coverage: 100.0% of statements
```

Though the first figure shows 0% coverage, this only tells us what percentage of statements are executed *directly*, by being called from our Go tests, and the answer is zero. Instead, our `hello.Main` function is being executed *indirectly*, in a subprocess.

The figure to look at here, then, is the "total coverage". This gives the percentage of statements executed by scripts. Since our `hello` script executes both of the two possible code paths in `hello.Main`, it covers it completely. Thus, 100% of statements.

Just to check that this is really being calculated properly, let's try deliberately reducing the coverage, by testing *only* the happy path behaviour in our script:

```
# With an argument, print a greeting using that value
exec hello Joumana
stdout 'hello to you, Joumana'
! stderr .
```

We're no longer causing the "if no arguments, error" code path to be executed, so we should see the total coverage go down:

```
total coverage: 60.0% of statements
```

Since we've generated a coverage profile (the file `cover.out`), we can use this with the `go tool cover` command, or our IDE. As we saw in the chapter on mutation testing,

the coverage profile can show us *exactly* which statements are and aren't executed by tests (including test scripts). If there are important code paths we're not currently covering, we can add or extend scripts so that they test those behaviours, too.

As we discussed in that chapter, test coverage isn't always the most important guide to the quality of our tests, since it only proves that statements were *executed*, not what they *do*. But it's very useful that we can test command-line tools and other programs *as binaries* using `testscript`, without losing our test coverage statistics.

Comparing output with files using `cmp`

Let's look at some more sophisticated ways we can test input and output from command-line tools using `testscript`.

For example, suppose we want to compare the program's output not against a string or regular expression, but against the contents of a *file*. As we've seen in previous chapters, these are sometimes referred to as *golden files*.

We can supply a golden file as part of the script file itself, delimiting its contents with a special *marker line* beginning and ending with a double hyphen (`--`).

Here's an example:

```
exec hello
cmp stdout golden.txt

-- golden.txt --
hello world
```

(Listing `hello/1`)

The marker line containing `golden.txt` begins a *file entry*: everything following the marker line will be written to `golden.txt` and placed in the script's work directory before it starts executing. We'll have more to say about file entries in a moment, but first, let's see what we can *do* with this file.

The `cmp` assertion can compare two files to see if they're the same. If they match exactly, the test passes. If they don't, the failure will be accompanied by a diff showing which parts didn't match.

If the program's output *doesn't* match the golden file, as it won't in this example, we'll see a failure message like this:

```
> exec hello
[stdout]
hello world
> cmp stdout golden.txt
--- stdout
+++ golden.txt
```

```
@@ -1,1 +0,0 @@
-hello world
@@ -0,0 +1,1 @@
+goodbye world

FAIL: testdata/script/hello.txtar:2: stdout and golden.txt differ
```

Alternatively, we can use `!` to negate the comparison, in which case the files must *not* match, and the test will fail if they do:

```
exec echo hello
! cmp stdout golden.txt


-- golden.txt --
goodbye world
```

(Listing hello/1)

The first argument to `cmp` can be the name of a file, but we can also use the special name `stdout`, meaning the standard output of the previous `exec`. Similarly, `stderr` refers to the standard error output.

If the program produces different output depending on the value of some environment variable, we can use the `cmpenv` assertion. This works like `cmp`, but interpolates environment variables in the golden file:

```
exec echo Running with home directory $HOME
cmpenv stdout golden.txt


-- golden.txt --
Running with home directory $HOME
```

(Listing hello/1)

When this script runs, the `$HOME` in the `echo` command will be expanded to the actual value of the `HOME` environment variable, whatever it is. But because we're using `cmpenv` instead of `cmp`, we *also* expand the `$HOME` in the golden file to the same value.

So, assuming the command's output is correct, the test will pass. This prevents our test from flaking when its behaviour depends on some environment variable that we don't control, such as `$HOME`.

More matching: `exists`, `grep`, and `-count`

Some programs create files directly, without producing any output on the terminal. If we just want to assert that a given file *exists* as a result of running the program, without worrying about the file's contents, we can use the `exists` assertion.

For example, suppose we have some program `myprog` that writes its output to a file specified by the `-o` flag. We can check for the existence of that file after running the program using `exists`:

```
exec myprog -o results.txt
exists results.txt
```

And if we *are* concerned about the exact contents of the results file, we can use `cmp` to compare it against a golden file:

```
exec myprog -o results.txt
cmp results.txt golden.txt


-- golden.txt --
hello world
```

If the two files match exactly, the assertion succeeds, but otherwise it will fail and produce a diff showing the mismatch. If the results file doesn't exist at all, that's also a failure.

On the other hand, if we don't need to match the entire file, but only part of it, we can use the `grep` assertion to match a regular expression:

```
exec myprog -o results.txt
grep '^hello' results.txt


-- golden.txt --
hello world
```

A `grep` assertion succeeds if the file matches the given expression *at least once*, regardless of how many matches there are. On the other hand, if it's important that there are a specific *number* of matches, we can use the `-count` flag to specify how many :

```
grep -count=1 'beep' result.txt


-- result.txt --
beep beep
```

In this example, we specified that the pattern `beep` should only match once in the target file, so this will fail:

```
> grep -count=1 'beep' result.txt
[result.txt]
beep beep

FAIL: beep.txtar:1: have 2 matches for `beep`, want 1
```

Because the script's work directory is automatically deleted after the test, we can't look at its contents—for example, to figure out why the program's not behaving as expected. To keep this directory around for troubleshooting, we can supply the `-testwork` flag to `go test`.

This will preserve the script's work directory intact, and also print the script's environment, including the `WORK` variable that tells us where to *find* that directory:

```
--- FAIL: TestScript/hello (0.01s)
    testscript.go:422:
        WORK=/private/var/folders/.../script-hello
        PATH=...
        ...
```

The `txtar` format: constructing test data files

We've seen already in this chapter that we can create files in the script's work directory, using this special syntax to indicate a named *file entry*:

```
-- golden.txt --
... file contents ...
```

(Listing `hello/1`)

The line beginning `--`, called a *file marker line*, tells `testscript` that everything following this line (until the next file marker) should be treated as the contents of `golden.txt`.

A file marker line must begin with two hyphens and a space, as in our example, and end with a space and two hyphens. The part in between these markers specifies the filename, which will be stripped of any surrounding whitespace.

In fact, we can create as many additional files as we want, simply by adding more file entries delimited by marker lines:

```
exec cat a.txt b.txt c.txt

-- a.txt --
...

-- b.txt --
...

-- c.txt --
...
```

(Listing `hello/1`)

Each marker line denotes the beginning of a new file, followed by zero or more lines of content, and ending at the next file marker line, if there is one. All these files will be created in the work directory before the script starts, so we can rely on them being present.

If we need to create folders, or even whole trees of files and folders, we can do that by using slash-separated paths in the file names:

```
-- misc/a.txt --
...

-- misc/subfolder/b.txt --
...

-- extra/c.txt --
...
```

(Listing hello/1)

When the script runs, the following tree of files will be created for it to use:

```
$WORK/
    misc/
        a.txt
        subfolder/
            b.txt
    extra/
        c.txt
```

This is a very neat way of constructing an arbitrary set of files and folders for the test, rather than having to create them using Go code, or copying them from somewhere in `testdata`. Instead, we can represent any number of text files as part of a *single* file, the test script itself.

This representation, by the way, is called `txtar` format, short for "text archive", and that's also the file extension we use for such files.

While this format is very useful in conjunction with `testscript`, a `txtar` file doesn't have to be a test script. It can also be used independently, as a simple way of combining multiple folders and text files into a single file.

To use the `txtar` format in programs directly, import the `txtar` package. For example, if you want to write a tool that can read data from such archives, or that allows users to supply input as `txtar` files, you can do that using the `txtar` package.

Supplying input to programs using `stdin`

Now that we know how to create arbitrary files in the script's work directory, we have another fun trick available. We can use one of those files to supply *input* to a program, as though it were typed by an interactive user, or piped into the program using a shell.

For example, remember the `Greet` function from the previous chapter? It prompted the user to enter their name, then printed a nice greeting to them. We were able to decouple it from the real terminal, for testing purposes, by passing it an `io.Reader` (for input) and an `io.Writer` (for output).

That's fine, but now we have a simpler alternative: we can run the program as a binary, meaning that it doesn't need special I/O streams. It can use standard input and output directly.

Let's see if we can use this idea to test our `Greet` example from the previous chapter. First, we'll set up a delegate `main` function that returns an exit status value, as we did with the `hello` program:

```
func Main() int {
    fmt.Println("Your name? ")
    scanner := bufio.NewScanner(os.Stdin)
    if !scanner.Scan() {
        return 1
    }
    fmt.Printf("Hello, %s!\n", scanner.Text())
    return 0
}
```

(Listing greet/6)

Just as with `hello`, we'll use the map passed to `testscript.RunMain` to associate our new custom program `greet` with the `greet.Main` function:

```
func TestMain(m *testing.M) {
    os.Exit(testscript.RunMain(m, map[string]func() int{
        "greet": greet.Main,
    }))
}
```

(Listing greet/6)

We'll add the usual parent test that calls `testscript.Run`:

```
func TestGreet(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir: "testdata/script",
```

```
    })
}
```

(Listing greet/6)

And here's a test script that runs the `greet` program, supplies it with fake "user" input via `stdin`, and checks its output:

```
stdin input.txt
exec greet
stdout 'Hello, John!'

-- input.txt --
John
```

(Listing greet/6)

First, the `stdin` statement specifies that standard input for the next program run by `exec` will come from `input.txt` (defined at the end of the script file, using a `txtar` file entry).

Next, we `exec` the `greet` command, and verify that its output matches `Hello, John!`. Very simple, but a powerful way to simulate any amount of user input for testing. Indeed, we could simulate a whole "conversation" with the user.

Suppose the program asks the user for their favourite food as well as their name, then prints a customised dining invitation:

```
func Main() int {
    fmt.Println("Your name? ")
    scanner := bufio.NewScanner(os.Stdin)
    if !scanner.Scan() {
        return 1
    }
    name := scanner.Text()
    fmt.Println("Your favourite food? ")
    if !scanner.Scan() {
        return 1
    }
    food := scanner.Text()
    fmt.Printf("Hello, %s. Care to join me for some %s?\n", name,
        food)
    return 0
}
```

(Listing greet/7)

How can we test this with a script? Because the program scans user input a line at a time, we can construct our "fake input" file to contain the user's name and favourite food on consecutive lines:

```
stdin input.txt
exec greet
stdout 'Hello, Kim. Care to join me for some barbecue?'

-- input.txt --
Kim
barbecue
```

(Listing greet/7)

We can go even further with `stdin`. We're not restricted to supplying input from a file; we can also use the *output* of a previous `exec`. This could be useful when one program is supposed to generate output which will be piped to another, for example:

```
exec echo hello
stdin stdout
exec cat
stdout 'hello'
```

First, we execute `echo hello`. Next, we say `stdin stdout`, meaning that the input for the next `exec` should be the output of the previous `exec` (in this case, that input will be the string `hello`).

Finally, we execute the `cat` command, which copies its input to its output, producing the final result of this "pipeline": `hello`. You can chain programs together using `stdin stdout` as many times as necessary.

This isn't quite like a shell pipeline, though, because there's no concurrency involved: `stdin` reads its entire input before continuing. This is fine for most scripts, but just be aware that the script won't proceed until the previous `exec` has finished and closed its output stream.

File operations

Just as in a traditional shell script, we can copy one file to another using `cp`:

```
cp a.txt b.txt
```

However, the first argument to `cp` can also be `stdout` or `stderr`, indicating that we want to copy the output of a previous `exec` to some file:

```
exec echo hello
cp stdout tmp.txt
```

We can also use `mv` to *move* a file (that is, rename it) instead of copying:

```
mv a.txt b.txt
```

We can also create a directory using `mkdir`, and then copy multiple files into it with `cp`:

```
mkdir data
cp a.txt b.txt c.txt data
```

The `cd` statement will change the current directory for subsequent programs run by `exec`:

```
cd data
```

To delete a file or directory, use the `rm` statement:

```
rm data
```

When used with a directory, `rm` acts recursively, like the shell's `rm -rf`: it deletes all contained files and subdirectories before deleting the directory itself.

To create a symbolic link from one file or directory to another, we can use the `symlink` statement, like this:

```
mkdir target
symlink source -> target
```

Note that the `->` is required, and it indicates the "direction" of the symlink. In this example, the link `source` will be created, pointing to the existing directory `target`.

Differences from shell scripts

While test scripts look a lot like shell scripts, they don't have any kind of control flow statements, such as loops or functions. Failing assertions will bail out of the script early, but that's it. On the plus side, this makes test scripts pretty easy to read and understand.

There *is* a limited form of conditional statement, as we'll see later in this chapter, but let's first look at a few other ways in which test scripts differ from shell scripts.

For one thing, they don't actually *use* a shell: commands run by `exec` are invoked directly, without being parsed by a shell first. So some of the familiar facilities of shell command lines, such as *globbing* (using wildcards such as `*` to represent multiple filenames) are not available to us directly.

That's not necessarily a problem, though. If we need to expand a glob expression, we can simply ask the shell to do it for us:

```
# List all files whose names begin with '.'
exec sh -c 'ls .*'
```

Similarly, we can't use the pipe character (`|`) to send the output of one command to another, as in a shell pipeline. Actually, we already know how to chain programs together in a test script using `stdin stdout`. But, again, if we'd rather invoke the shell to do this, we can:

```
# count the number of lines printed by 'echo hello'
exec sh -c 'echo hello | wc -l'
```

Comments and phases

A # character in a test script begins a comment, as we saw in the examples in the previous section. Everything else until the end of the line is considered part of the comment, and ignored. That means we can add comments after commands, too:

```
exec echo hello # this comment will not appear in output
```

Actually, comments in scripts aren't *entirely* ignored. They also delimit distinct sections, or *phases*, in the script.

For example, we can use a comment to tell the reader what's happening at each stage of a script that does several things:

```
# run an existing command: this will succeed
exec echo hello

# try to run a command that doesn't exist
exec bogus
```

This is informative, but that's not all. If the script fails at any point, `testscript` will print the detailed log output, as we've seen in previous examples. However, only the log of the *current* phase is printed; previous phases are suppressed. Only their comments appear, to show that they succeeded.

Here's the result of running the two-phase script shown in the previous example:

```
# run an existing command: this will succeed (0.003s)
# try to run a command that doesn't exist (0.000s)
> exec bogus
[exec: "bogus": executable file not found in $PATH]
```

Notice that the `exec` statement from the first phase (running `echo hello`) isn't shown here. Instead, we see only its associated comment, which `testscript` treats as a description of that phase, followed by the time it took to run:

```
# run an existing command: this will succeed (0.003s)
```

Separating the script into phases using comments like this can be very helpful for keeping the failure output concise, and of course it makes the script more readable too.

Conditions

Because we're running real programs on a real computer that we have limited information about, we may need to specify some *conditions* governing whether or not to perform an action.

For example, we can't always guarantee in advance that the program we want to run will actually be available on the test system. We can check for its existence by prefixing the script line with a condition:

```
[exec:sh] exec echo yay, we have a shell
```

The square brackets contain a condition, which can be true or false. If the condition is true, the rest of the line is executed. Otherwise, it's ignored.

In this example, the condition `[exec:sh]` is true if there is a program named `sh` in our `$PATH` that we have permission to execute. If there isn't, this line of the script will be ignored.

On the other hand, if we need to check for a program at a specific path, we can give that path as part of the condition:

```
[exec:/bin/sh] exec echo yay, we have /bin/sh
```

There are a few other built-in conditions available, but perhaps the most useful to us are the current Go version, the current operating system, and the CPU architecture of this machine:

```
# 'go1.x' is true if this is Go 1.x or higher
[go1.16] exec echo 'We have at least Go 1.16'

# Any known value of GOOS is also a valid condition
[darwin] exec echo 'We''re on macOS'

# As is any known value of GOARCH
[!arm64] exec echo 'This is a non-arm64 machine'
```

As you can see from that example, we can also *negate* a condition by prefixing it with `!`. For example, the condition `[!arm64]` is true only if the value of `GOARCH` is *not* `arm64`.

We can use a condition to control other script statements, too, not just `exec`. For example, we can use the `skip` statement to skip the test in some circumstances:

```
# Skip this test unless we have Go 1.18 or later
[!go1.18] skip

# Skip this test on Linux
[linux] skip
```

The `unix` condition is true when the target OS is one of those that Go considers "Unix-like", including Linux, macOS, FreeBSD, and others.

