

LEARN GO WITH TESTS



LEARN THE GO PROGRAMMING LANGUAGE AND
TEST DRIVEN DEVELOPMENT

Chris James & other cool people

Contents

Learn Go with Tests	3
Why unit tests and how to make them work for you	6
Hello, World	15
Integers	28
Iteration	32
Arrays and slices	35
Structs, methods & interfaces	47
Pointers & errors	61
Maps	75
Dependency Injection	89
Mocking	95
Concurrency	110
Select	120
Reflection	130
Sync	150
Context	157
Roman Numerals	168
Mathematics	194
Reading files	243
HTML Templates	264
Generics	290
Revisiting arrays and slices with generics	303
Introduction to acceptance testing	315

Learn Go with Tests - Scaling Acceptance Tests (and light intro to gRPC)	326
Working without mocks, stubs and spies	367
Build an application	386
HTTP Server	387
JSON, routing & embedding	411
IO and sorting	429
Command line and project structure	457
Time	473
WebSockets	499
OS Exec	524
Error types	527
Context-aware readers	532
HTTP Handlers Revisited	538
TDD Anti-patterns	548

Learn Go with Tests

[Art by Denise](#)

Support me

I am proud to offer this resource for free, but if you wish to give some appreciation

- [Tweet me @quii](#)
- Mastodon
- [Buy me a coffee](#)
- [Sponsor me on GitHub](#)

Learn test-driven development with Go

- Explore the Go language by writing tests

-
- **Get a grounding with TDD.** Go is a good language for learning TDD because it is a simple language to learn and testing is built-in
 - Be confident that you'll be able to start writing robust, well-tested systems in Go

Translations:

- 中文
- Português
- 日本語
- 한국어
- Türkçe

Background

I have some experience introducing Go to development teams and have tried different approaches as to how to grow a team from some people curious about Go into highly effective writers of Go systems.

What didn't work

Read the book An approach we tried was to take [the blue book](#) and every week discuss the next chapter along with the exercises.

I love this book but it requires a high level of commitment. The book is very detailed in explaining concepts, which is obviously great but it means that the progress is slow and steady - this is not for everyone.

I found that whilst a small number of people would read chapter X and do the exercises, many people didn't.

Solve some problems Katas are fun but they are usually limited in their scope for learning a language; you're unlikely to use goroutines to solve a kata.

Another problem is when you have varying levels of enthusiasm. Some people just learn way more of the language than others and when demonstrating what they have done end up confusing people with features the others are not familiar with.

This ends up making the learning feel quite unstructured and ad hoc.

What did work

By far the most effective way was by slowly introducing the fundamentals of the language by reading through [go by example](#), explor-

ing them with examples and discussing them as a group. This was a more interactive approach than "read chapter x for homework".

Over time the team gained a solid foundation of the grammar of the language so we could then start to build systems.

This to me seems analogous to practicing scales when trying to learn guitar.

It doesn't matter how artistic you think you are, you are unlikely to write good music without understanding the fundamentals and practicing the mechanics.

What works for me

When I learn a new programming language I usually start by messing around in a REPL but eventually, I need more structure.

What I like to do is explore concepts and then solidify the ideas with tests. Tests verify the code I write is correct and documents the feature I have learned.

Taking my experience of learning with a group and my own personal way I am going to try and create something that hopefully proves useful to other teams. Learning the fundamentals by writing small tests so that you can then take your existing software design skills and ship some great systems.

Who this is for

- People who are interested in picking up Go
- People who already know some Go, but want to explore testing more

What you'll need

- A computer!
- [Installed Go](#)
- A text editor
- Some experience with programming. Understanding of concepts like if, variables, functions etc.
- Comfortable using the terminal

Feedback

- Add issues/submit PRs [here](#) or [tweet me @quii](#)

[MIT license](#)

Why unit tests and how to make them work for you

[Here's a link to a video of me chatting about this topic](#)

If you're not into videos, here's wordy version of it.

Software

The promise of software is that it can change. This is why it is called soft ware, it is malleable compared to hardware. A great engineering team should be an amazing asset to a company, writing systems that can evolve with a business to keep delivering value.

So why are we so bad at it? How many projects do you hear about that outright fail? Or become "legacy" and have to be entirely re-written (and the re-writes often fail too!)

How does a software system "fail" anyway? Can't it just be changed until it's correct? That's what we're promised!

A lot of people are choosing Go to build systems because it has made a number of choices which one hopes will make it more legacy-proof.

- Compared to my previous life of Scala where I described how it has enough rope to hang yourself, Go has only 25 keywords and a lot of systems can be built from the standard library and a few other small libraries. The hope is that with Go you can write code and come back to it in 6 months time and it'll still make sense.
- The tooling in respect to testing, benchmarking, linting & shipping is first class compared to most alternatives.
- The standard library is brilliant.
- Very fast compilation speed for tight feedback loops
- The Go backward compatibility promise. It looks like Go will get generics and other features in the future but the designers have promised that even Go code you wrote 5 years ago will still build. I literally spent weeks upgrading a project from Scala 2.8 to 2.10.

Even with all these great properties we can still make terrible systems, so we should look to the past and understand lessons in software engineering that apply no matter how shiny (or not) your language is.

In 1974 a clever software engineer called [Manny Lehman](#) wrote [Lehman's laws of software evolution](#).

The laws describe a balance between forces driving new developments on one hand, and forces that slow down progress on the other hand.

These forces seem like important things to understand if we have any hope of not being in an endless cycle of shipping systems that turn into legacy and then get re-written over and over again.

The Law of Continuous Change

Any software system used in the real-world must change or become less and less useful in the environment

It feels obvious that a system has to change or it becomes less useful but how often is this ignored?

Many teams are incentivised to deliver a project on a particular date and then move on to the next project. If the software is "lucky" there is at least some kind of hand-off to another set of individuals to maintain it, but they didn't write it of course.

People often concern themselves with trying to pick a framework which will help them "deliver quickly" but not focusing on the longevity of the system in terms of how it needs to evolve.

Even if you're an incredible software engineer, you will still fall victim to not knowing the future needs of your system. As the business changes some of the brilliant code you wrote is now no longer relevant.

Lehman was on a roll in the 70s because he gave us another law to chew on.

The Law of Increasing Complexity

As a system evolves, its complexity increases unless work is done to reduce it

What he's saying here is we can't have software teams as blind feature factories, piling more and more features on to software in the hope it will survive in the long run.

We **have** to keep managing the complexity of the system as the knowledge of our domain changes.

Refactoring

There are many facets of software engineering that keeps software malleable, such as:

- Developer empowerment
- Generally "good" code. Sensible separation of concerns, etc etc
- Communication skills

-
- Architecture
 - Observability
 - Deployability
 - Automated tests
 - Feedback loops

I am going to focus on refactoring. It's a phrase that gets thrown around a lot "we need to refactor this" - said to a developer on their first day of programming without a second thought.

Where does the phrase come from? How is refactoring just different from writing code?

I know that I and many others have thought we were doing refactoring but we were mistaken

[Martin Fowler describes how people are getting it wrong](#)

However the term "refactoring" is often used when it's not appropriate. If somebody talks about a system being broken for a couple of days while they are refactoring, you can be pretty sure they are not refactoring.

So what is it?

Factorisation

When learning maths at school you probably learned about factorisation. Here's a very simple example

Calculate $1/2 + 1/4$

To do this you factorise the denominators, turning the expression into $2/4 + 1/4$ which you can then turn into $3/4$.

We can take some important lessons from this. When we factorise the expression we have **not changed the meaning of the expression**. Both of them equal $3/4$ but we have made it easier for us to work with; by changing $1/2$ to $2/4$ it fits into our "domain" easier.

When you refactor your code, you are trying to find ways of making your code easier to understand and "fit" into your current understanding of what the system needs to do. Crucially **you should not be changing behaviour**.

An example in Go Here is a function which greets name in a particular language

```
func Hello(name, language string) string {
```

```
if language == "es" {
    return "Hola, " + name
}

if language == "fr" {
    return "Bonjour, " + name
}

// imagine dozens more languages

return "Hello, " + name
}
```

Having dozens of if statements doesn't feel good and we have a duplication of concatenating a language specific greeting with , and the name. So I'll refactor the code.

```
func Hello(name, language string) string {
    return fmt.Sprintf(
        "%s, %s",
        greeting(language),
        name,
    )
}

var greetings = map[string]string {
    "es": "Hola",
    "fr": "Bonjour",
    //etc..
}

func greeting(language string) string {
    greeting, exists := greetings[language]

    if exists {
        return greeting
    }

    return "Hello"
}
```

The nature of this refactor isn't actually important, what's important is I haven't changed behaviour.

When refactoring you can do whatever you like, add interfaces, new types, functions, methods etc. The only rule is you don't change behaviour

When refactoring code you must not be changing behaviour

This is very important. If you are changing behaviour at the same time you are doing two things at once. As software engineers we learn to break systems up into different files/packages/functions/etc because we know trying to understand a big blob of stuff is hard.

We don't want to have to be thinking about lots of things at once because that's when we make mistakes. I've witnessed so many refactoring endeavours fail because the developers are biting off more than they can chew.

When I was doing factorisations in maths classes with pen and paper I would have to manually check that I hadn't changed the meaning of the expressions in my head. How do we know we aren't changing behaviour when refactoring when working with code, especially on a system that is non-trivial?

Those who choose not to write tests will typically be reliant on manual testing. For anything other than a small project this will be a tremendous time-sink and does not scale in the long run.

In order to safely refactor you need unit tests because they provide

- Confidence you can reshape code without worrying about changing behaviour
- Documentation for humans as to how the system should behave
- Much faster and more reliable feedback than manual testing

An example in Go A unit test for our Hello function could look like this

```
func TestHello(t *testing.T) {  
    got := Hello( "Chris" , es)  
    want := "Hola, Chris"  
  
    if got != want {  
        t.Errorf("got %q want %q", got, want)  
    }  
}
```

At the command line I can run go test and get immediate feedback as to whether my refactoring efforts have altered behaviour. In practice it's best to learn the magic button to run your tests within your editor/IDE.

You want to get in to a state where you are doing

- Small refactor

-
- Run tests
 - Repeat

All within a very tight feedback loop so you don't go down rabbit holes and make mistakes.

Having a project where all your key behaviours are unit tested and give you feedback well under a second is a very empowering safety net to do bold refactoring when you need to. This helps us manage the incoming force of complexity that Lehman describes.

If unit tests are so great, why is there sometimes resistance to writing them?

On the one hand you have people (like me) saying that unit tests are important for the long term health of your system because they ensure you can keep refactoring with confidence.

On the other you have people describing experiences of unit tests actually hindering refactoring.

Ask yourself, how often do you have to change your tests when refactoring? Over the years I have been on many projects with very good test coverage and yet the engineers are reluctant to refactor because of the perceived effort of changing tests.

This is the opposite of what we are promised!

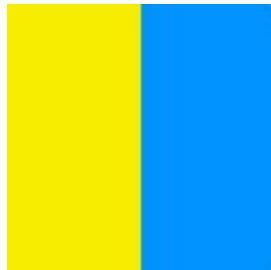
Why is this happening?

Imagine you were asked to develop a square and we thought the best way to accomplish that would be stick two triangles together.



We write our unit tests around our square to make sure the sides are equal and then we write some tests around our triangles. We want to make sure our triangles render correctly so we assert that the angles sum up to 180 degrees, perhaps check we make 2 of them, etc etc. Test coverage is really important and writing these tests is pretty easy so why not?

A few weeks later The Law of Continuous Change strikes our system and a new developer makes some changes. She now believes it would be better if squares were formed with 2 rectangles instead of 2 triangles.



She tries to do this refactor and gets mixed signals from a number of failing tests. Has she actually broken important behaviours here? She now has to dig through these triangle tests and try and understand what's going on.

It's not actually important that the square was formed out of triangles but **our tests have falsely elevated the importance of our implementation details**.

Favour testing behaviour rather than implementation detail

When I hear people complaining about unit tests it is often because the tests are at the wrong abstraction level. They're testing implementation details, overly spying on collaborators and mocking too much.

I believe it stems from a misunderstanding of what unit tests are and chasing vanity metrics (test coverage).

If I am saying just test behaviour, should we not just only write system/black-box tests? These kind of tests do have lots of value in terms of verifying key user journeys but they are typically expensive to write and slow to run. For that reason they're not too helpful for refactoring because the feedback loop is slow. In addition black box tests don't tend to help you very much with root causes compared to unit tests.

So what is the right abstraction level?

Writing effective unit tests is a design problem

Forgetting about tests for a moment, it is desirable to have within your system self-contained, decoupled "units" centered around key concepts in your domain.

I like to imagine these units as simple Lego bricks which have coherent APIs that I can combine with other bricks to make bigger systems. Underneath these APIs there could be dozens of things (types, functions et al) collaborating to make them work how they need to.

For instance if you were writing a bank in Go, you might have an "account" package. It will present an API that does not leak implementation detail and is easy to integrate with.

If you have these units that follow these properties you can write unit tests against their public APIs. By definition these tests can only be testing useful behaviour. Underneath these units I am free to refactor the implementation as much as I need to and the tests for the most part should not get in the way.

Are these unit tests?

YES. Unit tests are against "units" like I described. They were never about only being against a single class/function/whatever.

Bringing these concepts together

We've covered

- Refactoring
- Unit tests
- Unit design

What we can start to see is that these facets of software design reinforce each other.

Refactoring

- Gives us signals about our unit tests. If we have to do manual checks, we need more tests. If tests are wrongly failing then our tests are at the wrong abstraction level (or have no value and should be deleted).
- Helps us handle the complexities within and between our units.

Unit tests

- Give a safety net to refactor.

-
- Verify and document the behaviour of our units.

(Well designed) units

- Easy to write meaningful unit tests.
- Easy to refactor.

Is there a process to help us arrive at a point where we can constantly refactor our code to manage complexity and keep our systems malleable?

Why Test Driven Development (TDD)

Some people might take Lehman's quotes about how software has to change and overthink elaborate designs, wasting lots of time upfront trying to create the "perfect" extensible system and end up getting it wrong and going nowhere.

This is the bad old days of software where an analyst team would spend 6 months writing a requirements document and an architect team would spend another 6 months coming up with a design and a few years later the whole project fails.

I say bad old days but this still happens!

Agile teaches us that we need to work iteratively, starting small and evolving the software so that we get fast feedback on the design of our software and how it works with real users; TDD enforces this approach.

TDD addresses the laws that Lehman talks about and other lessons hard learned through history by encouraging a methodology of constantly refactoring and delivering iteratively.

Small steps

- Write a small test for a small amount of desired behaviour
- Check the test fails with a clear error (red)
- Write the minimal amount of code to make the test pass (green)
- Refactor
- Repeat

As you become proficient, this way of working will become natural and fast.

You'll come to expect this feedback loop to not take very long and feel uneasy if you're in a state where the system isn't "green" because it indicates you may be down a rabbit hole.

You'll always be driving small & useful functionality comfortably backed by the feedback from your tests.

Wrapping up

- The strength of software is that we can change it. Most software will require change over time in unpredictable ways; but don't try and over-engineer because it's too hard to predict the future.
- Instead we need to make it so we can keep our software malleable. In order to change software we have to refactor it as it evolves or it will turn into a mess
- A good test suite can help you refactor quicker and in a less stressful manner
- Writing good unit tests is a design problem so think about structuring your code so you have meaningful units that you can integrate together like Lego bricks.
- TDD can help and force you to design well factored software iteratively, backed by tests to help future work as it arrives.

Hello, World

[You can find all the code for this chapter here](#)

It is traditional for your first program in a new language to be [Hello, World](#).

- Create a folder wherever you like
- Put a new file in it called hello.go and put the following code inside it

```
package main
```

```
import "fmt"
```

```
func main() {
    fmt.Println("Hello, world")
}
```

To run it type go run hello.go.

How it works

When you write a program in Go, you will have a main package defined with a main func inside it. Packages are ways of grouping up related Go code together.

The func keyword is how you define a function with a name and a body.

With import "fmt" we are importing a package which contains the `Println` function that we use to print.

How to test

How do you test this? It is good to separate your "domain" code from the outside world (side-effects). The `fmt.Println` is a side effect (printing to stdout) and the string we send in is our domain.

So let's separate these concerns so it's easier to test

```
package main
```

```
import "fmt"
```

```
func Hello() string {
    return "Hello, world"
}
```

```
func main() {
    fmt.Println(Hello())
}
```

We have created a new function again with `func` but this time we've added another keyword `string` in the definition. This means this function returns a string.

Now create a new file called `hello_test.go` where we are going to write a test for our `Hello` function

```
package main
```

```
import "testing"
```

```
func TestHello(t *testing.T) {
    got := Hello()
    want := "Hello, world"

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

Go modules?

The next step is to run the tests. Enter go test in your terminal. If the tests pass, then you are probably using an earlier version of Go. However, if you are using Go 1.16 or later, then the tests will likely not run at all. Instead, you will see an error message like this in the terminal:

```
$ go test  
go: cannot find main module; see 'go help modules'
```

What's the problem? In a word, [modules](#). Luckily, the problem is easy to fix. Enter go mod init hello in your terminal. That will create a new file with the following contents:

```
module hello
```

```
go 1.16
```

This file tells the go tools essential information about your code. If you planned to distribute your application, you would include where the code was available for download as well as information about dependencies. For now, your module file is minimal, and you can leave it that way. To read more about modules, [you can check out the reference in the Golang documentation](#). We can get back to testing and learning Go now since the tests should run, even on Go 1.16.

In future chapters you will need to run go mod init SOMENAME in each new folder before running commands like go test or go build.

Back to Testing

Run go test in your terminal. It should've passed! Just to check, try deliberately breaking the test by changing the want string.

Notice how you have not had to pick between multiple testing frameworks and then figure out how to install. Everything you need is built in to the language and the syntax is the same as the rest of the code you will write.

Writing tests

Writing a test is just like writing a function, with a few rules

- It needs to be in a file with a name like xxx_test.go
- The test function must start with the word Test
- The test function takes one argument only t *testing.T
- In order to use the *testing.T type, you need to import "testing", like we did with fmt in the other file

For now, it's enough to know that your `t` of type `*testing.T` is your "hook" into the testing framework so you can do things like `t.Fail()` when you want to fail.

We've covered some new topics:

if If statements in Go are very much like other programming languages.

Declaring variables We're declaring some variables with the syntax `varName := value`, which lets us re-use some values in our test for readability.

t.Error We are calling the `Errorf` method on our `t` which will print out a message and fail the test. The `f` stands for format which allows us to build a string with values inserted into the placeholder values `%q`. When you made the test fail it should be clear how it works.

You can read more about the placeholder strings in the [fmt go doc](#). For tests `%q` is very useful as it wraps your values in double quotes.

We will later explore the difference between methods and functions.

Go doc

Another quality of life feature of Go is the documentation. You can launch the docs locally by running `godoc -http :8000`. If you go to [localhost:8000/pkg](#) you will see all the packages installed on your system.

The vast majority of the standard library has excellent documentation with examples. Navigating to <http://localhost:8000/pkg/testing/> would be worthwhile to see what's available to you.

If you don't have `godoc` command, then maybe you are using the newer version of Go (1.14 or later) which is [no longer including godoc](#). You can manually install it with `go install golang.org/x/tools/cmd/godoc@latest`.

Hello, YOU

Now that we have a test we can iterate on our software safely.

In the last example we wrote the test after the code had been written just so you could get an example of how to write a test and declare a function. From this point on we will be writing tests first.

Our next requirement is to let us specify the recipient of the greeting.

Let's start by capturing these requirements in a test. This is basic test driven development and allows us to make sure our test is actually testing what we want. When you retrospectively write tests there is the risk that your test may continue to pass even if the code doesn't work as intended.

```
package main

import "testing"

func TestHello(t *testing.T) {
    got := Hello("Chris")
    want := "Hello, Chris"

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

Now run go test, you should have a compilation error

```
./hello_test.go:6:18: too many arguments in call to Hello
    have (string)
    want ()
```

When using a statically typed language like Go it is important to listen to the compiler. The compiler understands how your code should snap together and work so you don't have to.

In this case the compiler is telling you what you need to do to continue. We have to change our function Hello to accept an argument.