```
[unix] exec echo 'It''s a UNIX system! I know this!'
```

In Go 1.19, the only supported operating systems for which `unix` is *not* true are `js`, `nacl`, `plan9`, `windows`, and `zos`.

Setting environment variables with `env`

Since command-line tools often use environment variables, it's important to be able to set and use these in scripts. To set a particular variable for the duration of the script, use an `env` statement:

```
env MYVAR=hello
```

For example, to test that some program `myprog` fails and prints an error message when the environment variable it needs has no value, we could write:

```
env AUTH_TOKEN=
! exec myprog
stderr 'AUTH_TOKEN must be set'
```

In scripts, we can refer to the value of an environment variable by using the `$` token followed by the name of the variable, just like in a shell script. This will be replaced by the value of the variable when the script runs. For example:

```
exec echo $PATH
```

This will cause `echo` to print the contents of `$PATH`, whatever it happens to be when the script is run.

Each script starts with an empty environment, apart from `$PATH`, and a few other predefined variables. For example, `HOME` is set to `/no-home`, because many programs expect to be able to find the user's home directory in `$HOME`.

If scripts need to know the absolute path of their work directory, it will be available as `$WORK`. For example, we could use `$WORK` as part of the value for another environment variable:

```
env CACHE_DIR=$WORK/.cache
```

To make scripts deterministic, though, the actual *value* of this variable will be replaced in the `testscript` output by the literal string `$WORK`. In fact, the value of `$WORK` will be different for every run and every test, but it will always *print* as simply `$WORK`.

Because some programs also expect to read the variable `$TMPDIR` to find out where to write their temporary files, `testscript` sets this too, and its value will be `$WORK/.tmp`.

A useful variable for cross-platform testing is `$exe`. Its value on Windows is `.exe`, but on all other platforms, it's the empty string. So you can specify the name of some program `prog` like this:

```
exec prog$exe
```

On Windows, this evaluates to running `prog.exe`, but on non-Windows systems, it will be simply `prog`.

Passing values to scripts via environment variables

Sometimes we need to supply dynamic information to a script, for example some value that's not known until the test is actually running. We can pass values like this to scripts using the environment, by using the `Setup` function supplied to `testscript.Run` as part of `Params`.

For example, suppose the program run by the script needs to connect to some server, but the address won't be known until we start the server in the test. Let's say it's generated by the same `randomLocalAddr` helper we used in the previous chapter.

Once we know the generated address, we can store it in a suitable environment variable, using the `Setup` function supplied to `testscript.Run`:

```
func TestScriptWithExtraEnvVars(t *testing.T) {
    t.Parallel()
    addr := randomLocalAddr(t)
    testscript.Run(t, testscript.Params{
        Dir: "testdata/script",
        Setup: func(env *testscript.Env) error {
            env.Setenv("SERVER_ADDR", addr)
            return nil
        },
    })
}
```

(Listing env/1)

The `Setup` function is run after any files in the test script have been extracted, but before the script itself starts. It receives an `Env` object containing the predefined environment variables to be supplied to the script.

At this point, our `Setup` function can call `env.Setenv` to add any extra environment variables it wants to make available in the script. For example, we can set the `SERVER_ADDR` variable to whatever the test server's address is.

A script can then get access to that value by referencing `$SERVER_ADDR`, just as it would with any other environment variable:

```
exec echo $SERVER_ADDR
stdout '127.0.0.1:[\d]+'
```

(Listing env/1)

The `stdout` assertion here is just to make sure that we did successfully set the `SERVER_ADDR` variable to something plausible. In a real test we could use this value as the address for the program to connect to.

Running programs in background with `&`

One nice feature of typical shells is the ability to run programs in the *background*: that is, concurrently. For example, in most shells we can write something like:

```
sleep 10 &
```

This starts the `sleep` program running as a separate process, but continues to the next line of the script so that we can do something else concurrently. We can do the same in a test script:

```
exec sleep 10 &
```

The trailing `&` ("ampersand") character tells `testscript` to execute the `sleep` program in the background, but carry on. Its output will be buffered so that we can look at it later, if we need to. We can start as many programs in the background as necessary, and they will all run concurrently with the rest of the script.

Because backgrounding a program with `&` doesn't wait for it to complete before continuing, we need a way to do that explicitly. Otherwise, the script might end before the background program is done. To do that, we use the `wait` statement:

```
exec sleep 10 &
wait
```

This pauses the script until all backgrounded programs have exited, and then continues. At this point the output from those programs, in the order they were started, will be available for testing using the `stdout`, `stderr`, and `cmp` assertions, just as if they'd been run in the foreground.

Why might we want to do this? Well, some programs might need some background service to be available in order for us to test them, such as a database or a web server.

At the end of the script, all programs still running in background will be stopped using the `os.Interrupt` signal (that is, the equivalent of typing Ctrl-C at the terminal), or, if necessary, `os.Kill`.

Let's see an example. Suppose we have some Go program that implements a web server, and we want to start it on some arbitrary port, then connect to it with `curl` and check that we get the expected welcome message. How could we test that?

First, let's set up a delegate `Main` function to run the server program, taking its listen address from `os.Args`. We'll associate this with a custom program named `listen`, using `testscript.RunMain` in the same way that we've done for previous examples.

Again, we'll use `randomLocalAddr` to generate a listen address, and put this value in the environment variable `SERVER_ADDR`, using `Setup`. The details are in Listing env/2, but they're essentially the same as the previous examples in this chapter.

Now we can write the script itself:

```
exec listen $SERVER_ADDR &
exec curl -s --retry-connrefused --retry 1 $SERVER_ADDR
stdout 'Hello from the Go web server'
wait
```

(Listing env/2)

The first thing we do is use `exec` to start our `listen` program on the generated address. Notice that we use the `&` token to execute the program concurrently, in background.

The next line of the script uses the well-known `curl` tool to make an HTTP connection to the given address, and print the response. Just as in our `ListenAsync` example from the previous chapter, if we try to connect *instantly* it probably won't work. We need to tell `curl` to retry if the first connection attempt is refused.

Once the connection has succeeded, we use `stdout` to assert that the server responds with the message "Hello from the Go web server".

If the script simply ended here, the background `listen` process would be killed with an interrupt signal, which would cause the test to fail. We don't want that, so we add a `wait` statement.

But if the server listened forever, then the `wait` statement would also wait for it forever (or, at least, until the test timed out). We don't want that either, so we've arranged for the server to politely shut itself down after serving exactly one request. Normal servers wouldn't do this, but it just makes this demonstration clearer.

Therefore, the `wait` statement pauses the script just long enough to allow this tidy shutdown to happen, and then exits.

The standalone `testscript` runner

The `testscript` language is so useful that it would be great to be able to use it even *outside* the context of Go tests. For example, we might like to write test scripts as standalone programs for use in automation pipelines and CI systems, or in non-Go software projects.

Wouldn't it be nice if we could write a test script and simply run it directly from the command line, without having to write Go code to do so? Well, you'll be delighted to know that we can do exactly this, by installing the standalone `testscript` tool:

```
go install github.com/rogpeppe/go-internal/cmd/testscript@latest
```

To use it, all we need to do is give the path to a script, or multiple scripts:

```
testscript testdata/script/*
```

This will run each script in turn and print `PASS` if it passes (and the comments describing each successful phase). Otherwise, it will print `FAIL`, accompanied by the same failure output we'd see when running the script in a Go test.

If any script fails, the exit status from `testscript` will be `1`, which is useful for detecting failures in automations.

If we want to log what the script is doing, we can use the `-v` flag, which prints verbose output whether the script passes or fails:

```
testscript -v echo.txtar
```

```
WORK=$WORK
[rest of environment omitted]

> exec echo hello
[stdout]
hello
> stdout 'hello'
PASS
```

To pass environment variables to scripts, we can specify them using the `-e` flag. Repeat the `-e` flag for each variable-value pair:

```
testscript -e VAR1=hello -e VAR2=goodbye script.txtar
```

Just as when running scripts from Go tests, each script will get its own work directory which is cleaned up afterwards. To preserve this directory and its contents, for example for troubleshooting, use the `-work` flag:

```
testscript -work script.txtar
```

```
temporary work directory: /var/.../testscript1116180846
PASS
```

By the way, if you use the VS Code editor, there's a useful extension available for syntax highlighting `txtar` files and scripts, called `vscode-txtar`. Not only does it highlight `testscript` syntax, it's also smart enough (in most cases) to identify the language in included files (for example, `.go` files) and highlight *them* appropriately. This makes editing non-trivial scripts a good deal easier.

Just for fun, we can even use `testscript` as an *interpreter*, using the "shebang line" syntax available on some Unix-like operating systems, including Linux and macOS.

For example, we could create a file named `hello.txtar` with the following contents:

```
#!/usr/bin/env testscript
exec echo hello
stdout 'hello'
```

The line beginning `#!` tells the system where to find the interpreter that should be used to execute the rest of the script. If you've wondered why some shell scripts start with `#!/bin/sh`, for example, now you know.

So, if we alter the file's permissions to make it executable, we can run this script directly from the command line:

```
chmod +x hello.txtar
./hello.txtar
PASS
```

Test scripts as issue repros

Because the flexible `txtar` format lets us represent not only a test script, but also multiple files and folders as a single copy-pastable block of text, it's a great way to submit test case information along with bug reports. Indeed, it's often used to report bugs in Go itself.

If we have found some bug in a program, for example, we can open an issue with the maintainers and provide them with a test script that very concisely demonstrates the problem:

```
# I was promised 'Go 2', are we there yet?
exec go version
stdout 'go version go2.\d+'
```

The maintainers can then run this script themselves to see if they can reproduce the issue:

```
testscript repro.txtar
# I was promised 'Go 2', are we there yet? (0.016s)
> exec go version
[stdout]
go version go1.18 darwin/amd64

> stdout 'go version go2.\d+'
FAIL: script.txt:3: no match for `go version go2.\d+` found in stdout
```

Supposing they're willing or able to fix the bug (if indeed it *is* a bug), they can check their proposed fix using the same script:

```
testscript repro.txtar

# I was promised 'Go 2', are we there yet? (0.019s)
PASS
```

Indeed, they can add the script to the project's own tests. By making it easy to submit, reproduce, and discuss test cases, `testscript` can even be a way to gently introduce automated testing into an environment where it's not yet seriously practiced:

> *How do you spread the use of automated testing? One way is to start asking for explanations in terms of test cases: "Let me see if I understand what you're saying. For example, if I have a Foo like this and a Bar like that, then the answer should be 76?"*
> —Kent Beck, "Test-Driven Development by Example"

Test scripts as... tests

Didn't we already talk about this? We started out describing `testscript` as an extension to Go tests. In other words, running a script via the `go test` command. But what if you want to flip the script *again*, by running the `go test` command from a script?

I'll give you an example. I maintain all the code examples from this and other books in GitHub repos. I'd like to check, every time I make changes, that all the code still behaves as expected. How can I do that?

I could manually run `go test` in every module subdirectory, but there are dozens in the tpg-tests repo alone, so this would take a while. I could even write a shell script that visits every directory and runs `go test`, and for a while that's what I did. But it's not quite good enough.

Because I'm developing the example programs step by step, some of the example tests *don't* pass, and that's by design. For example, if I start by writing a failing test for some feature, in the "guided by tests" style I prefer, then I *expect* it to fail, and that's what my script needs to check.

How can I specify that the `go test` command should fail in some directories, but pass in others? That sounds just like the `exec` assertion we learned about earlier, doesn't it? And we can use exactly that:

```
! exec go test
stdout 'want 4, got 5'
```

(Listing `double/1`)

You can see what this is saying: it's saying the tests should fail, and the failure output should contain the message "want 4, got 5". And that's exactly what the code does, so this *script* test passes. It might seem a little bizarre to have a test that passes if and only if another test fails, but in this case that's just what we want.

Many of the examples are supposed to pass, of course, so *their* test scripts specify that instead:

```
exec go test
```

(Listing double/2)

And in most cases that's the only assertion necessary. But there's something missing from these scripts that you may already have noticed. Where's the *code*?

We talked earlier in the chapter about the neat feature of the `txtar` format that lets us include arbitrary text files in a test script. These will automatically be created in the script's work directory before it runs. But we don't *see* any Go files included in these test scripts, so where do they come from? What code is the `go test` command testing?

Well, it's in the GitHub repo, of course: Listing double/2, for example. I don't want to have to copy and paste it all into the test script, and even if I did, that wouldn't help. The test script is supposed to tell me if the code in the *repo* is broken, not the second-hand copy of it I pasted into the script.

We need a way of including files in a script on the fly, so to speak. In other words, what we'd like to do is take some directory containing a bunch of files, bundle them all up in `txtar` format, add some test script code, and then *run* that script with `testscript`. Sounds complicated!

Not really. The txtar-c tool does exactly this. Let's install it:

```
go install github.com/bitfield/txtar-c@latest
```

Now we can create a `txtar` archive of any directory we like:

```
txtar-c .
```

The output is exactly what you'd expect: a `txtar` file that includes the contents of the current directory. Great! But we also wanted to include our `exec` assertion, so let's put that in a file called `test.txtar`, and have the tool automatically include it as a script:

```
txtar-c -script test.txtar .
```

We now have exactly the script that we need to run, and since we don't need to save it as a file, we can simply pipe it straight into `testscript`:

```
txtar-c -script test.txtar . |testscript
PASS
```

Very handy! So `txtar-c` and `testscript` make perfect partners when we want to run some arbitrary test script over a bunch of *existing* files, or even dozens of directories containing thousands of files. When it's not convenient to create the `txtar` archive manually, we can use `txtar-c` to do it automatically, and then use the standalone `testscript` runner to test the result.

Conclusion

To sum up, test scripts are a very Go-like solution to the problem of testing command-line tools: they can do a lot with a little, using a simple syntax that conceals some powerful machinery.

The real value of `testscript`, indeed, is that it strips away all the irrelevant boilerplate code needed to build binaries, execute them, read their output streams, compare the output with expectations, and so on.

Instead, it provides an elegant and concise notation for describing how our programs should behave under various different conditions. By relieving our brains of this unnecessary work, `testscript` sets us free to think about more advanced problems, such as: what are we really testing here?

10. Dependence day

> *The art of programming is the art of organizing complexity.*
> —Edsger W. Dijkstra, "Notes on Structured Programming"

In the previous two chapters we looked at some things that can make programs hard to test: unexported functions, user interaction, command-line interfaces, and so on. But probably the biggest challenge for test writers, especially when dealing with legacy systems, is code that has *dependencies*.

What do we mean by this? Simply, a software component that can't do its work in isolation. It needs something else in order to function properly: it *depends* on it. The dependency in question may be another software component, or it may be some external service or resource, such as a database or even the user's terminal.

This can be a problem for us when trying to test the component, because we need to satisfy all its dependencies just in order to *use* it in the first place, and that can be hard to do. Some dependencies are perfectly reasonable, and we just have to do our best to construct them and supply them in a test. For example, we might supply a buffer for a function to print to.

On the other hand, too *many* dependencies can be a design smell. If each component is designed to work together with a whole bunch of other components, and they all depend closely on one another, that makes the system difficult to understand or modify, as well as to test. We might even call this kind of problem *toxic codependency*. What can we do about it?

Just don't write untestable functions

Ideally, we wouldn't *write* code with lots of dependencies in the first place. As we've seen in other contexts, there's a deep synergy between testability and good design, and building components test-first is a good way to take advantage of that.

Simply following red-green-refactor doesn't mean that we'll *necessarily* produce a good design. Nothing can guarantee that. But it can help steer us away from some aspects of *bad* design: awkward APIs, pre-engineering, and over-engineering. And it can also help us avoid writing untestable functions.

> *When test-driving our code, it becomes very difficult to write complex code. First, because we write just enough code to satisfy the requirements, we discourage over-engineering and "big design up front".*
>
> *Second, whenever our code becomes a bit too complex and bloated, it also becomes difficult to test.*
> —Sandro Mancuso, "The Software Craftsman"

"Just don't write untestable functions" is great advice, for sure. But since we don't always have the luxury of developing a system guided by tests, we'll need some strategies for dealing with components that have already been written, and have some tricky dependency problems baked in.

One way to discover such components is through their effect on the tests. As we saw in an earlier chapter, it sometimes happens that certain tests become *brittle*, meaning that they tend to fail unexpectedly when we make minor changes to parts of the system.

Brittle tests are bad, but brittleness isn't always the fault of the tests. It can be a result of components having too many dependencies, particularly on other components.

> *Test brittleness is not just an attribute of how the tests are written; it's also related to the design of the system.*
>
> *If an object is difficult to decouple from its environment because it has many dependencies or its dependencies are hidden, its tests will fail when distant parts of the system change. It will be hard to judge the knock-on effects of altering the code.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

So the first and best way to tackle a testing problem caused by dependencies is to *change the design* of the system, if possible, so that the problem goes away.

For example, we may find that changing code in component A causes tests for component B to fail, and perhaps vice versa. This is a sign that A and B are tightly coupled: each depends closely on the behaviour of the other.

One response to a discovery like this might be to simply *acknowledge* the dependency, and merge the two components. In other words, we recognise that the tests are telling us these two things shouldn't be separate components: they should both be part of the *same* component. If they can't be used independently of each other, they're not really separate, even if the code structure makes it look that way.

Reduce the scope of the dependency

Merging two codependent components isn't always an option. There may be business reasons why they have to be kept separate, or perhaps it's just too big a refactoring to be practically possible at the moment. Or maybe the dependency is external to the system anyway, so we couldn't merge it even if we wanted to.

In that case, we can try to eliminate the dependency altogether, or at least reduce its scope. For example, suppose we have some method whose job is nominally to send an email in certain circumstances:

```
account.EmailUserIfAccountIsNearExpiry()
```

This kind of thing is a pig to test. How do we know it's behaving correctly? It sends email, or not, depending on whether the user's account is near expiry. That's an awkward thing to check from a test.

No doubt we could solve this problem if we really had to. We could set up a special email address on a dedicated domain, or a domain we already control, and we could write code to log in to an IMAP server and see if the email arrives, within some allowable time period.

That's a lot of machinery, and it's all very fragile and unreliable, and peppered with irrelevant points of failure (what if DNS isn't available? What if some intermediate mail relay decides to spamblock our emails?). We don't want to solve a problem like this: we want to *eliminate* it instead.

After all, what are we really testing here? The important thing about the method's behaviour is not that it actually sends email. It's that it *correctly decides that the user's account is near expiry*. Sending email is merely a consequence of that decision.

Once we've had this insight, it opens up a way for us to break this dependency, and split the behaviour into two separate components:

```
if account.IsNearExpiry() {
    account.EmailRenewalReminder()
}
```

Now we've changed a nasty "how do we test if this method sends email?" problem into a much nicer one: does the method return `true` or `false` given an account with a suitable expiry date?

To test that, we don't need an email server, or any kind of external infrastructure. We just need to set up two fake accounts, one with an imminent expiry date and one without, and check that the `IsNearExpiry` method returns `true` and `false` respectively.

We'll still need to test the actual email sending at some point. But it's actually easier to do that now that we've broken the coupling between the account expiry logic and the email sending code. Each of those components is easier to test independently, and since they *are* now independent, they *should* be separate components.

So we don't necessarily have to accept existing dependencies at face value. It's worth investigating a little and asking if the behaviours really *are* coupled for some important reason, or if the coupling is just a side-effect of poor design.

When we can improve both the testability *and* the design of the system by decoupling components, it absolutely makes sense to do that.

Be suspicious of dependency injection

A particularly bad way for a component to use dependencies is *implicitly*. For example, suppose we encounter some function like this:

```
CreateUserInDB(user)
```

We can infer that its job is to take some kind of user object or user ID, and create a record for the corresponding user in some database. But what's alarming about a function like this is that it *doesn't take the database*.

So how does it know which database to talk to? Presumably, that information must either be hard-wired into the function, or it must use some kind of global variable or global state to find out. Both of those alternatives are horrible, and what's worse is that they make it very difficult to test this function directly.

The problem is that the function has an *implicit dependency* on the database, whatever form that takes. It could be some global `*sql.DB` object floating around, initialised by some other part of the system, or it could be buried inside multiple levels of config structs. Or maybe the function itself creates a connection to a database server at some preset address.

Implicit dependencies like this are characteristic of code written without the guidance of tests. Because there's probably only one *real* database in production, the system just talks directly to it. It wasn't necessary for the author to provide a way to pass in alternate database connections, so they didn't.

One not very good solution to this problem is called *dependency injection*, which sounds complicated, but it just means "passing in dependencies as arguments". For example,

we could refactor the function to take an explicit database connection as a parameter:

```
CreateUserInDB(user, db)
```

This is *okay*, and it is an improvement, but it's really also just moving the furniture around, in a sense. We've made it possible to call this function from a test and pass it some `db` object that we construct for test purposes—but now we have to construct it!

Again, we could probably set up some kind of dedicated test database, or arrange to run a DB server locally for testing, and so on. We just don't *want* to.

And it's not really necessary, because the important thing about `CreateUserInDB` is not that it actually creates the user in a database. What's important is that, under the right circumstances, it *would* cause the user to be created.

That seems like a slightly finer distinction than in the "send email" example, but it's basically the same idea. Think about what could cause our test to fail. There are all sorts of *irrelevant* reasons that a user might not end up being created in the database, apart from a faulty `CreateUser` function.

The database server might not be running, or we might not be able to connect to it, or we might have the wrong credentials, or the wrong permissions, or the schema might be missing. Or there might be a bug in the database server itself (this has been known).

If it relies on making specific changes to a real database, our test will be brittle, meaning that it tests more than it should. Its job is only to test the code in `CreateUserInDB`, but it's also inadvertently testing our test environment, our network connection, and someone else's database product. Injecting the dependency doesn't fix that problem, because it's the *dependency* that's the problem.

Indeed, we should be a little suspicious of the very idea of dependency injection. Its purpose is to make it easier to write components that need lots of dependencies. But we don't *want* to make it easier to do that, because it's just a bad idea.

Avoid test-induced damage

Dependency injection can be habit-forming, and this can result in awkward API design choices. For example, suppose we're writing some function whose job is to make requests to some external service: weather information, perhaps.

Let's say it takes the user's location, queries some public weather API, and returns a brief forecast:

```
fc, err := weather.GetForecast(location)
```

Again, we wouldn't want to have to test this function. It has an implicit dependency on the *real API*, and that would make a test slow and flaky, because networks are slow and flaky. The API itself might occasionally be down or unresponsive, or just slow. It might cost us money to make real API calls, or cause the operators to complain about nuisance traffic generated by our tests.

The real issue here, though, as before, is that *we don't want to test their API*. It's not our code. If there *is* something wrong with it, we can't fix it in any case.

What we want to test is not only how our code behaves when *their* code is working, but also what happens when it *isn't*. And that's tricky: we can hardly ask them to take their API down so that we can test how our code reacts.

What can we do about this? Well, our first idea might be to inject the API's URL as an explicit dependency:

```
fc, err := weather.GetForecast(location, apiURL)
```

Now we can control what URL the function actually calls, so we could inject a fake version of the dependency. We could run a little local API server of our own, and pass in its URL from the test. A very simple one would do: it could just ignore the details of any request and always return hard-coded data.

But the price for this minor improvement is a little too high: we doubled the surface area of our *own* API, by requiring users to pass two parameters to `GetForecast` instead of one. That's 100% more paperwork.

Worse, the second parameter is only there for testing purposes, but real users will have to pass it too, because there are no optional arguments in Go. They'll always have to pass the same value: the URL of the real API.

```
fc, err := weather.GetForecast(location,
    "https://weather.example.com")
```

That's objectively worse, and many people would rightly regard such a change as *test-induced damage*. We don't want to smell up our API just to be able to inject the dependency for test purposes.

Again, we'd prefer to *eliminate* the dependency if we can, or at least *isolate* the part of the system that depends on it, so we can test the other parts more easily.

A test artefact that's intended to replace or simulate some system dependency, such as our simple local weather server, is known as a *fake*. As we'll see later in this chapter, there are many ways of faking dependencies. Let's first ask, though, if there's a way we can avoid having to use a fake in the first place.

"Chunk" behaviour into subcomponents

In the "send email" and "create user" examples, we didn't manage to eliminate the dependency altogether, but we did reduce its *scope* as much as possible. We refactored the code to extract any part of it that *doesn't* really need the dependency, and tested it in isolation. As a result, the tests were easier to write, and less brittle, and we made the system easier to understand.

What we've really been doing here is breaking down something that looks like a single

behaviour into smaller *chunks* of behaviour, which it turns out we can test independently.

How would this idea work with our weather example? It seems like `GetForecast` just calls the weather API, and that's it. What would the independent chunks be? Well, let's think about how we'd *implement* such a function. We might break it down into multiple steps like this:

1. Construct a suitable URL or HTTP request containing the user's location.
2. Send that request to the API.
3. Parse the API's response and format the data for delivery to the user.

Now we start to see some light, because we could *imagine* decoupling steps 1, 2, and 3. They are, in principle, independent behaviours, even though they'd usually be implemented together in a single function.

Let's start with step 1, constructing the request URL. How could we test this in isolation? We could use the "magic package" approach, which you may recall from previous chapters.

In other words, let's imagine we magically summon up some function: call it `FormatURL`, for instance. It takes the user's location and turns it into the URL we need to call.

We could test *that* function directly, without needing to make any network calls, by passing it a known input and comparing the result against what we expect. It's just a matter of string matching, which is something we're now pretty comfortable with.

Here's what that might look like:

```
func TestFormatURL_IncludesLocation(t *testing.T) {
    t.Parallel()
    location := "Nowhere"
    want := "https://weather.example.com/?q=Nowhere"
    got := weather.FormatURL(location)
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing `weather/1`)

Indeed, we could write a whole table test just about this, supplying different locations, very long strings, accented characters, and so on. If there's a bug in our URL formatting logic, it's actually *better* to have a test that targets that specific code.

What about step 3, parsing the API's response? Again, let's wave our wands (*Accio functionem!*) and magic something up. Let's call it `ParseResponse`, and write a test for it:

```
func TestParseResponse(t *testing.T) {
    t.Parallel()
    data := []byte(`{"weather":[{"main":"Clouds"}]}`)
    want := weather.Forecast{
        Summary: "Clouds",
    }
    got, err := weather.ParseResponse(data)
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing weather/1)

Again, this is straightforward, because the business of parsing the response data (we'll assume it's in JSON format) is in principle independent from actually *making* the API call.

All we need to do is construct a blob of JSON data that looks like what the API returns, and pass it to `ParseResponse` for processing. In fact, we could capture some data returned by the real API, and put it directly into our test.

With these two simple steps, we haven't *entirely* eliminated the dependency on the external API, but we have decoupled almost all of the important behaviour of our system from it, and tested it with some simple, isolated tests. It shouldn't be too hard to go ahead and write `FormatURL` and `ParseResponse` now, guided by these tests.

Reassemble the tested chunks

We can now rewrite `GetForecast` to use our two new, and independently-tested, "chunk" functions:

```
func GetForecast(location string) (Forecast, error) {
    URL := FormatURL(location)
    ... // make API request
    return ParseResponse(resp.Body)
}
```

The (omitted) code to make the API request and get the response is essentially standard practice, and just consists of calling `http.Get`, checking the response status, and so on. There isn't much to get wrong here, and we can probably test it adequately by run-

ning the program manually. After all, a working program is the real goal here, not just a bunch of tests.

> *The goal is not to write tests. The goal is to write correct software, and it's up to us to decide how much support we need to know that we're doing the right thing.*
> —Michael Feathers, "Testing Patience"

The really critical behaviours of the system, though—that is, the bits we're most likely to get wrong—are constructing the URL and parsing the response data. And we can test those as thoroughly as we need to, without the disadvantages of slow, flaky, or brittle tests.

By the way, a similar example of testing an API client is explored in much more detail in my companion book The Power of Go: Tools, and in a blog post titled An API client in Go, which you might find useful if you're working on a similar program.

In general, then, a good way to test behaviour which can't easily be decoupled from its dependencies is to *chunk* it: break it down into smaller, individually testable functions, most of which don't need the dependency, and each of which we can test in isolation from the others.

Then all we need to ensure is that we connect these chunks back together in the right way in the real program, which usually isn't too hard to test.

Extract and isolate the key logic

Sometimes a fairly simple behaviour is nevertheless wired tightly into a component with a complex API, making it awkward to test. For example, I recently consulted on a project involving a Go API server with dozens of complicated endpoints, each requiring a lot of fiddly paperwork to call.

Worse, it was very difficult to isolate the endpoint handlers themselves for testing, since they had lots of complex side effects on the world, including calling other services.

But the behaviour we actually wanted to *test* was very simple: that the system invokes the appropriate handler for the given type of request. The routing logic looked something like this:

```
switch requestType(req) {
case TypeA:
    HandleTypeA(req)
case TypeB:
    HandleTypeB(req)
case TypeC:
    HandleTypeC(req)
}
```

The most natural way to test something like this would be to spin up a local instance of the server and *make* requests of various types to it, checking that the appropriate handler gets invoked. But, as I say, this was far from straightforward in practice.

First, just to create a valid request in the first place required many screenfuls of code, different for each type of request. Second, the handlers had lots of unwanted side effects that would be difficult to suppress using fakes. Third, since the actual response was more or less the same for all requests, it was quite hard to tell in a test which handler had actually been called.

After some head-scratching, we decided to ask ourselves the ever-useful question "What are we really *testing* here?" The answer was "We're testing that requests are passed to the correct handler for their type."

When you frame the behaviour in these terms, it's suddenly a lot easier to figure out how to test it. Suppose, instead of using a `switch` statement to dispatch requests, the router used a map:

```
handler := map[RequestType]Handler{
    TypeA: HandleTypeA,
    TypeB: HandleTypeB,
    TypeC: HandleTypeC,
}
```

Now all the dispatching code needs to do is look up the right handler in the map for the request type, and call it:

```
handle := handler[requestType(req)]
handle(req)
```

We don't need to test map lookups or function calls; as long as the map data is correct, the dispatching will work. So now all we need to test is the map data. The end-to-end *behaviour* of the API router is very difficult to test, but the routing *logic* is very easy to test, provided we extract it and isolate it in the right way.

Isolate by using an adapter

No program is an island, and we often have to communicate with *other* programs in order to get our work done. For example, we might use some external database such as PostgreSQL, an internet API such as the weather service, or some executable binary that we run using the `os/exec` package.

Any external dependency of this kind presents both a design problem and a testing problem. Sometimes we can solve both problems at once, by using the *adapter pattern*.

An adapter is a way of grouping together all the code in our system that deals with a particular dependency. For example, the `GetForecast` function that we explored in

the previous section is a kind of adapter. In this case, it's an adapter that lets us talk to a specific weather service API.

Adapters are also sometimes called *ambassadors*: their job is to "represent" us to the external system, and vice versa. They deliver vital messages to the foreign embassy, translating them into the appropriate "language" so that they'll be understood. In turn, they translate and bring back to us the response in a language *we* can understand.

In the weather example, the "outbound" part of the adapter is the `FormatURL` function. It translates the query that we want to make into the weather API's own language: that is, a URL containing the user's location. We then *deliver* this message in the `GetForecast` function, using `http.Get`.

The "inbound" part of the adapter, `ParseResponse`, is responsible for handling the API's response in some arbitrary JSON format, let's say. It translates the raw response into the native `Forecast` struct type that we use internally to represent weather data.

The net effect of this adapter is to decouple all knowledge about the specific weather API from the rest of the system. We can treat it as an obliging ambassador that will ask questions of some foreign power on our behalf, and bring us back the results in a conveniently-shaped diplomatic bag.

Encapsulating all dependency-specific knowledge in a single component, then, solves both our design problem and our testability problem. It means that we don't need to *call* the remote API in our tests, and in turn the status of our tests doesn't depend on whether some external service is available.

Example: a database adapter

Let's see how the adapter pattern might work with a dependency like some SQL database, for example. Suppose we need to store product information for Acme Widgets, Inc, and we'd like to access it using the classic *CRUD* methods: Create, Read, Update, and Delete.

So let's say we define some `Widget` struct:

```
type Widget struct {
    ID   string
    Name string
}
```

(Listing `widget/1`)

Our first attempt at a constructor for `Widget` might look something like this, with the gory details omitted:

```
func Create(db *sql.DB, w Widget) (ID string, err error) {
    // SQL: create widgets table if it doesn't exist
```

```
    // SQL: insert into widgets table
    // handle possible error
    return w.ID, nil
}
```

We take some `*sql.DB` object representing a database handle, instantiated using some specific driver (for example, Postgres). We'll use that to execute the necessary SQL queries (omitted here) to add the specified new widget to the database.

This is fine, of course, and most Go applications that use databases look something like this. But it's a *little* awkward, in a couple of important ways. First, knowledge about the specific database server (for example, Postgres) is embedded in a function that should really only contain *business logic*. That is, code that implements rules about widgets for our specific customer or problem domain.

We don't want this key business logic all tangled up with the code to construct SQL queries for a specific database server. That's just bad design, because it violates the *Single Responsibility Principle*, that any given function should do more or less one thing. We'd have to copy and paste the same logic to any other function that stores widgets in a different way.

The more serious problem is that now it's impossible to *test* our widget logic without having an external database available, and making real queries against it. Even if this is only some local test server, it's still annoying. We can't just run `go test`: we have to use a Makefile or Docker Compose file or something to start the Postgres server first.

Actually, it *is* possible to start external services automatically in a Go test, either by running commands via `os/exec`, or by starting containers using a package such as `testcontainers`. That's a valid approach, but a bit heavyweight: it's sumo, not judo.

The adapter pattern gives us a more elegant way to design this problem out of existence. How would that work? Well, the underlying issue is that the widget logic is uncomfortably tightly coupled with the "storing things in Postgres" code. Let's start by breaking that dependency.

It's presumably not crucial to widgets that they be stored in *Postgres*, specifically. So let's invent some completely abstract *widget store*, described by an interface:

```
type Store interface {
    Store(Widget) (string, error)
}
```

(Listing `widget/1`)

We can implement this interface using any storage technology we choose. All we need to do is provide a suitable `Store` method, and make it work.

Now we can change `Create` to take an abstract `Store`, instead of something specific like a `*sql.DB`:

```
func Create(s Store, w Widget) (ID string, err error) {
    ID, err = s.Store(w)
    if err != nil {
        return "", err
    }
    return ID, nil
}
```

(Listing widget/1)

In a real application, `Create` would probably do some widget-related business logic (validation, for example), which we can imagine wanting to test in isolation.

To do that, we still need something that implements `Store`, for test purposes. But this can be as trivial as we like. In fact, we could use a Go map. The data won't be persistent, but that doesn't matter; it only needs to persist for the duration of the test.

```
type mapStore struct {
    m    *sync.Mutex
    data map[string]widget.Widget
}

func newMapStore() *mapStore {
    return &mapStore{
        m:    new(sync.Mutex),
        data: map[string]widget.Widget{},
    }
}

func (ms *mapStore) Store(w widget.Widget) (string, error) {
    ms.m.Lock()
    defer ms.m.Unlock()
    ms.data[w.ID] = w
    return w.ID, nil
}
```

(Listing widget/1)

Even though this is only a test fixture, we'd still like it to be concurrency safe, so that a `mapStore` *could* be shared between parallel tests if necessary. The protective mutex makes this possible, as we saw with the `kvstore` example in a previous chapter.

Great. With that preparatory work out of the way, we can go ahead and write a test for `Create`:

```
func TestCreate_GivesNoErrorForValidWidget(t *testing.T) {
    s := newMapStore()
    w := widget.Widget{
        ID: "test widget",
    }
    wantID := "test widget"
    gotID, err := widget.Create(s, w)
    if err != nil {
        t.Errorf("unexpected error: %v", err)
    }
    if wantID != gotID {
        t.Error(cmp.Diff(wantID, gotID))
    }
}
```

(Listing widget/1)

We can run *this* test without any awkward external dependencies, such as a Postgres server. That makes our test suite faster and easier to run, and by decoupling the widget logic from the storage logic, we've also improved the overall architecture of our package.

> *Every time you encounter a testability problem, there is an underlying design problem. If your code is not testable, then it is not a good design.*
> —Michael Feathers, "The Deep Synergy between Testability and Good Design"

In the real program, though, we'll probably want to store widget data in something like Postgres. So we'll also need an implementation of `Store` that uses Postgres as the underlying storage technology.

Suppose we write something like this, for example:

```
type PostgresStore struct {
    db *sql.DB
}

func (p *PostgresStore) Store(w Widget) (ID string, err error) {
    // horrible SQL goes here
    // handle errors, etc
    return ID, nil
}
```

(Listing widget/1)

This is an equally valid implementation of the `Store` interface, because it provides a `Store` method. The only major difference from `mapStore` is that there are about 1.3 million lines of code behind it, because it talks to Postgres. Thank goodness we don't have to test all that code just to know that `Create` works!

However, we *do* also need to know that our `PostgresStore` works. How can we test it? We could point it to a real Postgres server, but that just puts us right back where we started. Could we use *chunking* to avoid this?

Recall that in the `weather` example, we split up the adapter's behaviour into inbound and outbound chunks. "Outbound", in the `PostgresStore` case, would mean that, given a widget, the adapter generates the correct SQL query to insert it into the database. That's fairly easy to test, because it's just string matching.

What about the "inbound" side? Well, there's no inbound side in our example, because it's deliberately oversimplified, but we can imagine that in a real application we'd also need to *retrieve* widgets from the `Store`, and so we'd need to add a `Retrieve` method to the interface. Let's briefly talk about what that would involve, and how to test it.

In the Postgres case, implementing `Retrieve` would mean doing a SQL query to get the required data, and then translating the resulting `sql.Row` object, if any, to our `Widget` type.

This is awkward to test using a real database, as we've seen, but it's also pretty difficult to fake a `sql.DB`. Fortunately, we don't have to, because the `sqlmock` package does exactly this useful job.

We can use `sqlmock` to construct a very lightweight DB that does nothing but respond to any query with a fixed response. After all, we don't need to test that *Postgres* works; let's hope it does. All we need to test on *our* side is that if we get a row object containing some specified data, we can correctly translate it into a `Widget`.

Let's write a helper function to construct a `PostgresStore` using this fake DB:

```
import "github.com/DATA-DOG/go-sqlmock"

func fakePostgresStore(t *testing.T) widget.PostgresStore {
    db, mock, err := sqlmock.New()
    if err != nil {
        t.Fatal(err)
    }
    t.Cleanup(func() {
        db.Close()
    })
    query := "SELECT id, name FROM widgets"
    rows := sqlmock.NewRows([]string{"id", "name"}).
        AddRow("widget01", "Acme Giant Rubber Band")
```

```
    mock.ExpectQuery(query).WillReturnRows(rows)
    return widget.PostgresStore{
        DB: db,
    }
}
```

(Listing widget/1)

Now we can use this fake `PostgresStore` in a test. We'll call its `Retrieve` method and check that we get back the `Widget` described by our canned test data:

```
func TestPostgresStore_Retrieve(t *testing.T) {
    t.Parallel()
    ps := fakePostgresStore(t)
    want := widget.Widget{
        ID:   "widget01",
        Name: "Acme Giant Rubber Band",
    }
    got, err := ps.Retrieve("widget01")
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing widget/1)

Very neat! Finally, let's write the `Retrieve` method and check that it passes our test.

```
func (ps *PostgresStore) Retrieve(ID string) (Widget, error) {
    w := Widget{}
    ctx := context.Background()
    row := ps.DB.QueryRowContext(ctx,
        "SELECT id, name FROM widgets WHERE id = ?", ID)
    err := row.Scan(&w.ID, &w.Name)
    if err != nil {
        return Widget{}, err
    }
    return w, nil
}
```

(Listing `widget/1`)

And we *still* didn't need a real Postgres server, or any other external dependencies. Of course, our confidence in the correctness of the code only goes as far as our confidence that our SQL query is right, and it might not be.

Similarly, the canned row data returned by our fake might not match that returned by a real server. So at *some* point we'll need to test the program against a real server. The work we've done here, though, has greatly reduced the scope of our dependency on that server. We might only need it for one or two tests, run occasionally as a backstop (before a deployment, for example).

Fakes, mocks, stubs, doubles, and spies

The fake database in the `PostgresStore` example is an example of what's sometimes called a *test fake*, or *test double*. You'll find lots of colourful and mysterious terminology on this subject, but I'm not sure how helpful all these different words are. I tend to lump them all under the general heading of "fakes", and leave it at that.

By "fake" I mean any value that we construct in a test, purely for the purposes of testing, that in the *real* program would usually represent some external dependency, or maybe just some expensive operation that we want to avoid in tests.

It's worth taking a moment to outline the different kinds of things a fake can do, and how they relate to some of the testing terms you may come across.

The minimal fake does nothing at all, or perhaps just returns fixed responses to method calls; this is sometimes known as a *stub*. It exists purely to make the program compile, since something is required syntactically, but doesn't attempt to implement the behaviour under test.

What's wrong with stubs? They're a design smell. If the component under test is so tightly coupled to another component that it can't work *without* a stub version of it, then it's probably too tightly coupled. Also, if it *can* work with a stub version that does nothing, then apparently it isn't such an important dependency after all! We can usually change the design to remove the coupling.

A slightly more useful fake is something like our memory-based `MapStore`, which isn't actually "fake", even though we call it that. It's a real *store*, just not a persistent one. If the behaviour under test doesn't need persistent storage, then it's faster to test it using an in-memory version such as `MapStore`.

It's often useful in testing to have some kind of fake that can *record* how it's used. For example, in a previous chapter we used a `*bytes.Buffer` in a test as a "fake" `io.Writer`, so that the test can check what gets written to the buffer. This kind of fake is sometimes called a *spy*.

Don't be tempted to write mocks

We can use the information provided by a spy to fail the test, but you could also imagine a fake sophisticated enough to fail the test *itself* if the system doesn't call it correctly. This is known as a *mock*.

The term "mock" is often loosely used to mean any kind of test fake, but strictly speaking a mock is a fake that has pre-programmed *expectations* about how it's to be used, and *verifies* those expectations when used in a test.

For example, suppose we're building a `BuyWidget` function, and we need some kind of fake payment processing service, so that we don't have to make real payments in the test. We could use a stub for this, but then we wouldn't know whether or not the system actually *called* the service to make a "payment". That's an important part of the behaviour under test, so what could we do?

We could make a *spy* payment service that simply records the details of any payments that the system uses it to make, and then have the test check the spy afterwards. But another option would be to build a *mock* payment service, that takes the test's `*testing.T` as a parameter.

The mock's "make payment" method can be programmed to expect the exact payment details the system should provide, and fail the test if they don't match exactly. It could also fail the test if the "make payment" method doesn't get called at all.

Mocks can become very complex and sophisticated in their behaviour, sometimes almost as complex as the real component they're replacing. And that's a bad thing, at least if we want to build simple, reliable tests. How do you know that your mock accurately replicates the behaviour of the real system? If it doesn't, you're testing the wrong thing.

And there's another problem with mocking in tests. Because a mock is programmed to expect a specific sequence of calls and parameters, it's very tightly coupled to the way the system uses the real component. It observes the *indirect* outputs of the system, not its direct outputs. We're no longer testing just the user-visible behaviour of the system, we're testing its *implementation*, which makes our tests brittle.

If we refactored the system, changing its implementation without changing any of its public behaviour, we wouldn't want this to break any tests. But it would almost certainly break any tests that use mocks in this way. As we've seen in previous chapters, if your tests prove brittle, that usually points to a design problem that you can fix by decoupling components.

In other words, mocking *presupposes* your design, rather than letting the right design gradually emerge from the desired behaviour of the system. It's also a lot of work building all that mock code, and keeping it up to the date with the behaviour of the real component over the life of the project.

It's not that mocking is a bad idea in itself; mocks can be useful sometimes. It's just that if you find you can't test your system adequately *without* mocks, this is probably

a hint that you should redesign the system instead. And building a mock certainly shouldn't be our default approach. We should only resort to it once all the other options described in this chapter have been exhausted.

Because of the way Go's type system works, a mock object almost always requires an *interface*, so that we can inject one concrete type (the mock) where another (the real dependency) would normally go. Again, interfaces aren't a bad thing in themselves. The `Store` interface is useful, because it lets us choose between multiple *real* implementations of it.

Creating an interface *purely* to permit mocking, though, is test-induced damage: in fact, it's interface pollution.

A quick hypocrisy check: didn't we just use the `sqlmock` package earlier in this chapter? Yes. Technically, we used it as a *stub*, not a mock, and we didn't *write* it, we just imported it. But you have a point. If Go's `sql.DB` type were easier to fake without creating test-induced damage, we'd certainly want to avoid things like `sqlmock`. However, it feels like a lesser evil than spinning up an external database for tests that don't really need one.