Edit the Hello function to accept an argument of type string

```
func Hello(name string) string {
    return "Hello, world"
}
```

If you try and run your tests again your hello.go will fail to compile because you're not passing an argument. Send in "world" to make it compile.

```
func main() {
    fmt.Println(Hello("world"))
}
```

Now when you run your tests you should see something like

```
hello_test.go:10: got 'Hello, world' want 'Hello, Chris'
```

We finally have a compiling program but it is not meeting our requirements according to the test.

Let's make the test pass by using the name argument and concatenate it with Hello,

```
func Hello(name string) string {
    return "Hello, " + name
}
```

When you run the tests they should now pass. Normally as part of the TDD cycle we should now refactor.

A note on source control

At this point, if you are using source control (which you should!) I would commit the code as it is. We have working software backed by a test.

I wouldn't push to master though, because I plan to refactor next. It is nice to commit at this point in case you somehow get into a mess with refactoring - you can always go back to the working version.

There's not a lot to refactor here, but we can introduce another language feature, constants.

Constants

Constants are defined like so

```
const englishHelloPrefix = "Hello, "
```

We can now refactor our code

```
const englishHelloPrefix = "Hello, "
```

```
func Hello(name string) string {
    return englishHelloPrefix + name
}
```

After refactoring, re-run your tests to make sure you haven't broken anything.

It's worth thinking about creating constants to capture the meaning of values and sometimes to aid performance.

Hello, world... again

The next requirement is when our function is called with an empty string it defaults to printing "Hello, World", rather than "Hello, ".

Start by writing a new failing test

```
func TestHello(t *testing.T) {
    t.Run("saying hello to people", func(t *testing.T) {
        got := Hello("Chris")
        want := "Hello, Chris"

        if got != want {
            t.Errorf("got %q want %q", got, want)
        }
    })
    t.Run("say 'Hello, World' when an empty string is supplied", func(t *testing.T) {
        got := Hello("")
        want := "Hello, World"

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

Here we are introducing another tool in our testing arsenal, subtests. Sometimes it is useful to group tests around a "thing" and then have subtests describing different scenarios.

A benefit of this approach is you can set up shared code that can be used in the other tests.

While we have a failing test, let's fix the code, using an if.

```
const englishHelloPrefix = "Hello, "

func Hello(name string) string {
    if name == "" {
        name = "World"
    }
    return englishHelloPrefix + name
}
```

If we run our tests we should see it satisfies the new requirement and we haven't accidentally broken the other functionality.

It is important that your tests are clear specifications of what the code needs to do. But there is repeated code when we check if the message is what we expect.

Refactoring is not just for the production code!

Now that the tests are passing, we can and should refactor our tests.

```
func TestHello(t *testing.T) {
    t.Run("saying hello to people", func(t *testing.T) {
        got := Hello("Chris")
        want := "Hello, Chris"
        assertCorrectMessage(t, got, want)
    })

    t.Run("empty string defaults to 'world'", func(t *testing.T) {
        got := Hello("")
        want := "Hello, World"
        assertCorrectMessage(t, got, want)
    })
}

func assertCorrectMessage(t testing.TB, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("got %q want %q", got, want)
    }
}
```

What have we done here?

We've refactored our assertion into a new function. This reduces duplication and improves readability of our tests. We need to pass in *t* *testing.T so that we can tell the test code to fail when we need to.

For helper functions, it's a good idea to accept a testing.TB which is an interface that *testing.T and *testing.B both satisfy, so you can call helper functions from a test, or a benchmark (don't worry if words like "interface" mean nothing to you right now, it will be covered later).

t.Helper() is needed to tell the test suite that this method is a helper. By doing this when it fails the line number reported will be in our function call rather than inside our test helper. This will help other developers track down problems easier. If you still don't understand, comment it out, make a test fail and observe the test output. Comments in Go are a great way to add additional information to your code, or in this case, a quick way to tell the compiler to ignore a line. You can comment out the *t*.Helper() code by adding two forward slashes // at the beginning of the line. You should see that line turn grey or change to another color than the rest of your code to indicate it's now commented out.

Back to source control

Now we are happy with the code I would amend the previous commit so we only check in the lovely version of our code with its test.

Discipline

Let's go over the cycle again

- Write a test
- Make the compiler pass
- Run the test, see that it fails and check the error message is meaningful
- Write enough code to make the test pass
- Refactor

On the face of it this may seem tedious but sticking to the feedback loop is important.

Not only does it ensure that you have relevant tests, it helps ensure you design good software by refactoring with the safety of tests.

Seeing the test fail is an important check because it also lets you see what the error message looks like. As a developer it can be very hard to work with a codebase when failing tests do not give a clear idea as to what the problem is.

By ensuring your tests are fast and setting up your tools so that running tests is simple you can get in to a state of flow when writing your code.

By not writing tests you are committing to manually checking your code by running your software which breaks your state of flow and you won't be saving yourself any time, especially in the long run.

Keep going! More requirements

Goodness me, we have more requirements. We now need to support a second parameter, specifying the language of the greeting. If a language is passed in that we do not recognise, just default to English.

We should be confident that we can use TDD to flesh out this functionality easily!

Write a test for a user passing in Spanish. Add it to the existing suite.

```
t.Run("in Spanish", func(t *testing.T) {
    got := Hello("Elodie", "Spanish")
    want := "Hola, Elodie"
```

```
    assertCorrectMessage(t, got, want)
})
```

Remember not to cheat! Test first. When you try and run the test, the compiler should complain because you are calling Hello with two arguments rather than one.

```
./hello_test.go:27:19: too many arguments in call to Hello
    have (string, string)
        want (string)
```

Fix the compilation problems by adding another string argument to Hello

```
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }
    return englishHelloPrefix + name
}
```

When you try and run the test again it will complain about not passing through enough arguments to Hello in your other tests and in hello.go

```
./hello.go:15:19: not enough arguments in call to Hello
    have (string)
        want (string, string)
```

Fix them by passing through empty strings. Now all your tests should compile and pass, apart from our new scenario

```
hello_test.go:29: got 'Hello, Elodie' want 'Hola, Elodie'
```

We can use if here to check the language is equal to "Spanish" and if so change the message

```
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    if language == "Spanish" {
        return "Hola, " + name
    }
    return englishHelloPrefix + name
}
```

The tests should now pass.

Now it is time to refactor. You should see some problems in the code, "magic" strings, some of which are repeated. Try and refactor it your-

self, with every change make sure you re-run the tests to make sure your refactoring isn't breaking anything.

```
const spanish = "Spanish"
const englishHelloPrefix = "Hello, "
const spanishHelloPrefix = "Hola, "

func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    if language == spanish {
        return spanishHelloPrefix + name
    }
    return englishHelloPrefix + name
}
```

French

- Write a test asserting that if you pass in "French" you get "Bonjour,"
- See it fail, check the error message is easy to read
- Do the smallest reasonable change in the code

You may have written something that looks roughly like this

```
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    if language == spanish {
        return spanishHelloPrefix + name
    }
    if language == french {
        return frenchHelloPrefix + name
    }
    return englishHelloPrefix + name
}
```

switch

When you have lots of if statements checking a particular value it is common to use a switch statement instead. We can use switch to refactor the code to make it easier to read and more extensible if we wish to add more language support later

```
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    prefix := englishHelloPrefix

    switch language {
        case "French":
            prefix = frenchHelloPrefix
        case "Spanish":
            prefix = spanishHelloPrefix
    }

    return prefix + name
}
```

Write a test to now include a greeting in the language of your choice and you should see how simple it is to extend our amazing function.

one...last...refactor?

You could argue that maybe our function is getting a little big. The simplest refactor for this would be to extract out some functionality into another function.

```
const (
    french = "French"
    spanish = "Spanish"

    englishHelloPrefix = "Hello, "
    spanishHelloPrefix = "Hola, "
    frenchHelloPrefix = "Bonjour, "
)

func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    return greetingPrefix(language) + name
}

func greetingPrefix(language string) (prefix string) {
    switch language {
```

```
case french:  
    prefix = frenchHelloPrefix  
case spanish:  
    prefix = spanishHelloPrefix  
default:  
    prefix = englishHelloPrefix  
}  
return  
}
```

A few new concepts:

- In our function signature we have made a named return value (prefix string).
- This will create a variable called prefix in your function.
 - It will be assigned the "zero" value. This depends on the type, for example ints are 0 and for strings it is "".
 - * You can return whatever it's set to by just calling return rather than return prefix.
 - This will display in the Go Doc for your function so it can make the intent of your code clearer.
- default in the switch case will be branched to if none of the other case statements match.
- The function name starts with a lowercase letter. In Go, public functions start with a capital letter and private ones start with a lowercase. We don't want the internals of our algorithm to be exposed to the world, so we made this function private.
- Also, we can group constants in a block instead of declaring them each on their own line. It's a good idea to use a line between sets of related constants for readability.

Wrapping up

Who knew you could get so much out of Hello, world?

By now you should have some understanding of:

Some of Go's syntax around

- Writing tests
- Declaring functions, with arguments and return types
- if, const and switch
- Declaring variables and constants

The TDD process and why the steps are important

- Write a failing test and see it fail so we know we have written a relevant test for our requirements and seen that it produces an easy to understand description of the failure
- Writing the smallest amount of code to make it pass so we know we have working software
- Then refactor, backed with the safety of our tests to ensure we have well-crafted code that is easy to work with

In our case we've gone from Hello() to Hello("name"), to Hello("name", "French") in small, easy to understand steps.

This is of course trivial compared to "real world" software but the principles still stand. TDD is a skill that needs practice to develop, but by breaking problems down into smaller components that you can test, you will have a much easier time writing software.

Integers

You can find all the code for this chapter here

Integers work as you would expect. Let's write an Add function to try things out. Create a test file called adder_test.go and write this code.

Note: Go source files can only have one package per directory. Make sure that your files are organised into their own packages. [Here is a good explanation on this.](#)

Your project directory might look something like this:

```
learnGoWithTests
|
|-> helloworld
|   |- hello.go
|   |- hello_test.go
|
|-> integers
|   |- adder_test.go
|
|- go.mod
|- README.md
```

Write the test first

package integers

```
import "testing"

func TestAdder(t *testing.T) {
    sum := Add(2, 2)
    expected := 4

    if sum != expected {
        t.Errorf("expected '%d' but got '%d'", expected, sum)
    }
}
```

You will notice that we're using %d as our format strings rather than %q. That's because we want it to print an integer rather than a string.

Also note that we are no longer using the main package, instead we've defined a package named integers, as the name suggests this will group functions for working with integers such as Add.

Try and run the test

Run the test go test

Inspect the compilation error

./adder_test.go:6:9: undefined: Add

Write the minimal amount of code for the test to run and check the failing test output

Write enough code to satisfy the compiler and that's all - remember we want to check that our tests fail for the correct reason.

```
package integers
```

```
func Add(x, y int) int {
    return 0
}
```

When you have more than one argument of the same type (in our case two integers) rather than having (x int, y int) you can shorten it to (x, y int).

Now run the tests and we should be happy that the test is correctly reporting what is wrong.

adder_test.go:10: expected '4' but got '0'

If you have noticed we learnt about named return value in the [last](#) section but aren't using the same here. It should generally be used

when the meaning of the result isn't clear from context, in our case it's pretty much clear that Add function will add the parameters. You can refer [this](#) wiki for more details.

Write enough code to make it pass

In the strictest sense of TDD we should now write the minimal amount of code to make the test pass. A pedantic programmer may do this

```
func Add(x, y int) int {  
    return 4  
}
```

Ah hah! Foiled again, TDD is a sham right?

We could write another test, with some different numbers to force that test to fail but that feels like [a game of cat and mouse](#).

Once we're more familiar with Go's syntax I will introduce a technique called "Property Based Testing", which would stop annoying developers and help you find bugs.

For now, let's fix it properly

```
func Add(x, y int) int {  
    return x + y  
}
```

If you re-run the tests they should pass.

Refactor

There's not a lot in the actual code we can really improve on here.

We explored earlier how by naming the return argument it appears in the documentation but also in most developer's text editors.

This is great because it aids the usability of code you are writing. It is preferable that a user can understand the usage of your code by just looking at the type signature and documentation.

You can add documentation to functions with comments, and these will appear in Go Doc just like when you look at the standard library's documentation.

```
// Add takes two integers and returns the sum of them.  
func Add(x, y int) int {  
    return x + y  
}
```

Examples

If you really want to go the extra mile you can make [examples](#). You will find a lot of examples in the documentation of the standard library.

Often code examples that can be found outside the codebase, such as a readme file often become out of date and incorrect compared to the actual code because they don't get checked.

Go examples are executed just like tests so you can be confident examples reflect what the code actually does.

Examples are compiled (and optionally executed) as part of a package's test suite.

As with typical tests, examples are functions that reside in a package's _test.go files. Add the following ExampleAdd function to the adder_test.go file.

```
func ExampleAdd() {
    sum := Add(1, 5)
    fmt.Println(sum)
    // Output: 6
}
```

(If your editor doesn't automatically import packages for you, the compilation step will fail because you will be missing import "fmt" in adder_test.go. It is strongly recommended you research how to have these kind of errors fixed for you automatically in whatever editor you are using.)

If your code changes so that the example is no longer valid, your build will fail.

Running the package's test suite, we can see the example function is executed with no further arrangement from us:

```
$ go test -v
==== RUN TestAdder
--- PASS: TestAdder (0.00s)
==== RUN ExampleAdd
--- PASS: ExampleAdd (0.00s)
```

Please note that the example function will not be executed if you remove the comment // Output: 6. Although the function will be compiled, it won't be executed.

By adding this code the example will appear in the documentation inside godoc, making your code even more accessible.

To try this out, run godoc -http=:6060 and navigate to <http://localhost:6060/pkg/>

Inside here you'll see a list of all the packages and you'll be able to find your example documentation.

If you publish your code with examples to a public URL, you can share the documentation of your code at [pkg.go.dev](#). For example, [here](#) is the finalised API for this chapter. This web interface allows you to search for documentation of standard library packages and third-party packages.

Wrapping up

What we have covered:

- More practice of the TDD workflow
- Integers, addition
- Writing better documentation so users of our code can understand its usage quickly
- Examples of how to use our code, which are checked as part of our tests

Iteration

[You can find all the code for this chapter here](#)

To do stuff repeatedly in Go, you'll need for. In Go there are no while, do, until keywords, you can only use for. Which is a good thing!

Let's write a test for a function that repeats a character 5 times.

There's nothing new so far, so try and write it yourself for practice.

Write the test first

```
package iteration
```

```
import "testing"
```

```
func TestRepeat(t *testing.T) {
    repeated := Repeat("a")
    expected := "aaaaa"

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

Try and run the test

```
./repeat_test.go:6:14: undefined: Repeat
```

Write the minimal amount of code for the test to run and check the failing test output

Keep the discipline! You don't need to know anything new right now to make the test fail properly.

All you need to do right now is enough to make it compile so you can check your test is written well.

package iteration

```
func Repeat(character string) string {  
    return ""  
}
```

Isn't it nice to know you already know enough Go to write tests for some basic problems? This means you can now play with the production code as much as you like and know it's behaving as you'd hope.

```
repeat_test.go:10: expected 'aaaaa' but got ''
```

Write enough code to make it pass

The for syntax is very unremarkable and follows most C-like languages.

```
func Repeat(character string) string {  
    var repeated string  
    for i := 0; i < 5; i++ {  
        repeated = repeated + character  
    }  
    return repeated  
}
```

Unlike other languages like C, Java, or JavaScript there are no parentheses surrounding the three components of the for statement and the braces {} are always required. You might wonder what is happening in the row

var repeated string

as we've been using := so far to declare and initializing variables. However, := is simply [short hand for both steps](#). Here we are declar-

ing a string variable only. Hence, the explicit version. We can also use var to declare functions, as we'll see later on.

Run the test and it should pass.

Additional variants of the for loop are described [here](#).

Refactor

Now it's time to refactor and introduce another construct += assignment operator.

```
const repeatCount = 5

func Repeat(character string) string {
    var repeated string
    for i := 0; i < repeatCount; i++ {
        repeated += character
    }
    return repeated
}
```

+= called "the Add AND assignment operator", adds the right operand to the left operand and assigns the result to left operand. It works with other types like integers.

Benchmarking

Writing [benchmarks](#) in Go is another first-class feature of the language and it is very similar to writing tests.

```
func BenchmarkRepeat(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Repeat("a")
    }
}
```

You'll see the code is very similar to a test.

The testing.B gives you access to the cryptically named b.N.

When the benchmark code is executed, it runs b.N times and measures how long it takes.

The amount of times the code is run shouldn't matter to you, the framework will determine what is a "good" value for that to let you have some decent results.

To run the benchmarks do go test -bench=. (or if you're in Windows Powershell go test -bench=".")

```
goos: darwin
goarch: amd64
pkg: github.com/quii/learn-go-with-tests/for/v4
10000000          136 ns/op
PASS
```

What 136 ns/op means is our function takes on average 136 nanoseconds to run (on my computer). Which is pretty ok! To test this it ran it 10000000 times.

NOTE by default Benchmarks are run sequentially.

Practice exercises

- Change the test so a caller can specify how many times the character is repeated and then fix the code
- Write ExampleRepeat to document your function
- Have a look through the [strings](#) package. Find functions you think could be useful and experiment with them by writing tests like we have here. Investing time learning the standard library will really pay off over time.

Wrapping up

- More TDD practice
- Learned for
- Learned how to write benchmarks

Arrays and slices

You can find all the code for this chapter [here](#)

Arrays allow you to store multiple elements of the same type in a variable in a particular order.

When you have an array, it is very common to have to iterate over them. So let's use [our new-found knowledge of for](#) to make a Sum function. Sum will take an array of numbers and return the total.

Let's use our TDD skills

Write the test first

Create a new folder to work in. Create a new file called sum_test.go and insert the following:

```
package main

import "testing"

func TestSum(t *testing.T) {
    numbers := [5]int{1, 2, 3, 4, 5}

    got := Sum(numbers)
    want := 15

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

Arrays have a fixed capacity which you define when you declare the variable. We can initialize an array in two ways:

- [N]type{value1, value2, ..., valueN} e.g. numbers := [5]int{1, 2, 3, 4, 5}
- [...]type{value1, value2, ..., valueN} e.g. numbers := [...]int{1, 2, 3, 4, 5}

It is sometimes useful to also print the inputs to the function in the error message. Here, we are using the %v placeholder to print the "default" format, which works well for arrays.

[Read more about the format strings](#)

Try to run the test

If you had initialized go mod with go mod init main you will be presented with an error _testmain.go:13:2: cannot import "main". This is because according to common practice, package main will only contain integration of other packages and not unit-testable code and hence Go will not allow you to import a package with name main.

To fix this, you can rename the main module in go.mod to any other name.

Once the above error is fixed, if you run go test the compiler will fail with the familiar ./sum_test.go:10:15: undefined: Sum error. Now we can proceed with writing the actual method to be tested.

Write the minimal amount of code for the test to run and check the failing test output

In sum.go

```
package main

func Sum(numbers [5]int) int {
    return 0
}
```

Your test should now fail with a clear error message

```
sum_test.go:13: got 0 want 15 given, [1 2 3 4 5]
```

Write enough code to make it pass

```
func Sum(numbers [5]int) int {
    sum := 0
    for i := 0; i < 5; i++ {
        sum += numbers[i]
    }
    return sum
}
```

To get the value out of an array at a particular index, just use array[index] syntax. In this case, we are using for to iterate 5 times to work through the array and add each item onto sum.

Refactor

Let's introduce [range](#) to help clean up our code

```
func Sum(numbers [5]int) int {
    sum := 0
    for _, number := range numbers {
        sum += number
    }
    return sum
}
```

[range](#) lets you iterate over an array. On each iteration, [range](#) returns two values - the index and the value. We are choosing to ignore the index value by using [blank identifier](#).

Arrays and their type

An interesting property of arrays is that the size is encoded in its type. If you try to pass an [4]int into a function that expects [5]int, it won't compile. They are different types so it's just the same as trying to pass a string into a function that wants an int.

You may be thinking it's quite cumbersome that arrays have a fixed length, and most of the time you probably won't be using them!

Go has slices which do not encode the size of the collection and instead can have any size.

The next requirement will be to sum collections of varying sizes.

Write the test first

We will now use the [slice type](#) which allows us to have collections of any size. The syntax is very similar to arrays, you just omit the size when declaring them

```
mySlice := []int{1,2,3} rather than myArray := [3]int{1,2,3}
func TestSum(t *testing.T) {

    t.Run("collection of 5 numbers", func(t *testing.T) {
        numbers := [5]int{1, 2, 3, 4, 5}

        got := Sum(numbers)
        want := 15

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

    t.Run("collection of any size", func(t *testing.T) {
        numbers := []int{1, 2, 3}

        got := Sum(numbers)
        want := 6

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

Try and run the test

This does not compile

```
./sum_test.go:22:13: cannot use numbers (type []int) as type [5]int  
in argument to Sum
```

Write the minimal amount of code for the test to run and check the failing test output

The problem here is we can either

- Break the existing API by changing the argument to Sum to be a slice rather than an array. When we do this, we will potentially ruin someone's day because our other test will no longer compile!
- Create a new function

In our case, no one else is using our function, so rather than having two functions to maintain, let's have just one.

```
func Sum(numbers []int) int {  
    sum := 0  
    for _, number := range numbers {  
        sum += number  
    }  
    return sum  
}
```

If you try to run the tests they will still not compile, you will have to change the first test to pass in a slice rather than an array.