What about "mock" struct types like `errReader`, though? You may recall from the chapter on testing error behaviour that we built a little implementation of `io.Reader` that simply always returns an error when you try to read it.

This is fine: it isn't technically a mock, because it doesn't fail the test by itself. But it is a kind of fake: specifically, a fake that's designed to work incorrectly. We may call it instead, perhaps, a *crock*.

Turn time into data

Remember the example of a function with an implicit dependency on a global database connection? That's a particularly awkward kind of dependency, because it's a *singleton*: the system is designed under the assumption that there will only ever be one database.

A singleton like this is bad design for all sorts of reasons, not least because it makes it difficult to substitute a fake for testing. We first had to make the dependency *explicit*, to create a *seam* in the system where a test fake could be injected. That also removes the singleton problem, because the database now just becomes a parameter to the part of the system that actually depends on it.

Another implicit singleton dependency that can make programs hard to test is slightly less obvious: it's *time*. Any program that calls `time.Now`, or some equivalent, by definition behaves differently at different times of day. Thus, the results of the tests will also depend on the time of day they are run (or the day of the week, day of the month, or even the year).

Consider a test like this, for instance, which is similar to one I came across in a real project:

```
func TestOneHourAgo(t *testing.T) {
    t.Parallel()
    now := time.Now()
    then := past.OneHourAgo()
    if now.Hour() == 0 {
        assert.Equal(t, 23, then.Hour())
        assert.Equal(t, now.Day()-1, then.Day())
    } else {
        assert.Equal(t, now.Hour()-1, then.Hour())
        assert.Equal(t, now.Day(), then.Day())
    }
    assert.Equal(t, now.Month(), then.Month())
    assert.Equal(t, now.Year(), then.Year())
}
```

(Listing past/1)

Can you see what `past.OneHourAgo` is meant to do? That's right: it returns a `time.Time` value representing the instant that is one hour before now. A simple-minded way to test this would be to check that the result is exactly the same as `time.Now`, except that the `Hour` part is one less.

And that test would pass most of the time, assuming that `OneHourAgo` behaves correctly. But if we were to run it during the hour between midnight and 1am, it would fail, because the result of `OneHourAgo` would actually be some time after 11pm the *previous* day.

Hence the special case we see in this test, when `now.Hour() == 0`. The programmer correctly identified the problem: the test is flaky because it depends on the actual time of day it's run. However, they opted for the wrong solution: patch the test to have different expectations at different times of day.

So what's the *right* solution here? It must be one that eliminates the flakiness: that is, it makes the result of the test independent of the time of day. To do that, we have to break the function's implicit dependency on the current time.

To look at it another way, our confidence in the correctness of `OneHourAgo` will be much reduced if we know that this test never runs at midnight. It's fun when games like NetHack behave differently at midnight: it's not so fun when our tests do.

> *On a full moon, [NetHack] players start with a +1 to Luck, meaning that random effects like fountain drinking tend to be slightly better and the player hits more often in combat. On Friday the 13th, players start out with a base Luck of -1, meaning that prayer never works. Undead creatures do double damage during the hour of midnight, it's harder to tame dogs on full-moon nights, and gremlins sometimes steal from the player between 10pm and 6am.*
> —John Harris, "Exploring Roguelike Games"

To avoid attracting the attention of gremlins, what should we do? One option would be to make the dependency explicit, as with the database example. In other words, instead of expecting `OneHourAgo` to find out the current time itself, we could just *tell* it the time:

```
now := time.Now()
then := past.OneHourAgo(now)
```

This is a start, and it does make the function more useful, since it can now calculate the instant an hour before *any* time, not just the current time. But it doesn't make the test less flaky, because unless we can ensure that the test is regularly run just after midnight, we still can't be confident that the function handles this situation correctly.

But the point of this change, of course, is so that in a *test*, we can pass it some *specific* time. While we're at it, let's get rid of these unwanted `asserts`. We can now replace them with a direct comparison, because we know *exactly* what time to expect:

```
func TestOneHourAgo(t *testing.T) {
    t.Parallel()
    testTime, err := time.Parse(time.RFC3339, "2022-08-05T00:50:25Z")
    if err != nil {
        t.Fatal(err)
    }
    want, err := time.Parse(time.RFC3339, "2022-08-04T23:50:25Z")
    if err != nil {
        t.Fatal(err)
    }
    got := past.OneHourAgo(testTime)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing past/2)

This works, and now we can explicitly test midnight handling, as well as some other times of day. In fact, this is a good candidate for a table test.

But we've created a little bit of test-induced damage as a result of this change. Now everyone who wants to use this function has to construct a `time.Time` value to pass it, even though that will almost always be simply the result of `time.Now()`.

We also have to refactor all calls to `OneHourAgo` throughout the system to take this argument, which is annoying. So turning a singleton into an explicit parameter is not always the best answer. What could we do instead?

Well, the real problem here is that the function calls `time.Now` to get the current time,

and we can't change the result of that call from a *test*. Could we create some kind of a *mock clock*, inject this as a dependency to the system as a whole, and use that to control the behaviour of time-related functions? Or even intercept and hijack time-related calls using some kind of fake time library?

Sure. But maybe that isn't necessary. All we need to fake is a function, so why don't we *use* a function? For example, supposing there were some global variable `past.Now`, whose type is `func() time.Time`, and whose default value is the `time.Now` function itself:

```
// careful: not 'time.Now()'
var Now = time.Now
```

(Listing past/3)

Note that we're not *calling* `time.Now` here. We're assigning that *function itself* to a variable, `Now`.

We can replace all calls to `time.Now()` throughout the system with calls to `Now()`, without changing its behaviour in any way:

```
func OneHourAgo() time.Time {
    return Now().Add(-time.Hour)
}
```

(Listing past/3)

But we've created a seam where we can inject our *own* "now" function:

```
func TestOneHourAgo(t *testing.T) {
    testTime, err := time.Parse(time.RFC3339, "2022-08-05T00:50:25Z")
    if err != nil {
        t.Fatal(err)
    }
    past.Now = func() time.Time {
        return testTime
    }
    want, err := time.Parse(time.RFC3339, "2022-08-04T23:50:25Z")
    if err != nil {
        t.Fatal(err)
    }
    got := past.OneHourAgo()
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing past/3)

A *fake func* like this is much simpler than a mock object, doesn't need a useless single-method interface, and doesn't create extra paperwork for users. Indeed, it's completely transparent to users, and that's partly the point.

It's up to you how you prefer to construct the `testTime`. A standard date format such as `RFC3339` is ideal, but if you find it clearer to express the relevant instants using a different layout, by all means do so:

```
const layout = "Jan 2, 2006 at 3:04pm (MST)"
testTime, err := time.Parse(layout, "Feb 3, 2013 at 12:54am (PST)")
...
```

One final point to bear in mind is that a test like this can't be run in parallel, because we're modifying a global variable. Imagine a whole bunch of concurrent tests, each trying to set `past.Now` to its own fake func, all at once. Chaos ensues, so we need to make this test sequential, just like any other test that needs to fake out a singleton.

The basic trick here is to *turn time into data*. Instead of having the system magically (and invisibly) be dependent on some singleton clock, make time values just another piece of data that the system accepts as input.

Because Go provides a `time.Time` data type specifically for representing instants of time as data, this makes our lives somewhat easier, but there's one thing we should be careful to remember: despite what the birthday cards say, *time is not just a number*.

We may have some sense that time is represented internally in Go as a number (the number of nanoseconds since *epoch*, which is just some fixed reference time that everybody agrees on). But it's also more complicated than that.

> *People assume that time is a strict progression of cause to effect, but actually, from a non-linear, non-subjective viewpoint, it's more like a big ball of wibbly-wobbly... timey-wimey... stuff.*
> —The Tenth Doctor, "Blink"

In fact, there isn't just one system clock on your computer. There are two: a *wall clock*, which gives the current time, and a *monotonic clock*. The point about the wall clock is that it can stop, change, or even go backwards (during a DST change or leap second, for example). So for *calculations* about time, the Go `time.Time` type includes the monotonic clock reading too. This, as the name suggests, goes only in one direction: forwards.

The implication of this is that we mustn't treat times, either in tests or in system code, as simple numbers that we can do math on, or even compare directly with the == operator.

Instead, we should use the API provided by the `time` package for safely manipulating times:

```
func TestOneHourAgo(t *testing.T) {
    t.Parallel()
    now := time.Now()
    want := now.Add(-time.Hour)
    got := past.OneHourAgo()
    delta := want.Sub(got).Abs()
    if delta > 10*time.Microsecond {
        t.Errorf("want %v, got %v", want, got)
    }
}
```

(Listing past/4)

We can compare `time.Time` values safely using their `Equal` method, but that wouldn't work here. The `got` time won't be *exactly* the same instant as `want`, even if our math is correct: it takes a non-zero time to do that computation!

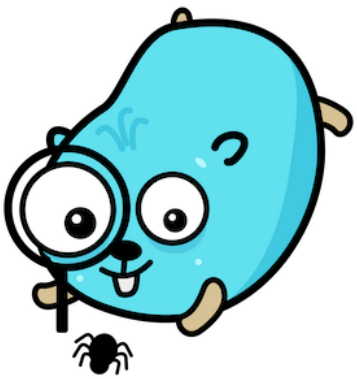
Instead, a better idea is to find the difference between the two times, using the `Sub` method to produce a `time.Duration`, and then checking that the difference is small enough to satisfy our requirements.

By the way, it's now abundantly clear from this test that we don't even need to write a `OneHourAgo` function at all. The best kind of test, indeed, is one that shows us we don't need the function… or the test.

In the next and final chapter, we'll turn our attention from the particular to the general: what makes a test *suite* and its associated codebase good? Or, more interestingly, bad? Let's find out.

11. Suite smells

> *A friend had just started a consulting gig with a new team, so I asked him, "How are they doing?" He said, "They're writing legacy code, man."*
> —Michael Feathers, "Working Effectively with Legacy Code"



We've talked a lot in the book so far about specific techniques for writing individual tests, especially in difficult circumstances. In this final chapter we'll turn our magnifying glass on to the test *suite*: that is, the whole collection of tests for a particular project or system, considered *as* a whole.

We'll be hunting for certain problems which are known to lurk in the darker corners of test suites, and looking at some ways to address them. How should we assess the overall quality of tests for a given project? How can we go about extending and improving them?

No tests

Let's start with a problem that affects a good many projects, even today: a complete lack of tests. If you're relatively new to the industry, this might surprise you: "Doesn't everyone write tests? Isn't it considered best practice?"

Sure. But best practice is what a company *says* it does. What *actually* goes on is another matter. In the startup phase, speed is more important than maintainability. Assuming

the product makes it to market, the business focus will naturally be on rapid growth, rather than building solid foundations for the long term.

Eventually, though, the cost of adding anything to an untested codebase becomes unmanageable. Indeed, the cost of touching the code *at all* becomes so high that everyone starts finding excuses not to do it. What can we do?

The answer is pretty simple: write some tests! You don't need to get permission. Just go ahead and do it. If you're tasked with touching some small piece of the code that isn't tested, add a test, then complete your task.

Sometimes people feel nervous about doing this, because they haven't obtained everybody's buy-in, perhaps from a company-wide conversation about whether or not testing is a good idea. Don't worry about this. It *is* a good idea. Just go do it.

> *You don't need to convince the customer (or your manager) that code quality is important. You just need to convince yourself.*
> —Robert C. Martin

It's hard to imagine a sensible engineering manager getting upset about someone writing tests, but I suppose it's possible. One of this book's reviewers warned me that this "guerilla testing" idea could get someone fired. Well, okay; don't go *crazy*, but I don't think anyone should need to get permission to do their job well.

> *Every organization, team, and project is different, and provocative as it may sound, that shouldn't affect how the developers work. At the end of the day, it's you who'll make changes to the software and fix the bugs, irrespective of the quality assurance process. Therefore, it's in your interest that the software be both testable and tested.*
> —Alexander Tarlinder, "Developer Testing: Building Quality Into Software"

If you *do* get fired as a result of applying the advice in this book, that's probably for the best, all things considered. But if it happens, please get in touch, and I'll make it my personal mission to get you a job with a better company: one where people are rewarded, not punished, for producing software that actually works.

> *The development teams at many companies are focused on high-level business goals, lack any direct incentive to improve code quality, and perceive an investment in code quality to be at odds with shipping on-time... No one is paying, rewarding, or pressuring them to maintain a high level of code quality.*
> —Mike Bland, "Goto Fail, Heartbleed, and Unit Testing Culture"

Not only do you not need permission for writing tests, you also don't need to convince your co-workers to do it. Testing isn't a religion, and it doesn't need converts. Write good software the way *you* do it, and let others do it *their* way.

If your way is better, people will eventually come around to it. You don't need to nag them, proselytise them, or lecture them about testing. Don't be the person who fights for an idea: be the person who has ideas worth fighting for.

> *There's no room for testing religions; do what works.*

> —Tim Bray, "Testing in the Twenties"

Most developers, once they've experienced the joy and the lightness of heart that comes from working on well-tested code, become enthusiastic advocates for it themselves. No fighting needed.

> *Once you've worked on a system with extensive automated tests, I don't think you'll want to go back to working without them. You get this incredible sense of freedom to change the code, refactor, and keep making frequent releases to end users.*
> —Emily Bache, "The Coding Dojo Handbook"

Legacy code

Adding tests to a codebase that doesn't have them can be a daunting prospect, but let's take it one step at a time. You only need to test what you touch.

Suppose you have some monster function, and you need to make a change to line 1337 of it. The prospect of writing a test for the *entire function* fills you with horror and sadness (I've been there). Instead, use the divide-and-conquer technique.

See if a small block of lines around where you're working can be extracted into its own little function. If so, do this, add a little test for the little function, and make your change, leaving the rest of the code alone. Repeat this process every time you touch some part of the system.

> *Over time, tested areas of the code base surface like islands rising out of the ocean. Work in these islands becomes much easier. Over time, the islands become large landmasses. Eventually, you'll be able to work in continents of test-covered code.*
> —Michael Feathers, "Working Effectively with Legacy Code"

The nice thing about this strategy is that any effort you put into test writing will automatically end up right where it's needed: those parts of the system where you're regularly making changes. Of course tests for *everything* would be great, but we have to start somewhere, and right now, that's here.

> *If you have a big system, then the parts that you touch all the time should be absolutely rock solid, so you can make daily changes confidently. As you drift out to the periphery of the system, to parts that don't change often, the tests can be spottier and the design uglier without interfering with your confidence.*
> —Kent Beck, "Test-Driven Development by Example"

But what if you can't add a test for something because the function is untestable? You could refactor it to be testable, but that's not safe, because you don't have a test. It's Catch-22.

This dilemma is characteristic of legacy systems, otherwise known as "here be dragons" code:

> *Legacy code is code without tests, but more importantly, it's code that isn't designed with testability in mind. Such code tends to be riddled with temporal coupling and indirect input. It houses a multitude of monster methods, and its components are intertwined in a bizarre web of dependencies that have spun out of control. Adding any kind of test to such code may be no small feat.*
> —Alexander Tarlinder, "Developer Testing: Building Quality Into Software"

Making changes to a system like this is risky, but making them without the support of tests is even more dangerous:

> *Changes in a system can be made in two primary ways. I like to call them "Edit and Pray" and "Cover and Modify". Unfortunately, "Edit and Pray" is pretty much the industry standard.*
> —Michael Feathers, "Working Effectively with Legacy Code"

Can we do better? Here's one emergency rescue procedure for untestable functions that I've used successfully. Ignore the existing function and write a test for the function you'd *like* to have. Then make that test pass by writing a new function from scratch. Finally, delete the old function.