Write enough code to make it pass

It turns out that fixing the compiler problems were all we need to do here and the tests pass!

Refactor

We already refactored Sum - all we did was replace arrays with slices, so no extra changes are required. Remember that we must not neglect our test code in the refactoring stage - we can further improve our Sum tests.

```
func TestSum(t *testing.T) {  
  
    t.Run("collection of 5 numbers", func(t *testing.T) {  
        numbers := []int{1, 2, 3, 4, 5}
```

```
got := Sum(numbers)
want := 15

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

t.Run("collection of any size", func(t *testing.T) {
    numbers := []int{1, 2, 3}

    got := Sum(numbers)
    want := 6

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

It is important to question the value of your tests. It should not be a goal to have as many tests as possible, but rather to have as much confidence as possible in your code base. Having too many tests can turn in to a real problem and it just adds more overhead in maintenance. **Every test has a cost**.

In our case, you can see that having two tests for this function is redundant. If it works for a slice of one size it's very likely it'll work for a slice of any size (within reason).

Go's built-in testing toolkit features a [coverage tool](#). Whilst striving for 100% coverage should not be your end goal, the coverage tool can help identify areas of your code not covered by tests. If you have been strict with TDD, it's quite likely you'll have close to 100% coverage anyway.

Try running

```
go test -cover
```

You should see

```
PASS
coverage: 100.0% of statements
```

Now delete one of the tests and check the coverage again.

Now that we are happy we have a well-tested function you should

commit your great work before taking on the next challenge.

We need a new function called SumAll which will take a varying number of slices, returning a new slice containing the totals for each slice passed in.

For example

SumAll([]int{1,2}, []int{0,9}) would return []int{3, 9}

or

SumAll([]int{1,1,1}) would return []int{3}

Write the test first

```
func TestSumAll(t *testing.T) {  
    got := SumAll([]int{1, 2}, []int{0, 9})  
    want := []int{3, 9}  
  
    if got != want {  
        t.Errorf("got %v want %v", got, want)  
    }  
}
```

Try and run the test

./sum_test.go:23:9: undefined: SumAll

Write the minimal amount of code for the test to run and check the failing test output

We need to define SumAll according to what our test wants.

Go can let you write [variadic functions](#) that can take a variable number of arguments.

```
func SumAll(numbersToSum ...[]int) []int {  
    return nil  
}
```

This is valid, but our tests still won't compile!

./sum_test.go:26:9: invalid operation: got != want (slice can only be compared to nil)

Go does not let you use equality operators with slices. You could write a function to iterate over each got and want slice and check their

values but for convenience sake, we can use `reflect.DeepEqual` which is useful for seeing if any two variables are the same.

```
func TestSumAll(t *testing.T) {  
  
    got := SumAll([]int{1, 2}, []int{0, 9})  
    want := []int{3, 9}  
  
    if !reflect.DeepEqual(got, want) {  
        t.Errorf("got %v want %v", got, want)  
    }  
}
```

(make sure you import `reflect` in the top of your file to have access to `DeepEqual`)

It's important to note that `reflect.DeepEqual` is not "type safe" - the code will compile even if you did something a bit silly. To see this in action, temporarily change the test to:

```
func TestSumAll(t *testing.T) {  
  
    got := SumAll([]int{1, 2}, []int{0, 9})  
    want := "bob"  
  
    if !reflect.DeepEqual(got, want) {  
        t.Errorf("got %v want %v", got, want)  
    }  
}
```

What we have done here is try to compare a slice with a string. This makes no sense, but the test compiles! So while using `reflect.DeepEqual` is a convenient way of comparing slices (and other things) you must be careful when using it.

Change the test back again and run it. You should have test output like the following

```
sum_test.go:30: got [] want [3 9]
```

Write enough code to make it pass

What we need to do is iterate over the varargs, calculate the sum using our existing `Sum` function, then add it to the slice we will return

```
func SumAll(numbersToSum ...[]int) []int {  
    lengthOfNumbers := len(numbersToSum)  
    sums := make([]int, lengthOfNumbers)
```

```
for i, numbers := range numbersToSum {
    sums[i] = Sum(numbers)
}

return sums
}
```

Lots of new things to learn!

There's a new way to create a slice. make allows you to create a slice with a starting capacity of the len of the numbersToSum we need to work through.

You can index slices like arrays with mySlice[N] to get the value out or assign it a new value with =

The tests should now pass.

Refactor

As mentioned, slices have a capacity. If you have a slice with a capacity of 2 and try to do mySlice[10] = 1 you will get a runtime error.

However, you can use the append function which takes a slice and a new value, then returns a new slice with all the items in it.

```
func SumAll(numbersToSum ...[]int) []int {
    var sums []int
    for _, numbers := range numbersToSum {
        sums = append(sums, Sum(numbers))
    }

    return sums
}
```

In this implementation, we are worrying less about capacity. We start with an empty slice sums and append to it the result of Sum as we work through the varargs.

Our next requirement is to change SumAll to SumAllTails, where it will calculate the totals of the "tails" of each slice. The tail of a collection is all items in the collection except the first one (the "head").

Write the test first

```
func TestSumAllTails(t *testing.T) {
    got := SumAllTails([]int{1, 2}, []int{0, 9})
    want := []int{2, 9}
```

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

Try and run the test

```
./sum_test.go:26:9: undefined: SumAllTails
```

Write the minimal amount of code for the test to run and check the failing test output

Rename the function to SumAllTails and re-run the test

```
sum_test.go:30: got [3 9] want [2 9]
```

Write enough code to make it pass

```
func SumAllTails(numbersToSum ...[]int) []int {
    var sums []int
    for _, numbers := range numbersToSum {
        tail := numbers[1:]
        sums = append(sums, Sum(tail))
    }

    return sums
}
```

Slices can be sliced! The syntax is slice[low:high]. If you omit the value on one of the sides of the : it captures everything to that side of it. In our case, we are saying "take from 1 to the end" with numbers[1:]. You may wish to spend some time writing other tests around slices and experiment with the slice operator to get more familiar with it.

Refactor

Not a lot to refactor this time.

What do you think would happen if you passed in an empty slice into our function? What is the "tail" of an empty slice? What happens when you tell Go to capture all elements from myEmptySlice[1:]?

Write the test first

```
func TestSumAllTails(t *testing.T) {  
  
    t.Run("make the sums of some slices", func(t *testing.T) {  
        got := SumAllTails([]int{1, 2}, []int{0, 9})  
        want := []int{2, 9}  
  
        if !reflect.DeepEqual(got, want) {  
            t.Errorf("got %v want %v", got, want)  
        }  
    })  
  
    t.Run("safely sum empty slices", func(t *testing.T) {  
        got := SumAllTails([]int{}, []int{3, 4, 5})  
        want := []int{0, 9}  
  
        if !reflect.DeepEqual(got, want) {  
            t.Errorf("got %v want %v", got, want)  
        }  
    })  
}
```

Try and run the test

```
panic: runtime error: slice bounds out of range [recovered]  
    panic: runtime error: slice bounds out of range
```

Oh no! It's important to note that while the test has compiled, it has a runtime error.

Compile time errors are our friend because they help us write software that works,
runtime errors are our enemies because they affect our users.

Write enough code to make it pass

```
func SumAllTails(numbersToSum ...[]int) []int {  
    var sums []int  
    for _, numbers := range numbersToSum {  
        if len(numbers) == 0 {  
            sums = append(sums, 0)  
        } else {  
            tail := numbers[1:]  
            sums = append(sums, Sum(tail))  
        }  
    }  
    return sums  
}
```

```
    }
}

return sums
}
```

Refactor

Our tests have some repeated code around the assertions again, so let's extract those into a function.

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

    checkSums := func(t testing.TB, got, want []int) {
        t.Helper()
        if !reflect.DeepEqual(got, want) {
            t.Errorf("got %v want %v", got, want)
        }
    }

    t.Run("make the sums of tails of", func(t *testing.T) {
        got := SumAllTails([]int{1, 2}, []int{0, 9})
        want := []int{2, 9}
        checkSums(t, got, want)
    })

    t.Run("safely sum empty slices", func(t *testing.T) {
        got := SumAllTails([]int{}, []int{3, 4, 5})
        want := []int{0, 9}
        checkSums(t, got, want)
    })
}
```

We could've created a new function `checkSums` like we normally do, but in this case, we're showing a new technique, assigning a function to a variable. It might look strange but, it's no different to assigning a variable to a string, or an int, functions in effect are values too.

It's not shown here, but this technique can be useful when you want to bind a function to other local variables in "scope" (e.g between some `{}`). It also allows you to reduce the surface area of your API.

By defining this function inside the test, it cannot be used by other functions in this package. Hiding variables and functions that don't need to be exported is an important design consideration.

A handy side-effect of this is this adds a little type-safety to our code.

If a developer mistakenly adds a new test with `checkSums(t, got, "dave")` the compiler will stop them in their tracks.

```
$ go test  
./sum_test.go:52:21: cannot use "dave" (type string) as type []int in argument to checkSums
```

Wrapping up

We have covered

- Arrays
- Slices
 - The various ways to make them
 - How they have a fixed capacity but you can create new slices from old ones using `append`
 - How to slice, slices!
- `len` to get the length of an array or slice
- Test coverage tool
- `reflect.DeepEqual` and why it's useful but can reduce the type-safety of your code

We've used slices and arrays with integers but they work with any other type too, including arrays/slices themselves. So you can declare a variable of `[]string` if you need to.

[Check out the Go blog post on slices](#) for an in-depth look into slices. Try writing more tests to solidify what you learn from reading it.

Another handy way to experiment with Go other than writing tests is the Go playground. You can try most things out and you can easily share your code if you need to ask questions. [I have made a go playground with a slice in it for you to experiment with.](#)

[Here is an example](#) of slicing an array and how changing the slice affects the original array; but a "copy" of the slice will not affect the original array. [Another example](#) of why it's a good idea to make a copy of a slice after slicing a very large slice.

Structs, methods & interfaces

[You can find all the code for this chapter here](#)

Suppose that we need some geometry code to calculate the perimeter of a rectangle given a height and width. We can write a `Perimeter(width float64, height float64)` function, where `float64` is for floating-point numbers like 123.45.

The TDD cycle should be pretty familiar to you by now.

Write the test first

```
func TestPerimeter(t *testing.T) {
    got := Perimeter(10.0, 10.0)
    want := 40.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

Notice the new format string? The f is for our float64 and the .2 means print 2 decimal places.

Try to run the test

```
./shapes_test.go:6:9: undefined: Perimeter
```

Write the minimal amount of code for the test to run and check the failing test output

```
func Perimeter(width float64, height float64) float64 {
    return 0
}
```

Results in shapes_test.go:10: got 0.00 want 40.00.

Write enough code to make it pass

```
func Perimeter(width float64, height float64) float64 {
    return 2 * (width + height)
}
```

So far, so easy. Now let's create a function called Area(width, height float64) which returns the area of a rectangle.

Try to do it yourself, following the TDD cycle.

You should end up with tests like this

```
func TestPerimeter(t *testing.T) {
    got := Perimeter(10.0, 10.0)
    want := 40.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

```
func TestArea(t *testing.T) {
    got := Area(12.0, 6.0)
    want := 72.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

And code like this

```
func Perimeter(width float64, height float64) float64 {
    return 2 * (width + height)
}

func Area(width float64, height float64) float64 {
    return width * height
}
```

Refactor

Our code does the job, but it doesn't contain anything explicit about rectangles. An unwary developer might try to supply the width and height of a triangle to these functions without realising they will return the wrong answer.

We could just give the functions more specific names like RectangleArea. A neater solution is to define our own type called Rectangle which encapsulates this concept for us.

We can create a simple type using a **struct**. A **struct** is just a named collection of fields where you can store data.

Declare a struct like this

```
type Rectangle struct {
    Width float64
    Height float64
}
```

Now let's refactor the tests to use Rectangle instead of plain float64s.

```
func TestPerimeter(t *testing.T) {
    rectangle := Rectangle{10.0, 10.0}
    got := Perimeter(rectangle)
    want := 40.0

    if got != want {
```

```
        t.Errorf("got %.2f want %.2f", got, want)
    }
}

func TestArea(t *testing.T) {
    rectangle := Rectangle{12.0, 6.0}
    got := Area(rectangle)
    want := 72.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

Remember to run your tests before attempting to fix. The tests should show a helpful error like

```
./shapes_test.go:7:18: not enough arguments in call to Perimeter
    have (Rectangle)
    want (float64, float64)
```

You can access the fields of a struct with the syntax of myStruct.field.

Change the two functions to fix the test.

```
func Perimeter(rectangle Rectangle) float64 {
    return 2 * (rectangle.Width + rectangle.Height)
}

func Area(rectangle Rectangle) float64 {
    return rectangle.Width * rectangle.Height
}
```

I hope you'll agree that passing a Rectangle to a function conveys our intent more clearly, but there are more benefits of using structs that we will cover later.

Our next requirement is to write an Area function for circles.

Write the test first

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

    t.Run("rectangles", func(t *testing.T) {
        rectangle := Rectangle{12, 6}
        got := Area(rectangle)
        want := 72.0

        if got != want {
```

```
        t.Errorf("got %g want %g", got, want)
    }
}

t.Run("circles", func(t *testing.T) {
    circle := Circle{10}
    got := Area(circle)
    want := 314.1592653589793

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

As you can see, the f has been replaced by g, with good reason. Use of g will print a more precise decimal number in the error message ([fmt options](#)). For example, using a radius of 1.5 in a circle area calculation, f would show 7.068583 whereas g would show 7.0685834705770345.

Try to run the test

```
./shapes_test.go:28:13: undefined: Circle
```

Write the minimal amount of code for the test to run and check the failing test output

We need to define our Circle type.

```
type Circle struct {
    Radius float64
}
```

Now try to run the tests again

```
./shapes_test.go:29:14: cannot use circle (type Circle) as type
Rectangle in argument to Area
```

Some programming languages allow you to do something like this:

```
func Area(circle Circle) float64 {}
func Area(rectangle Rectangle) float64 {}
```

But you cannot in Go

```
./shapes.go:20:32: Area redeclared in this block
```

We have two choices:

-
- You can have functions with the same name declared in different packages. So we could create our Area(Circle) in a new package, but that feels overkill here.
 - We can define **methods** on our newly defined types instead.

What are methods?

So far we have only been writing functions but we have been using some methods. When we call t.Errorif we are calling the method Errorif on the instance of our t (testing.T).

A method is a function with a receiver. A method declaration binds an identifier, the method name, to a method, and associates the method with the receiver's base type.

Methods are very similar to functions but they are called by invoking them on an instance of a particular type. Where you can just call functions wherever you like, such as Area(rectangle) you can only call methods on "things".

An example will help so let's change our tests first to call methods instead and then fix the code.

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

    t.Run("rectangles", func(t *testing.T) {
        rectangle := Rectangle{12, 6}
        got := rectangle.Area()
        want := 72.0

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

    t.Run("circles", func(t *testing.T) {
        circle := Circle{10}
        got := circle.Area()
        want := 314.1592653589793

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

If we try to run the tests, we get

```
./shapes_test.go:19:19: rectangle.Area undefined (type Rectangle has no field or method Area)
./shapes_test.go:29:16: circle.Area undefined (type Circle has no field or method Area)
```

type Circle has no field or method Area

I would like to reiterate how great the compiler is here. It is so important to take the time to slowly read the error messages you get, it will help you in the long run.

Write the minimal amount of code for the test to run and check the failing test output

Let's add some methods to our types

```
type Rectangle struct {
    Width float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return 0
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return 0
}
```

The syntax for declaring methods is almost the same as functions and that's because they're so similar. The only difference is the syntax of the method receiver func (receiverName ReceiverType) MethodName(args).

When your method is called on a variable of that type, you get your reference to its data via the receiverName variable. In many other programming languages this is done implicitly and you access the receiver via this.

It is a convention in Go to have the receiver variable be the first letter of the type.

r Rectangle

If you try to re-run the tests they should now compile and give you some failing output.

Write enough code to make it pass

Now let's make our rectangle tests pass by fixing our new method

```
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

If you re-run the tests the rectangle tests should be passing but circle should still be failing.

To make circle's Area function pass we will borrow the Pi constant from the math package (remember to import it).

```
func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

Refactor

There is some duplication in our tests.

All we want to do is take a collection of shapes, call the Area() method on them and then check the result.

We want to be able to write some kind of checkArea function that we can pass both Rectangles and Circles to, but fail to compile if we try to pass in something that isn't a shape.

With Go, we can codify this intent with **interfaces**.

[Interfaces](#) are a very powerful concept in statically typed languages like Go because they allow you to make functions that can be used with different types and create highly-decoupled code whilst still maintaining type-safety.

Let's introduce this by refactoring our tests.

```
func TestArea(t *testing.T) {
    checkArea := func(t testing.TB, shape Shape, want float64) {
        t.Helper()
        got := shape.Area()
        if got != want {
            t.Errorf("got %g want %g", got, want)
        }
    }

    t.Run("rectangles", func(t *testing.T) {
        rectangle := Rectangle{12, 6}
        checkArea(t, rectangle, 72.0)
    })
}
```

```
    })  
  
    t.Run("circles", func(t *testing.T) {  
        circle := Circle{10}  
        checkArea(t, circle, 314.1592653589793)  
    })  
  
}
```

We are creating a helper function like we have in other exercises but this time we are asking for a Shape to be passed in. If we try to call this with something that isn't a shape, then it will not compile.

How does something become a shape? We just tell Go what a Shape is using an interface declaration

```
type Shape interface {  
    Area() float64  
}
```

We're creating a new type just like we did with Rectangle and Circle but this time it is an interface rather than a struct.

Once you add this to the code, the tests will pass.

Wait, what?

This is quite different to interfaces in most other programming languages. Normally you have to write code to say My type Foo implements interface Bar.

But in our case

- Rectangle has a method called Area that returns a float64 so it satisfies the Shape interface
- Circle has a method called Area that returns a float64 so it satisfies the Shape interface
- string does not have such a method, so it doesn't satisfy the interface
- etc.

In Go **interface resolution is implicit**. If the type you pass in matches what the interface is asking for, it will compile.

Decoupling

Notice how our helper does not need to concern itself with whether the shape is a Rectangle or a Circle or a Triangle. By declaring an

interface, the helper is decoupled from the concrete types and only has the method it needs to do its job.

This kind of approach of using interfaces to declare **only what you need** is very important in software design and will be covered in more detail in later sections.

Further refactoring

Now that you have some understanding of structs we can introduce "table driven tests".

[Table driven tests](#) are useful when you want to build a list of test cases that can be tested in the same manner.

```
func TestArea(t *testing.T) {  
  
    areaTests := []struct {  
        shape Shape  
        want float64  
    }{  
        {Rectangle{12, 6}, 72.0},  
        {Circle{10}, 314.1592653589793},  
    }  
  
    for _, tt := range areaTests {  
        got := tt.shape.Area()  
        if got != tt.want {  
            t.Errorf("got %g want %g", got, tt.want)  
        }  
    }  
}
```

The only new syntax here is creating an "anonymous struct", areaTests. We are declaring a slice of structs by using []struct with two fields, the shape and the want. Then we fill the slice with cases.