Legacy systems aren't much fun to deal with, but on the plus side, they usually work pretty well. That's how they got to *be* legacy systems. Whatever bugs remain are at least well understood, and people know how to work around them.

We usually don't need to make many changes to legacy systems, and when we do, a combination of testing and refactoring can make it safe. The important thing to remember about legacy systems is to avoid writing more of them.

> *The system turns to spaghetti really, really fast. Complexity isn't one mistake you make, it's hundreds or thousands of mistakes made by many people over a period of time. You don't notice it happening, and once it's happened, it's almost impossible to fix. It's so overwhelming, you never get to it.*
> —John Ousterhout, "A Philosophy of Software Design"

Insufficient tests

Some systems *have* tests, but not enough of them, or they're not very good. What can we do in this case? Apart from steadily growing the test suite by applying the "test what you touch" strategy outlined in the previous section, can we do anything to fix the underlying *causes* of inadequate testing?

For example, maybe a developer is just in a big hurry, and skips writing a test because they feel it will slow them down. Sure, but sometimes steering is more important than speed. If the program doesn't *work*, does it really matter how fast you can write it?

> *You don't save time if you save on quality. Because it comes back. It's like when someone asks me "I need to lose a couple of pounds, do you have a tip for me?" And I say, "Yeah. Donate blood."*
> —Klaus Leopold, "Blockers: Treasures of Improvement"

Of course, if someone is new to writing tests, it will be a slow process at first, because they don't know what to do. Once you have a little experience, testing becomes much easier and faster. Half the battle is just having the confidence to know that you *can* test anything, if you put your mind to it.

A good way to build this confidence is to agree with your team that for a short trial period, you'll write tests for everything you do. If anyone gets stuck on how to test something, the whole team will pitch in to help.

> *When I work with teams, I often start by asking them to take part in an experiment. For an iteration, we try to make no change to the code without having tests that cover the change. If anyone thinks that they can't write a test, they have to call a quick meeting in which they ask the group whether it is possible to write the test.*
>
> *The beginnings of those iterations are terrible. People feel that they aren't getting all the work done that they need to. But slowly, they start to discover that they are revisiting better code. Their changes are getting easier, and they know in their gut that this is what it takes to move forward in a better way.*
> —Michael Feathers, "Working Effectively with Legacy Code"

Sometimes tests end up being insufficient or ineffective because they're written *last*. You might have noticed that, throughout this book, we've seen many important advantages to writing tests *first*. The most persuasive of those advantages for me is that it just makes my work easier.

However, some people don't feel this way, and that's okay. Instead, they find it easier to implement the behaviour first, and then figure out how to test it. If the result is a correct program with robust tests, then who cares what order those things are written in?

> *For some people, [writing tests first] works fine. More power to 'em. It almost never works for me in a pure form, because my coding style tends to be chaotic in the early stages, I keep refactoring and refactoring the functions all the time.*
>
> *There's room for argument here, none for dogma. If your engineers are producing code with effective tests, don't be giving them any static about how it got that way.*
> —Tim Bray, "Testing in the Twenties"

It's always tempting when you find a methodology that works well for *you* to assume that it's the right choice for everyone. But people are just too diverse for that to be true. There's no One Right Way to develop software (or anything else).

I write tests first myself, as I mentioned. That suits me, and many others, and it's the method I teach and recommend to my students. But it's not a *moral* issue. There's no point trying to shame people because they don't write their tests first, or getting into arguments about how "my TDD is purer than yours".

Instead, as Tim Bray suggests, judge by results. If someone is producing great, well-tested code by writing tests last, more power to 'em! Some very smart people work this way, so there must be something to it.

On the other hand, if the code *isn't* good, and the tests tend to be feeble, optimistic, or missing altogether, one of the contributory factors may be that the developers are trying to write tests last. Switching to a test-first style, though unfamiliar at first, may improve their results.

Ineffective code review

Code review is hard. Once code *exists*, people tend to get defensive about it. It takes great tact and skill to guide someone to improve their code after the fact, especially if it has to be done through text alone. A face-to-face, or at least voice-to-voice, conversation is much better.

Perhaps for this reason, most code review is rather cursory, and often pays little attention to tests, instead preferring to focus on little, line-by-line nitpicks and style issues. A more effective code review strategy is to focus *primarily* on tests.

Try this ten-step checklist for code review:

1. Do we even know *why* this change is being made? What is the business or user requirement that's being addressed, or bug that's being fixed? If we don't know this, how can we possibly decide whether or not the proposed change does what it's supposed to?
2. For each thing the change is supposed to accomplish, is there a test expressing that requirement? Does it *really* test that behaviour, or just claim to?
3. Do the tests pass with a fresh checkout of the source code? This might seem an obvious thing to ask, but it's surprising how often the answer is no. Presumably the tests pass on the *author's* machine, but they may have a hidden dependency on the presence of some file, binary, or environment variable. A fresh checkout on a different machine can often catch these problems.
4. Do the tests detect obvious bugs in the behaviour of the new code? Insert some bugs (review the chapter on mutation testing for more about this) and see if the tests catch them. If not, don't approve the changes until the tests can detect *at least* the bugs you thought of.
5. When the tests fail, do they produce accurate, helpful, complete information about the failure? Would you have to read the test code itself to understand the failure, or does the message contain everything you need to diagnose the bug? Exercise each possible failure path to check this.
6. If the new code accepts arbitrary data as input, has it been fuzz tested? (See the chapter on fuzz testing for more about this.) If not, add a fuzz test, and run it for a while. If the new code accepts a slice, map, or string as input, does it cope when those inputs are empty? Add test cases for this and check that they pass. In general, do the tests exercise the new code with a convincing sample and range of possible inputs?

7. Do the tests completely cover the new code? If not, what paths are left uncovered? Are any of them so trivial that they can be verified by eye? Or are they in practice unnecessary, and can they simply be removed instead?
8. Does the code do *more* than is required to pass the tests? In other words, of the code paths that *are* covered by tests, are they over-engineered? Could that code be simplified or shortened, or just do less, and still pass the test? If so, make those changes.
9. Do the *tests* test more than is required to verify the new code? It's always tempting to say to oneself, "While we're here, let's add a test for…" In other words, do the tests poke their noses into something that isn't really their business? Or are they brittle because in testing the new code, they rely too much on the correctness of *other* code in the system?
10. Is it clear from the test code alone how the system is supposed to behave? If not, refactor the test to make this clearer. Is the test *itself* relatively straightforward and obvious? If you showed it to a customer or non-technical business person, would they get the general idea? If not, why not?

Optimistic tests

Even when tests provide both broad and deep coverage of behaviour, they can still be too *optimistic*. That is, they're designed only to confirm that the system works, not to prove that it doesn't.

Surprisingly, one problem that optimistic tests can miss is when the system does *nothing at all*.

For example, consider a test for some function `user.Create`, like this:

```
func TestCreate(t *testing.T) {
    t.Parallel()
    user.Create("Alice")
    if !user.Exists("Alice") {
        t.Error("Alice not created")
    }
}
```

(Listing user/5)

At first glance, this is plausible. We create a test user, Alice, and check if she subsequently exists. If not, the test fails. That part is fine. So, can you see what's missing? If not, you might like to take a minute to think about it before reading on.

It's always worth asking of any test whether it rigorously checks its *preconditions* as well as its postconditions. The developer's focus, naturally enough, tends to be on the state

of the world *after* the operation under test, but that can result in some risky assumptions about its *prior* state.

Suppose `Create` actually does nothing. How could this test pass? Only when Alice already exists. And would we know if that was the case?

Our test checks that Alice exists *after* calling `Create`, but what's missing is a check that she doesn't exist *beforehand*.

In other words, suppose we don't clean up the database after each test run, so if Alice was *ever* created, she'll still be there. And suppose someone later introduces a bug into `Create` that prevents it from actually creating users.

To put it another way, here's an obviously incorrect implementation that nevertheless passes this test:

```
type User struct {
    Name string
}

var (
    m     = new(sync.Mutex)
    users = map[string]*User{
        "Alice": {
            Name: "Alice",
        },
    }
)

func Create(name string) {}

func Exists(name string) bool {
    m.Lock()
    defer m.Unlock()
    _, ok := users[name]
    return ok
}
```

(Listing user/5)

We thought we were testing `Create`, but we really aren't, because `Create` does nothing at all, yet the test doesn't detect that. Alice *always* exists, so the test always passes. This kind of *mirage test* is especially dangerous, because it looks like you have a test, but you don't.

It turns out that `Create` needs not just to leave the world in a state where Alice exists. What's important about `Create` is that it *changes* the world from a state where Alice doesn't exist to one where she does.

You might think that goes without saying, but we've just proved that it doesn't. We need to pay attention to preconditions as well as postconditions, according to the contract that the system under test is supposed to fulfil.

Let's write a test that *would* catch this bug, then:

```
func TestCreate(t *testing.T) {
    t.Parallel()
    if user.Exists("Alice") {
        t.Fatal("Alice unexpectedly exists")
    }
    user.Create("Alice")
    if !user.Exists("Alice") {
        t.Error("Alice not created")
    }
}
```

(Listing user/6)

The difference is very simple, but important: we check our preconditions.

What about *this* test, then?

```
func TestDelete(t *testing.T) {
    t.Parallel()
    user.Create("Alice")
    user.Delete("Alice")
    if user.Exists("Alice") {
        t.Error("Alice still exists after delete")
    }
}
```

Again, this looks reasonable on a cursory inspection. It creates Alice, deletes her (sorry, Alice), and then ensures that she no longer exists. What could be wrong with `Delete` that this test wouldn't catch?

Well, what if *both* `Create` and `Delete` do nothing at all? That seems like a pretty major bug, yet this test doesn't detect it. There are no preconditions, so the outcome of the test is the same whether `Create` and `Delete` actually have any effect or not. The test isn't wrong, as far as it goes: it just doesn't *go* far enough. There's a pretty big loophole in it.

This kind of bug isn't as unlikely as you might think, either. I've made this exact mistake

in the past: I stubbed out `Create` and `Delete` methods with placeholders, then forgot that I hadn't finished them, because the test was passing. It's easy to do.

What we're missing here, in fact, is another precondition: that the user *does* exist before we try to delete them.

```
func TestDelete(t *testing.T) {
    t.Parallel()
    user.Create("Alice")
    if !user.Exists("Alice") {
        t.Error("Alice not created")
    }
    user.Delete("Alice")
    if user.Exists("Alice") {
        t.Error("Alice still exists after delete")
    }
}
```

(Listing user/6)

If `Create` doesn't do anything, this test will fail at the first check. If `Delete` doesn't do anything, it'll fail at the second.

In any non-trivial codebase, you're pretty much guaranteed to find at least a few tests that are optimistically feeble in this way. Look for any test that doesn't properly establish its preconditions, and fix it. This will add a lot of value to the test suite overall.

Another example of this kind of problem is when the test fails to check some important but implicit *postconditions*. For example, in `TestDelete`, the explicit postcondition here is that Alice shouldn't exist after deletion, so what are we implicitly missing? What else could a reasonable person ask for from a `Delete` function?

As usual, a productive way to answer that is to think about possible bugs in `Delete`. Suppose, for example, that `Delete` mistakenly deletes not only Alice, but *all* users in the database. That kind of thing is surprisingly easy to do, especially with SQL queries (omitting a `WHERE` clause, for example).

If calling `Delete` on a single user instead nukes the whole database, that's a pretty major bug, wouldn't you say? This test doesn't detect it, because it focuses only on what *should* happen, and ignores what shouldn't.

> *It is tempting to test that the code does what it should do and leave it at that, but it's arguably even more important to test that it doesn't do what it shouldn't do.*
> —Mike Bland, "Goto Fail, Heartbleed, and Unit Testing Culture"

How *could* we detect such a bug, then? Quite easily, it turns out.

Here's what we do. We create *two* users in the test, but delete only one of them. Then we check that the one we deleted doesn't exist, and the one we didn't delete still exists:

```
func TestDelete(t *testing.T) {
    t.Parallel()
    user.Create("Alice")
    if !user.Exists("Alice") {
        t.Error("Alice not created")
    }
    user.Create("Bob")
    if !user.Exists("Bob") {
        t.Error("Bob not created")
    }
    user.Delete("Alice")
    if user.Exists("Alice") {
        t.Error("Alice still exists after delete")
    }
    if !user.Exists("Bob") {
        t.Error("Bob was unexpectedly deleted")
    }
}
```

This test has accumulated a bit of paperwork, so let's refactor that out into a helper function:

```
func TestDelete(t *testing.T) {
    t.Parallel()
    createUserOrFail(t, "Alice")
    createUserOrFail(t, "Bob")
    user.Delete("Alice")
    if user.Exists("Alice") {
        t.Error("Alice still exists after delete")
    }
    if !user.Exists("Bob") {
        t.Error("Bob was unexpectedly deleted")
    }
}

func createUserOrFail(t *testing.T, name string) {
    t.Helper()
    user.Create(name)
    if !user.Exists(name) {
        t.Errorf("%s not created", name)
```

```
    }
}
```

(Listing user/7)

Who would have thought that there were so much scope for things to go wrong with a seemingly simple `Delete` function? Well, *you* would, because you've read this book.

If you find yourself, as a result, becoming thoroughly sceptical about the idea that *anything* works the way it's supposed to, congratulations: you're thinking like a tester.

Persnickety tests

Sometimes, though not often, people can take testing a bit *too* much to heart, and test more than strictly necessary. As we've seen, it's much easier to err the other way, and leave out important things such as preconditions and implicit postconditions, like not deleting all the users in the database. But overtesting does afflict some test suites.

For example, there are many things that a reasonable person might expect from the system that nevertheless don't really need to be tested, because they're pretty hard to miss in practice:

> *Generally, you don't need to write down things like, "The computer should not explode, the operating system should not crash, the user should not be injured, errors should produce error messages, the application should load and not destroy other applications, and execution should not take so long that the user thinks the system has gone into an infinite loop."*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

Beyond these "halt and catch fire" issues, it's important to keep tests focused on only the part of the system they're supposed to care about, and on only the *behaviour* that matters. They should avoid checking for irrelevant things:

> *Specify precisely what should happen and no more. In a test, focus the assertions on just what's relevant to the scenario being tested. Avoid asserting values that aren't driven by the test inputs, and avoid reasserting behavior that is covered in other tests.*
> —Steve Freeman and Nat Pryce, "Growing Object-Oriented Software, Guided by Tests"

In particular, don't test for the impossible. Even though good testers are pessimistic (they would prefer the term "realistic"), they don't waste time writing tests for situations that mathematically can't occur, such as a value being both true and false at the same time. That's taking scepticism too far.

> *To doubt everything or to believe everything are two equally convenient solutions; both dispense with the necessity of reflection.*
> —Henri Poincaré, "Science and Hypothesis"

Beware also of simply comparing too much. As we've seen in some examples in this book, it can be convenient to compare a function's *entire* result against an expected value, rather than individually checking each of its fields.

That makes sense when all the fields are affected by the behaviour under test. But when they aren't, checking irrelevant fields makes the test brittle, and obscures its real purpose. The same applies to checking entire strings or output files, when only certain parts of the data are actually important.