We then iterate over them just like we do any other slice, using the struct fields to run our tests.

You can see how it would be very easy for a developer to introduce a new shape, implement Area and then add it to the test cases. In addition, if a bug is found with Area it is very easy to add a new test case to exercise it before fixing it.

Table driven tests can be a great item in your toolbox, but be sure that you have a need for the extra noise in the tests. They are a great fit when you wish to test various implementations of an interface, or

if the data being passed in to a function has lots of different requirements that need testing.

Let's demonstrate all this by adding another shape and testing it; a triangle.

Write the test first

Adding a new test for our new shape is very easy. Just add {Triangle{12, 6}, 36.0}, to our list.

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

    areaTests := []struct {
        shape Shape
        want float64
    }{
        {Rectangle{12, 6}, 72.0},
        {Circle{10}, 314.1592653589793},
        {Triangle{12, 6}, 36.0},
    }

    for _, tt := range areaTests {
        got := tt.shape.Area()
        if got != tt.want {
            t.Errorf("got %g want %g", got, tt.want)
        }
    }
}
```

Try to run the test

Remember, keep trying to run the test and let the compiler guide you toward a solution.

Write the minimal amount of code for the test to run and check the failing test output

```
./shapes_test.go:25:4: undefined: Triangle
```

We have not defined Triangle yet

```
type Triangle struct {
    Base float64
    Height float64
}
```

Try again

```
./shapes_test.go:25:8: cannot use Triangle literal (type Triangle) as type Shape in field value:  
    Triangle does not implement Shape (missing Area method)
```

It's telling us we cannot use a Triangle as a shape because it does not have an Area() method, so add an empty implementation to get the test working

```
func (t Triangle) Area() float64 {  
    return 0  
}
```

Finally the code compiles and we get our error

```
shapes_test.go:31: got 0.00 want 36.00
```

Write enough code to make it pass

```
func (t Triangle) Area() float64 {  
    return (t.Base * t.Height) * 0.5  
}
```

And our tests pass!

Refactor

Again, the implementation is fine but our tests could do with some improvement.

When you scan this

```
{Rectangle{12, 6}, 72.0},  
{Circle{10}, 314.1592653589793},  
{Triangle{12, 6}, 36.0},
```

It's not immediately clear what all the numbers represent and you should be aiming for your tests to be easily understood.

So far you've only been shown syntax for creating instances of structs MyStruct{val1, val2} but you can optionally name the fields.

Let's see what it looks like

```
{shape: Rectangle{Width: 12, Height: 6}, want: 72.0},  
{shape: Circle{Radius: 10}, want: 314.1592653589793},  
{shape: Triangle{Base: 12, Height: 6}, want: 36.0},
```

In [Test-Driven Development by Example](#) Kent Beck refactors some tests to a point and asserts:

The test speaks to us more clearly, as if it were an assertion
of truth, **not a sequence of operations**

(emphasis in the quote is mine)

Now our tests - rather, the list of test cases - make assertions of truth about shapes and their areas.

Make sure your test output is helpful

Remember earlier when we were implementing Triangle and we had the failing test? It printed shapes_test.go:31: got 0.00 want 36.00.

We knew this was in relation to Triangle because we were just working with it. But what if a bug slipped in to the system in one of 20 cases in the table? How would a developer know which case failed? This is not a great experience for the developer, they will have to manually look through the cases to find out which case actually failed.

We can change our error message into %#v got %g want %g. The %#v format string will print out our struct with the values in its field, so the developer can see at a glance the properties that are being tested.

To increase the readability of our test cases further, we can rename the want field into something more descriptive like hasArea.

One final tip with table driven tests is to use t.Run and to name the test cases.

By wrapping each case in a t.Run you will have clearer test output on failures as it will print the name of the case

```
--- FAIL: TestArea (0.00s)
    --- FAIL: TestArea/Rectangle (0.00s)
        shapes_test.go:33: main.Rectangle{Width:12, Height:6} got 72.00 want 72.10
```

And you can run specific tests within your table with go test -run TestArea/Rectangle.

Here is our final test code which captures this

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

    areaTests := []struct {
        name   string
        shape  Shape
        hasArea float64
    }{
        {name: "Rectangle", shape: Rectangle{Width: 12, Height: 6}, hasArea: 72.0},
        {name: "Circle", shape: Circle{Radius: 10}, hasArea: 314.1592653589793},
    }
}
```

```
{name: "Triangle", shape: Triangle{Base: 12, Height: 6}, hasArea: 36.0},  
}  
  
for _, tt := range areaTests {  
    // using tt.name from the case to use it as the `t.Run` test name  
    t.Run(tt.name, func(t *testing.T) {  
        got := tt.shape.Area()  
        if got != tt.hasArea {  
            t.Errorf("%#v got %g want %g", tt.shape, got, tt.hasArea)  
        }  
    })  
}  
}
```

Wrapping up

This was more TDD practice, iterating over our solutions to basic mathematic problems and learning new language features motivated by our tests.

- Declaring structs to create your own data types which lets you bundle related data together and make the intent of your code clearer
- Declaring interfaces so you can define functions that can be used by different types ([parametric polymorphism](#))
- Adding methods so you can add functionality to your data types and so you can implement interfaces
- Table driven tests to make your assertions clearer and your test suites easier to extend & maintain

This was an important chapter because we are now starting to define our own types. In statically typed languages like Go, being able to design your own types is essential for building software that is easy to understand, to piece together and to test.

Interfaces are a great tool for hiding complexity away from other parts of the system. In our case our test helper code did not need to know the exact shape it was asserting on, only how to "ask" for its area.

As you become more familiar with Go you will start to see the real strength of interfaces and the standard library. You'll learn about interfaces defined in the standard library that are used everywhere and by implementing them against your own types, you can very quickly re-use a lot of great functionality.

Pointers & errors

You can find all the code for this chapter here

We learned about structs in the last section which let us capture a number of values related around a concept.

At some point you may wish to use structs to manage state, exposing methods to let users change the state in a way that you can control.

Fintech loves Go and uhhh bitcoins? So let's show what an amazing banking system we can make.

Let's make a Wallet struct which lets us deposit Bitcoin.

Write the test first

```
func TestWallet(t *testing.T) {  
    wallet := Wallet{}  
  
    wallet.Deposit(10)  
  
    got := wallet.Balance()  
    want := 10  
  
    if got != want {  
        t.Errorf("got %d want %d", got, want)  
    }  
}
```

In the [previous example](#) we accessed fields directly with the field name, however in our very secure wallet we don't want to expose our inner state to the rest of the world. We want to control access via methods.

Try to run the test

```
./wallet_test.go:7:12: undefined: Wallet
```

Write the minimal amount of code for the test to run and check the failing test output

The compiler doesn't know what a Wallet is so let's tell it.

```
type Wallet struct{}
```

Now we've made our wallet, try and run the test again

```
./wallet_test.go:9:8: wallet.Deposit undefined (type Wallet has no field or method Deposit)
./wallet_test.go:11:15: wallet.Balance undefined (type Wallet has no field or method Balance)
```

We need to define these methods.

Remember to only do enough to make the tests run. We need to make sure our test fails correctly with a clear error message.

```
func (w Wallet) Deposit(amount int) {
}

func (w Wallet) Balance() int {
    return 0
}
```

If this syntax is unfamiliar go back and read the structs section.

The tests should now compile and run

```
wallet_test.go:15: got 0 want 10
```

Write enough code to make it pass

We will need some kind of balance variable in our struct to store the state

```
type Wallet struct {
    balance int
}
```

In Go if a symbol (variables, types, functions et al) starts with a lowercase symbol then it is private outside the package it's defined in.

In our case we want our methods to be able to manipulate this value, but no one else.

Remember we can access the internal balance field in the struct using the "receiver" variable.

```
func (w Wallet) Deposit(amount int) {
    w.balance += amount
}

func (w Wallet) Balance() int {
    return w.balance
}
```

With our career in fintech secured, run the test suite and bask in the passing test

```
wallet_test.go:15: got 0 want 10
```

That's not quite right

Well this is confusing, our code looks like it should work. We add the new amount onto our balance and then the balance method should return the current state of it.

In Go, **when you call a function or a method the arguments are copied.**

When calling func (w Wallet) Deposit(amount int) the w is a copy of whatever we called the method from.

Without getting too computer-sciency, when you create a value - like a wallet, it is stored somewhere in memory. You can find out what the address of that bit of memory with &myVal.

Experiment by adding some prints to your code

```
func TestWallet(t *testing.T) {  
  
    wallet := Wallet{}  
  
    wallet.Deposit(10)  
  
    got := wallet.Balance()  
  
    fmt.Printf("address of balance in test is %v \n", &wallet.balance)  
  
    want := 10  
  
    if got != want {  
        t.Errorf("got %d want %d", got, want)  
    }  
}  
  
func (w Wallet) Deposit(amount int) {  
    fmt.Printf("address of balance in Deposit is %v \n", &w.balance)  
    w.balance += amount  
}
```

The \n escape character prints a new line after outputting the memory address. We get the pointer (memory address) of something by placing an & character at the beginning of the symbol.

Now re-run the test

```
address of balance in Deposit is 0xc420012268  
address of balance in test is 0xc420012260
```

You can see that the addresses of the two balances are different. So when we change the value of the balance inside the code, we are

working on a copy of what came from the test. Therefore the balance in the test is unchanged.

We can fix this with pointers. [Pointers](#) let us point to some values and then let us change them. So rather than taking a copy of the whole Wallet, we instead take a pointer to that wallet so that we can change the original values within it.

```
func (w *Wallet) Deposit(amount int) {
    w.balance += amount
}

func (w *Wallet) Balance() int {
    return w.balance
}
```

The difference is the receiver type is `*Wallet` rather than `Wallet` which you can read as "a pointer to a wallet".

Try and re-run the tests and they should pass.