> *The easiest way to avoid brittle tests is to check only the properties you care about. Be selective in your assertions. Don't check for exact string matches, for example, but look for relevant substrings that will remain unchanged as the program evolves.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

Watch out for tests that lazily compare output against a golden file, for example, when the behaviour they're testing is only about a small subset of that file. And we saw in the chapter on errors that most of the time a test should not assert the exact value of an error, but only that there *is* some error, when there's supposed to be.

Some exposure to the idea of *property-based* testing, such as we had in the chapter on fuzzing, can also be helpful for constructing robust tests. For example, what's important about a result is often not its *exact* value, but some *property* of the value, especially an invariant property.

I recently reviewed a program that needed to create a "fingerprint" of a piece of data, for deduplication purposes. In case the same data was submitted to the system later, the fingerprint would enable the system to recognise it without actually having to store all the data, which could be very large.

A cryptographic digest, or hash value, is an obvious way to do this, so the program had a test something like this:

```
func TestFingerprint(t *testing.T) {
    t.Parallel()
    data := []byte("These pretzels are making me thirsty.")
    want := md5.Sum(data)
    got := fingerprint.Hash(data)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing `fingerprint/1`)

The implementation of `Hash` doesn't matter, but let's assume it's something like this:

```
func Hash(data []byte) [md5.Size]byte {
```

```
    return md5.Sum(data)
}
```

(Listing fingerprint/1)

Fine. But MD5 is insecure, so I suggested using a SHA-256 hash instead:

```
func Hash(data []byte) [sha256.Size]byte {
    return sha256.Sum256(data)
}
```

(Listing fingerprint/2)

This broke the test, which makes no sense, because `Hash` *works*. So what's the problem?

Well, what are we really testing here? Not that `Hash` produces an *MD5* hash, specifically; that's incidental. What matters is that the same data should always hash to the same value, whatever that value actually is. And, no less importantly, that *different* data should hash to different values.

So we ended up with a test something like this instead:

```
func TestFingerprint(t *testing.T) {
    t.Parallel()
    data := []byte("These pretzels are making me thirsty.")
    orig := fingerprint.Hash(data)
    same := fingerprint.Hash(data)
    different := fingerprint.Hash([]byte("Hello, Newman"))
    if !cmp.Equal(orig, same) {
        t.Error("same data produced different hash")
    }
    if cmp.Equal(orig, different) {
        t.Error("different data produced same hash")
    }
}
```

(Listing fingerprint/2)

This test is better, because it doesn't care about the implementation details of `Hash`, such as which algorithm is used. What it cares about is that the same input to `Hash` always gives the same result, so it catches bugs like an unstable hash algorithm or even a random result.

And, because a maliciously lazy implementation of `Hash` might simply *always* return the same fixed value, the test also requires that different data hashes to different values.

This isn't completely bug-proof, of course. It's possible, for example, that `Hash` always produces the same value unless the input is exactly "Hello, Newman". We could use

fuzz testing to tackle this, or just a table test with a bunch of different inputs.

But you get the point. We made the test both less brittle and less feeble without adding much extra code; all it took was a little extra *thinking*.

Over-precise comparisons

Another way of overtesting is by making unnecessarily exacting comparisons. For example, two near-identical floating-point values may not actually compare equal in Go, because their representation in memory is inherently imprecise. Floating-point math involves a kind of *lossy compression*.

If some function returns a `float64` result, then, it's *saying* that the exact value isn't important, only that it should be close enough. Let's hope so, anyway, because we won't *get* the exact value, unless we just happen to luck on a number whose binary expansion fits neatly into 64 bits.

A floating-point result that's close *enough* will satisfy most users, though, unless they're mathematicians. Or perhaps game programmers:

> *Wanting to support the original [DOOM] demos means everything has to match exactly with the original game, as the demos are simply a sequence of input events, and the slightest calculation or behavior difference can cause "desync", ending up, for example, with the player running into, and shooting at, walls. I hit one such case, trying to save space by reflecting the quarter cycles of the sine table instead of storing all of them, and this caused desync even though the errors are at most 1/65536! On the bright side though, the demos turn out to be really good regression tests.*
> —Graham Sanderson, "The Making of RP2040 Doom"

So how can a *test* compare values in a suitably chilled-out way that doesn't sweat the smallest bits of precision? In general, all we need to do is check that the difference between the two numbers is smaller than some arbitrary *epsilon* value that we choose in advance.

If you recall, this is what we did in the previous chapter when we needed to compare two `time.Time` values expected to be very close, but not identical. And the `go-cmp` package has a built-in way of making such comparisons for floating-point numbers:

```
if !cmp.Equal(want, got, cmpopts.EquateApprox(0, 0.00001)) {
    t.Errorf("not close enough: want %.5f, got %.5f", want, got)
}
```

The exact value of epsilon that gives you confidence in the correctness of the answer will, of course, vary depending on the situation. A good way to refine it is (with a passing test) to keep making it smaller until the test fails. Then make it one order of magnitude bigger.

Too many tests

This might sound like a nice problem to have, and for most developers it would be. But every line of code in the project is a form of technical debt, and if it doesn't need to be there, it's just dead weight:

> *If we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent".*
> —Edsger W. Dijkstra, "On the cruelty of really teaching computing science"

A test suite is like a garden: from time to time, it benefits from a little judicious pruning. As the system evolves, some of its behaviours will no longer be used, or needed. The tests for those behaviours should be removed, along with their implementations.

And as new tests are written, they may well render some older tests unnecessary, or obsolete. Indeed, that's very likely if we gradually build up the system in small increments, as we've discussed throughout this book.

For example, we might build a component that does ten things, each of which has its own test. But once it's complete, we might able to replace those ten individual tests with a *single* test for the whole component.

In this case, a better analogy is with scaffolding: the tests support you and keep you safe during construction, but you can remove them once the building is up.

This isn't always the case, though. Suppose we took this idea to the extreme, and just wrote a single test for the entire system. Even if that were possible, it wouldn't be a good idea. Sure, the test would tell us if there's a bug anywhere in the system. But it wouldn't tell us *where*.

Location, in other words, can be as important as detection. It does the detective no good to know only that the murder suspect is somewhere in North America. An exhaustive search of the area would simply take too long.

If we agree that modularity is a desirable attribute of a well-designed system, then the structure of the tests should reflect the structure of the system itself. It should at least be possible, in principle, to pull any particular component out, along with its tests, and use it in isolation. If it's not, then you don't really *have* components, just a big ball of mud.

Some duplication of tests is inevitable, then, and it's not necessarily a problem. Indeed, it's useful to test some critical behaviours from a variety of different angles, and in a variety of ways. But we don't need to test the same thing in the same way more than once, so regular inspection and pruning of redundant tests is helpful for the flourishing of a codebase.

A good way to decide whether a particular test is pulling its own weight is to look at its *delta coverage*:

> *Herb Derby came up with this metric of "delta coverage". You look at your test suite and you measure what coverage each test adds uniquely that no other test*

> *provides.*
> *If you have lots of tests with no delta coverage, so that you could just delete them and not lose your ability to exercise parts of the system, then you should delete those tests, unless they have some some communication purpose.*
> Kent Beck, "Is TDD Dead?"

The `deltacoverage` tool is an (experimental) attempt to automate this.

In the same conversation, Martin Fowler suggests a more intuitive, but no less useful, way of assessing redundancy in test suites:

> *If you can't confidently change the code, you don't have enough tests. But if you change the code and you have to fix a bunch of tests, then you have too many tests.*
> Martin Fowler, "Is TDD Dead?"

Test frameworks

The straightforward and lightweight style of testing we've used in this book, relying only on the standard `testing` package (plus `go-cmp`), doesn't appeal to everyone. Some people would prefer to write their tests in a different way, using other third-party packages such as `testify` or `ginkgo/gomega`.

This is a snippet from a test written using `testify`, to give you an idea what it looks like:

```
header := GetBasicAuthHeader("grafana", "1234")
username, password, err := DecodeBasicAuthHeader(header)
require.NoError(t, err)
assert.Equal(t, "grafana", username)
assert.Equal(t, "1234", password)
```

And here's a snippet of a test written using `ginkgo` and `gomega`, which is even stranger:

```
var _ = Describe("Extensions", func() {
    Context("fetching app events", func() {
        It("no error occurs", func() {
            now := time.Now()
            before := now.Add(-10 * time.Minute)
            objs, err := target.GetEvents()
            Ω(err).ShouldNot(HaveOccurred())
            Ω(objs).Should(HaveLen(2))
            Ω(objs[0].GUID).Should(Equal("event1-guid"))
            Ω(objs[0].CreatedAt).Should(BeTemporally("==", before))
            Ω(objs[0].UpdatedAt).Should(BeTemporally("==", now))
            Ω(objs[0].Type).Should(Equal("event1-type"))
```

```
            ... // lots more 'should's
        })
    })
})
```

Weird flex, but okay. If you find this kind of thing appealing, I celebrate your diversity. So what's wrong with alternative test frameworks like this?

Well, nothing. They're just not necessary. Go itself already provides everything we need to write clear, effective, and easy-to-understand tests. There is nothing missing from Go that needs to be supplied by some third-party package. Indeed, adding such a package makes tests *harder* to understand, since now you have to be familiar with the API of that package.

If you want to write clear, effective, and easy-to-understand tests in a way that any Go programmer should be familiar with, then the best way to do that is:

```
import "testing"
```

Even if plain old `testing` doesn't appeal to you at first, stick with it. The unfamiliar is almost always unappealing. Once it *becomes* familiar, you'll wonder why you ever had a problem with it.

By contrast, adding a third-party framework adds an unnecessary dependency to your project, but more importantly, it adds an unnecessary learning curve for every new developer that joins the project. It'll be harder for them to understand the existing tests, and to write new tests. The last thing we want to do is discourage people from writing tests.

Let's talk briefly about one specific reason that developers sometimes reach for a third-party testing framework: to write *assertions*. For example:

```
assert.Equal(want, got)
```

One problem with this way of expressing tests is that it presumes the happy path. In other words, the test writer knows the behaviour they expect, and they simply assert it. If the assertion fails, the test fails.

Here's what we see when a `testify` assertion fails, for example:

```
Not equal:
expected: 0
actual  : 1
```

This isn't really helpful, because it merely reports that 1 does not equal 0, which we knew already.

> *An assertion function suffers from premature abstraction. It treats the failure of this particular test as a mere difference of two integers, so we forfeit the opportunity to provide meaningful context.*
> —Alan Donovan & Brian Kernighan, "The Go Programming Language"

A *helpful* test, by contrast, would explain what the failure *means*, and perhaps even offer some suggestions about what bug might be responsible:

```
if err == nil {
    t.Error("want error for empty slice; missing len() check?")
}
```

Or:

```
if !called {
    t.Error("test handler not called—incorrect URL set on client?")
}
```

It's possible to add helpful information like this in assertion-based tests, of course, but hardly anyone does. The API doesn't do much to encourage it; if anything, it encourages you *not* to think about the implications of a failure, and just move on to the next assertion instead.

A slightly more subtle problem with test code written using `testify` and friends is that it *doesn't look like real Go code*. Throughout this book we've discussed the value of being the first user of your own package, by writing tests against it.

But real users don't write assertions! A big advantage of *standard* Go tests is that they look just like ordinary Go code, because that's what they are. Stick to the standard library for testing unless there's a really compelling reason to use something else.

Flaky tests

Flaky tests sometimes fail, sometimes pass, regardless of whether the system is correct. There are many reasons for flaky tests, so let's look at a couple of them, with some possible solutions.

Timing issues can be a source of flakiness, as we've seen in previous chapters. In particular, fixed sleeps in tests are a bad idea (see the next section for more about these). Eliminate these wherever possible and replace them with code that only waits as long as strictly necessary.

When you need to *test* timing itself, use the shortest possible interval. For example, don't test a timer function with a one-second duration when one millisecond would work just as well.

We've also seen examples in this book of where the *time of day* can affect the test. As in those examples, we can eliminate this cause of flakiness by turning time into data, and if necessary injecting a fake `Now()` function to return a canned time of day.

Flakiness can also sometimes arise from *ordering issues*. Some data structures in Go are inherently unordered: maps, for example. Comparing these needs special care.

For example, iterating over a map comparing its elements is not good enough: the iteration order of maps is unspecified in Go. Instead, we can use the `go-cmp` package. Its `cmp.Equal` function compares maps regardless of iteration order:

```
func TestCompareMaps(t *testing.T) {
    t.Parallel()
    want := map[int]bool{1: true, 2: false}
    got := map[int]bool{2: false, 1: true}
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
// pass
```

(Listing order/1)

On the other hand, slices *are* inherently ordered, and `cmp.Equal` respects this:

```
func TestCompareSlices(t *testing.T) {
    t.Parallel()
    want := []int{1, 2, 3}
    got := []int{3, 2, 1}
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
// fail:
// -    1, 2, 3,
// +    3, 2, 1,
```

(Listing order/1)

But sometimes we don't actually care about the order. Maybe we get these results from some concurrent computations, and we don't know what order they will show up in. We just want to know that we *have* the right results.

To compare two slices for equal *elements*, then, regardless of order, we can use an option to `cmp.Equal` to sort them by some user-defined function:

```
func TestCompareSortedSlices(t *testing.T) {
    t.Parallel()
    want := []int{1, 2, 3}
    got := []int{3, 2, 1}
    sort := cmpopts.SortSlices(func(a, b int) bool {
```

```
        return a < b
    })
    if !cmp.Equal(want, got, sort) {
        t.Error(cmp.Diff(want, got, sort))
    }
}
// pass
```

(Listing order/1)

Whatever the cause of a flaky test suite, it's a serious problem. Left untreated, it will continuously erode value from the tests, until eventually they become useless and ignored by all. It should be a red flag to hear something like:

> "Oh yeah, that test just fails sometimes."

As soon as you hear that, you know that the test has become useless. Delete it, if the flakiness really can't be fixed. Thou shalt not suffer a flaky test to live. As soon as it starts flaking, it stops being a useful source of feedback, and bad tests are worse than no tests.

A *brittle* test is not the same thing as a flaky test: a brittle test fails when you change something unrelated, whereas a flaky test fails when it feels like it. Fixing brittle tests is usually a matter of decoupling entangled components, or simply reducing the scope (and thus sharpening the focus) of the test.

On the other hand, flaky tests can require some time and effort to find the underlying cause and address it. Only do this if the test is really worth it; if not, just delete it.

Shared worlds

Another possible underlying cause of flaky tests is a *shared world*: some test environment that's slow or expensive to set up, so to save time it's shared between many tests.

The trouble with a shared world is that any given test can't know much about the state of that world. Other tests may have already run and changed the world; other tests may change it *while* this test is running. That's a recipe for flakiness.

The best way to avoid this is to give each test its own private world, where it can know the initial state precisely, and do what it likes without worrying about interference with, or from, concurrent tests.

Where this isn't possible, though, and concurrent tests are forced to share a world, they will need to adopt a "take nothing but pictures, leave nothing but footprints" approach. In other words, the test mustn't rely on any particular initial state, it mustn't change anything that could affect other tests, and it should leave the world in the same state as it found it.

When tests have to share a world, they should ideally work just in their own little corner of it, without trespassing on other tests' territory.

For example, if the tests have to share a database server, then try to have them each create their own individual *database* to work in, named after the test. If that's not possible, then each test should at least have a database *table* to itself, to avoid locking issues and conflicts.

If even that's not possible, then the test should try to do everything inside a transaction, and roll it back at the end of the test. In a database that supports this, each test will "see" the data as it was at the start of its transaction, regardless of what any other test is doing in another transaction.

Importantly, the test should never *delete* anything it didn't *create*. Otherwise, it might delete something it wasn't supposed to. Also, don't write a test runner script or automation step that drops and recreates the database. It would be disastrous if some bug or accident one day triggered this on the production database instead of test.

Most systems have safeguards to prevent tests being accidentally run against production, but inevitably this will happen anyway at some point. Instead, design tests so that it *would* be safe to run them against live data, even if you don't plan to.

In other words, it should always be safe for the tests to run against the production DB, because one day they will.

Failing tests

What if some tests aren't just flaky, but fail all the time, because bugs aren't being fixed? This is a very dangerous situation, and without prompt action the tests will rapidly become completely useless.

Why? Because if tests are allowed to fail for a while without being fixed, people soon stop trusting them, or indeed paying *any* attention to them:

> "Oh yeah, that test always fails."

We can never have any failing tests, just as we can never have any bugs:

> *Don't fix bugs later; fix them now.*
> —Steve Maguire, "Writing Solid Code: Development Philosophies for Writing Bug-Free Programs"

As soon as any test starts failing, fixing it should be everyone's top priority. No one is allowed to deploy any code change that's not about fixing this bug.

Once you let one failing test slip through the net, all the other tests become worthless:

> *If you find yourself working on a project with quite a few broken windows, it's all too easy to slip into the mindset of "All the rest of this code is crap, I'll just follow suit."*

> —David Thomas & Andrew Hunt, "The Pragmatic Programmer: Your Journey to Mastery"

This so-called *zero defects methodology* sounds radical, but it really isn't. After all, what's the alternative?

> *The very first version of Microsoft Word for Windows was considered a "death march" project. Managers were so insistent on keeping to the schedule that programmers simply rushed through the coding process, writing extremely bad code, because bug-fixing was not a part of the formal schedule.*
> *Indeed, the schedule became merely a checklist of features waiting to be turned into bugs. In the post-mortem, this was referred to as "infinite defects methodology".*
> —Joel Spolsky, "The Joel Test: 12 Steps to Better Code"

Fixing bugs now is cheaper, quicker, and makes more business sense than fixing them later. The product should be ready to ship at all times, without bugs.

If you already *have* a large backlog of bugs, or failing tests, but the company's still in business, then maybe those bugs aren't really that critical after all. The best way out may be to declare voluntary *bug bankruptcy*: just close all old bugs, or delete all failing tests. Bugs that people *do* care about will pretty soon be re-opened.

> *If necessary, you can declare bug amnesty: remove all bugs from the bug database and send a notice to all issuers telling them to resubmit a bug if they're still concerned about it. If they remain concerned, add the bug back into the queue. This will greatly reduce the backlog.*
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

Slow tests

The world's greatest test suite does us no good if it takes too long to run. How long is too long? Well, if we're running tests every few minutes, clearly even a few minutes is too long. We simply won't run the tests often enough to get the fast feedback we need from them.

> *By running the test suite frequently, at least several times a day, you're able to detect bugs soon after they are introduced, so you can just look in the recent changes, which makes it much easier to find them.*
> —Martin Fowler, "Self-Testing Code"

One way or the other, then, we don't want to be more than about five minutes away from passing tests. So, again, how long is *too* long for a test suite to run?

Kent Beck suggests that ten minutes is a psychologically significant length of time:

> *The equivalent of 9.8 m/s^2 is the ten-minute test suite. Suites that take longer than ten minutes inevitably get trimmed, or the application tuned up, so the suite takes ten minutes again.*

—Kent Beck, "Test-Driven Development by Example"

We may perhaps call this psychological limit the *Beck time*. Beyond the ten-minute mark, the problem is so obvious to everybody that people are willing to put effort into speeding up the test suite. Below that time, people will probably grumble but put up with it.

That certainly doesn't mean that a ten-minute test suite is okay: it's not, for the reasons we've discussed. Let's look at a few simple ways to reduce the overall run-time of the test suite to something more manageable.

1. *Parallel tests*. As we've seen throughout this book, an inability to run certain tests in parallel is usually a design smell. Refactor so that each test has its own world, touches no global state, and can thus run in parallel. Adding parallelism to a suite that doesn't have it should speed it up by about an order of magnitude.
2. *Eliminate unnecessary I/O*. Once you go off the chip, things get slow. Do everything on the chip as far as possible, avoiding I/O operations such as network calls or accessing disk files. For example, we discussed using an `fstest.MapFS` as an in-memory filesystem in the chapter on "Users shouldn't do that", and using memory-backed `io.Readers` and `io.Writers` instead of real files.
3. *No remote network calls*. Instead of calling some remote API, call a local fake instead. Local networking happens right in the kernel, and while it's still not *fast*, it's a lot faster than actually going out onto the wire.
4. *Share fixtures between tests*. Any time you have some expensive fixture setup to do, such as loading data into a database, try to share its cost between as many tests as possible, so that they can all use it. If necessary, do the setup in a single test and then run a bunch of subtests against it.

   However, we need to be careful that the tests don't then become flaky as a result of too much fixture sharing. A flaky test is worse than a slow test.
5. *No fixed sleeps*. As we saw in the chapter on testing the untestable, a test that can't proceed until some concurrent operation has completed should use the "wait for success" pattern. This minimises wasted time, whereas a fixed sleep maximises it (or causes flaky tests, which is also bad).
6. *Throw hardware at the problem*. When you've made the test suite as fast as it can go and it's still slow, just run it on a faster computer. If the tests are mostly CPU-bound, rent a 256-core cloud machine and have it pull and run the tests on demand. CPU time costs a lot less than programmer time, especially given that hiring cheap programmers costs more money than it saves.
7. *Run slow tests nightly*. This is a last resort, but it might come to that. If you have a few tests that simply *can't* be speeded up any more, and they're dragging down the rest of the suite, extract them to a separate "slow test" suite, and run it on a schedule. Every night, perhaps; certainly no less frequently than that. Even nightly isn't great, but it's better than not running tests at all.

A fragrant future

Since you've made it to the end of this book (thanks for your patience), you now have everything you need to detect and eliminate evil design vapours, dispel unpleasant test suite odours, and build software that's fragrant and delightful—guided, of course, by tests.

I hope you've found a few useful things in the book that will change the way you write software for the better. I also hope it won't get you fired, unless it's for the right reasons.

Let's leave the last word in this book to the man whose work inspired it, and me, along with many others:

> *If you know what to type, then type. If you don't know what to type, then take a shower, and stay in the shower until you know what to type. Many teams would be happier, more productive, and smell a whole lot better if they took this advice.*
> —Kent Beck, "Test-Driven Development by Example"

About this book

Who wrote this?

John Arundel is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

- Learn Go remotely with me

Cover photo

NASA software engineering chief Melba Mouton, who is pictured on the cover of this book, did much important early work on automated testing, documentation, and other parts of the development process. She was a prominent advocate for software testing, which was used from the earliest days of the space program.

Mouton, who held both bachelor's and master's degrees in mathematics, led a team of NASA mathematicians known as "computers", many of them women, like those celebrated in the movie "Hidden Figures". Later, she became the head programmer and section chief at Goddard Space Flight Center, and received the Apollo Achievement Award as well as an Exceptional Performance Award for her work.

Thanks, Melba.

Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not doing this for the money: I'm doing it so that I can help bring the power of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

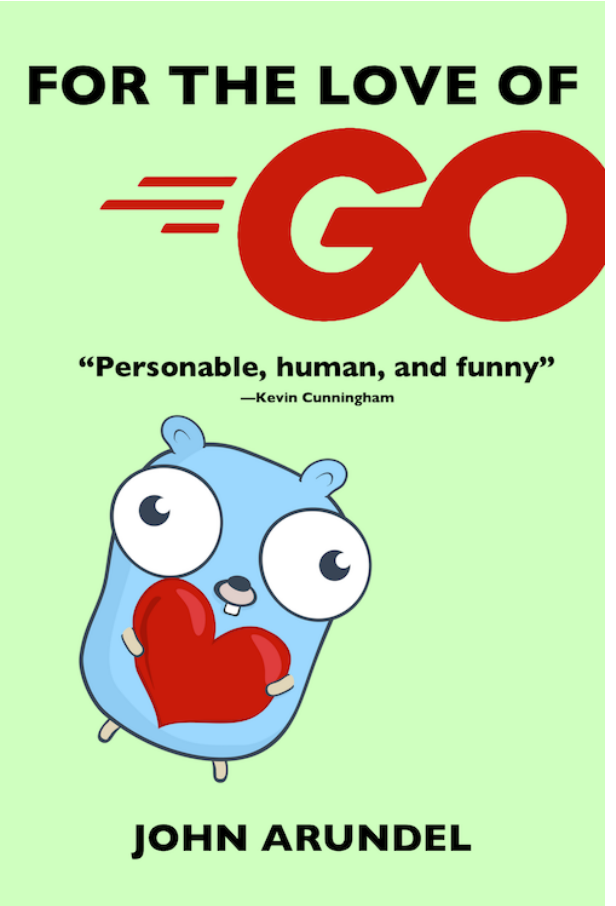
Mailing list

If you'd like to hear about it first when I publish new books, or even join my exclusive group of beta readers to give feedback on drafts in progress, you can subscribe to my mailing list here:

- Subscribe to Bitfield updates

For the Love of Go

For the Love of Go is a book introducing the Go programming language, suitable for complete beginners, as well as those with experience programming in other languages.



If you've used Go before but feel somehow you skipped something important, this book will build your confidence in the fundamentals. Take your first steps toward mastery with this fun, readable, and easy-to-follow guide.

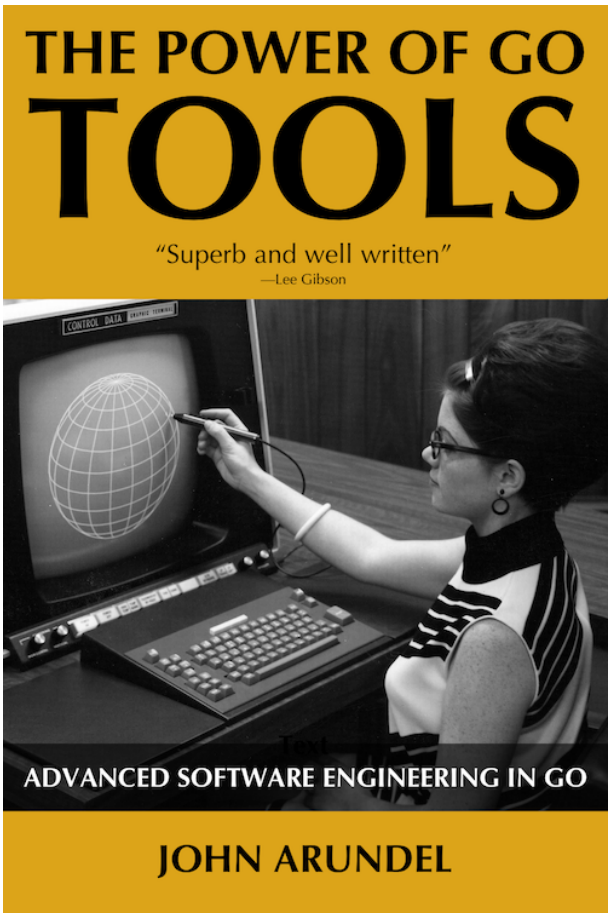
Throughout the book we'll be working together to develop a fun and useful project in Go: an online bookstore called Happy Fun Books. You'll learn how to use Go to store data about real-world objects such as books, how to write code to manage and modify that data, and how to build useful and effective programs around it.

The For the Love of Go: Video Course also includes this book.

The Power of Go: Tools

Are you ready to unlock the power of Go, master obviousness-oriented programming, and learn the secrets of Zen mountaineering? Then you're ready for The Power of Go:

Tools.



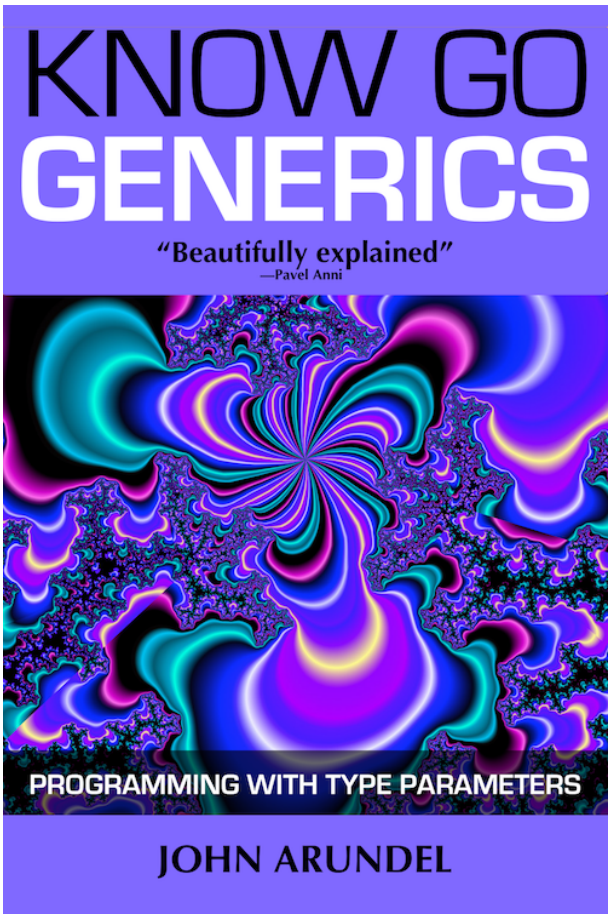
It's the next step on your software engineering journey, explaining how to write simple, powerful, idiomatic, and even beautiful programs in Go.

This friendly, supportive, yet challenging book will show you how master software engineers think, and guide you through the process of designing production-ready command-line tools in Go step by step. It's the perfect companion to *The Power of Go: Tests*.

Know Go: Generics

Things are changing! Go beyond the basics, and master the new generics features introduced in Go 1.18. Learn all about type parameters and constraints in Go and how to use

them, with this easy-to-read but comprehensive guide.



If you're new to Go and generics, and wondering what all the fuss is about, this book is for you! If you have some experience with Go already, but want to learn about the new generics features, this book is also for you. And if you've been waiting impatiently for Go to just get generics already so you can use it, don't worry: this book is for you too!

You don't need an advanced degree in computer science or tons of programming experience. Know Go: Generics explains what you need to know in plain, ordinary language, with simple examples that will show you what's new, how the language changes will affect you, and exactly how to use generics in your own programs and packages.

As you'd expect from the author of For the Love of Go and The Power of Go: Tools, it's fun and easy reading, but it's also packed with powerful ideas, concepts, and techniques that you can use in real-world applications.

Further reading

You can find more of my books on Go here:

- Go books by John Arundel

You can find more Go tutorials and exercises here:

- Go tutorials from Bitfield

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- Bitfield Consulting on YouTube

Credits

Gopher images by MariaLetta.

Acknowledgements

Many people read early drafts of this book and gave me their feedback, which was hugely valuable, and contributed a lot. I'm sorry I can't thank everyone individually, but I'm especially grateful for the thoughtful comments of Michele Adduci, Dave Bailey, Sean Burgoyne, Mary Clemons, Kevin Cunningham, Ivan Fetch, Paul Jolly, Jackson Kato, Manoj Kumar, Glenn Lewis, Joanna Liana, Giuseppe Maxia, Daniel Martí, and Peter Nunn.

If you'd like to be one of my beta readers in future, please go to my website, enter your email address, and tick the appropriate box to join my mailing list:

- https://bitfieldconsulting.com/