Now you might wonder, why did they pass? We didn't dereference the pointer in the function, like so:

```
func (w *Wallet) Balance() int {
    return (*w).balance
}
```

and seemingly addressed the object directly. In fact, the code above using `(*w)` is absolutely valid. However, the makers of Go deemed this notation cumbersome, so the language permits us to write `w.balance`, without an explicit dereference. These pointers to structs even have their own name: struct pointers and they are [automatically dereferenced](#).

Technically you do not need to change `Balance` to use a pointer receiver as taking a copy of the balance is fine. However, by convention you should keep your method receiver types the same for consistency.

Refactor

We said we were making a Bitcoin wallet but we have not mentioned them so far. We've been using `int` because they're a good type for counting things!

It seems a bit overkill to create a struct for this. `int` is fine in terms of the way it works but it's not descriptive.

Go lets you create new types from existing ones.

The syntax is type MyName OriginalType

```
type Bitcoin int

type Wallet struct {
    balance Bitcoin
}

func (w *Wallet) Deposit(amount Bitcoin) {
    w.balance += amount
}

func (w *Wallet) Balance() Bitcoin {
    return w.balance
}

func TestWallet(t *testing.T) {

    wallet := Wallet{}

    wallet.Deposit(Bitcoin(10))

    got := wallet.Balance()

    want := Bitcoin(10)

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

To make Bitcoin you just use the syntax Bitcoin(999).

By doing this we're making a new type and we can declare methods on them. This can be very useful when you want to add some domain specific functionality on top of existing types.

Let's implement `Stringer` on Bitcoin

```
type Stringer interface {
    String() string
}
```

This interface is defined in the `fmt` package and lets you define how your type is printed when used with the `%s` format string in prints.

```
func (b Bitcoin) String() string {
    return fmt.Sprintf("%d BTC", b)
}
```

As you can see, the syntax for creating a method on a type declaration is the same as it is on a struct.

Next we need to update our test format strings so they will use String() instead.

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

To see this in action, deliberately break the test so we can see it

```
wallet_test.go:18: got 10 BTC want 20 BTC
```

This makes it clearer what's going on in our test.

The next requirement is for a Withdraw function.

Write the test first

Pretty much the opposite of Deposit()

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

    t.Run("deposit", func(t *testing.T) {
        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        got := wallet.Balance()

        want := Bitcoin(10)

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

    t.Run("withdraw", func(t *testing.T) {
        wallet := Wallet{balance: Bitcoin(20)}

        wallet.Withdraw(Bitcoin(10))

        got := wallet.Balance()

        want := Bitcoin(10)

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

```
    }
  })
}
```

Try to run the test

```
./wallet_test.go:26:9: wallet.Withdraw undefined (type Wallet has no
field or method Withdraw)
```

Write the minimal amount of code for the test to run and check the failing test output

```
func (w *Wallet) Withdraw(amount Bitcoin) {
}
wallet_test.go:33: got 20 BTC want 10 BTC
```

Write enough code to make it pass

```
func (w *Wallet) Withdraw(amount Bitcoin) {
    w.balance -= amount
}
```

Refactor

There's some duplication in our tests, lets refactor that out.

```
func TestWallet(t *testing.T) {
    assertBalance := func(t testing.TB, wallet Wallet, want Bitcoin) {
        t.Helper()
        got := wallet.Balance()

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

    t.Run("deposit", func(t *testing.T) {
        wallet := Wallet{}
        wallet.Deposit(Bitcoin(10))
        assertBalance(t, wallet, Bitcoin(10))
    })
}
```

```
t.Run("withdraw", func(t *testing.T) {
    wallet := Wallet{balance: Bitcoin(20)}
    wallet.Withdraw(Bitcoin(10))
    assertBalance(t, wallet, Bitcoin(10))
})

}
```

What should happen if you try to Withdraw more than is left in the account? For now, our requirement is to assume there is not an over-draft facility.

How do we signal a problem when using Withdraw?

In Go, if you want to indicate an error it is idiomatic for your function to return an err for the caller to check and act on.

Let's try this out in a test.

Write the test first

```
t.Run("withdraw insufficient funds", func(t *testing.T) {
    startingBalance := Bitcoin(20)
    wallet := Wallet{startingBalance}
    err := wallet.Withdraw(Bitcoin(100))

    assertBalance(t, wallet, startingBalance)

    if err == nil {
        t.Error("wanted an error but didn't get one")
    }
})
```

We want Withdraw to return an error if you try to take out more than you have and the balance should stay the same.

We then check an error has returned by failing the test if it is nil.

nil is synonymous with null from other programming languages. Errors can be nil because the return type of Withdraw will be error, which is an interface. If you see a function that takes arguments or returns values that are interfaces, they can be nullable.

Like null if you try to access a value that is nil it will throw a **runtime panic**. This is bad! You should make sure that you check for nils.

Try and run the test

```
./wallet_test.go:31:25: wallet.Withdraw(Bitcoin(100)) used as value
```

The wording is perhaps a little unclear, but our previous intent with Withdraw was just to call it, it will never return a value. To make this compile we will need to change it so it has a return type.

Write the minimal amount of code for the test to run and check the failing test output

```
func (w *Wallet) Withdraw(amount Bitcoin) error {
    w.balance -= amount
    return nil
}
```

Again, it is very important to just write enough code to satisfy the compiler. We correct our Withdraw method to return error and for now we have to return something so let's just return nil.

Write enough code to make it pass

```
func (w *Wallet) Withdraw(amount Bitcoin) error {
    if amount > w.balance {
        return errors.New("oh no")
    }

    w.balance -= amount
    return nil
}
```

Remember to import errors into your code.

errors.New creates a new error with a message of your choosing.

Refactor

Let's make a quick test helper for our error check to improve the test's readability

```
assertError := func(t testing.TB, err error) {
    t.Helper()
    if err == nil {
        t.Error("wanted an error but didn't get one")
    }
}
```

And in our test

```
t.Run("withdraw insufficient funds", func(t *testing.T) {
    startingBalance := Bitcoin(20)
```

```
wallet := Wallet{startingBalance}
err := wallet.Withdraw(Bitcoin(100))

assertError(t, err)
assertBalance(t, wallet, startingBalance)
})
```

Hopefully when returning an error of "oh no" you were thinking that we might iterate on that because it doesn't seem that useful to return.

Assuming that the error ultimately gets returned to the user, let's update our test to assert on some kind of error message rather than just the existence of an error.

Write the test first

Update our helper for a string to compare against.

```
assertError := func(t testing.TB, got error, want string) {
    t.Helper()
    if got == nil {
        t.Fatal("didn't get an error but wanted one")
    }

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

And then update the caller

```
t.Run("withdraw insufficient funds", func(t *testing.T) {
    startingBalance := Bitcoin(20)
    wallet := Wallet{startingBalance}
    err := wallet.Withdraw(Bitcoin(100))

    assertError(t, err, "cannot withdraw, insufficient funds")
    assertBalance(t, wallet, startingBalance)
})
```

We've introduced `t.Fatal` which will stop the test if it is called. This is because we don't want to make any more assertions on the error returned if there isn't one around. Without this the test would carry on to the next step and panic because of a nil pointer.

Try to run the test

```
wallet_test.go:61: got err 'oh no' want 'cannot withdraw, insufficient funds'
```

Write enough code to make it pass

```
func (w *Wallet) Withdraw(amount Bitcoin) error {
    if amount > w.balance {
        return errors.New("cannot withdraw, insufficient funds")
    }
    w.balance -= amount
    return nil
}
```

Refactor

We have duplication of the error message in both the test code and the Withdraw code.

It would be really annoying for the test to fail if someone wanted to reword the error and it's just too much detail for our test. We don't really care what the exact wording is, just that some kind of meaningful error around withdrawing is returned given a certain condition.

In Go, errors are values, so we can refactor it out into a variable and have a single source of truth for it.

```
var ErrInsufficientFunds = errors.New("cannot withdraw, insufficient funds")

func (w *Wallet) Withdraw(amount Bitcoin) error {
    if amount > w.balance {
        return ErrInsufficientFunds
    }
    w.balance -= amount
    return nil
}
```

The var keyword allows us to define values global to the package.

This is a positive change in itself because now our Withdraw function looks very clear.

Next we can refactor our test code to use this value instead of specific strings.

```
func TestWallet(t *testing.T) {
    t.Run("deposit", func(t *testing.T) {
        wallet := Wallet{}
        wallet.Deposit(Bitcoin(10))
        assertBalance(t, wallet, Bitcoin(10))
    })

    t.Run("withdraw with funds", func(t *testing.T) {
        wallet := Wallet{Bitcoin(20)}
        wallet.Withdraw(Bitcoin(10))
        assertBalance(t, wallet, Bitcoin(10))
    })

    t.Run("withdraw insufficient funds", func(t *testing.T) {
        wallet := Wallet{Bitcoin(20)}
        err := wallet.Withdraw(Bitcoin(100))

        assertError(t, err, ErrInsufficientFunds)
        assertBalance(t, wallet, Bitcoin(20))
    })
}

func assertBalance(t testing.TB, wallet Wallet, want Bitcoin) {
    t.Helper()
    got := wallet.Balance()

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

func assertError(t testing.TB, got, want error) {
    t.Helper()
    if got == nil {
        t.Fatal("didn't get an error but wanted one")
    }

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

And now the test is easier to follow too.

I have moved the helpers out of the main test function just so when someone opens up a file they can start reading our assertions first, rather than some helpers.

Another useful property of tests is that they help us understand the real usage of our code so we can make sympathetic code. We can see here that a developer can simply call our code and do an equals check to ErrInsufficientFunds and act accordingly.

Unchecked errors

Whilst the Go compiler helps you a lot, sometimes there are things you can still miss and error handling can sometimes be tricky.

There is one scenario we have not tested. To find it, run the following in a terminal to install errcheck, one of many linters available for Go.

```
go install github.com/kisielk/errcheck@latest
```

Then, inside the directory with your code run errcheck .

You should get something like

```
wallet_test.go:17:18: wallet.Withdraw(Bitcoin(10))
```

What this is telling us is that we have not checked the error being returned on that line of code. That line of code on my computer corresponds to our normal withdraw scenario because we have not checked that if the Withdraw is successful that an error is not returned.

Here is the final test code that accounts for this.

```
func TestWallet(t *testing.T) {  
  
    t.Run("deposit", func(t *testing.T) {  
        wallet := Wallet{}  
        wallet.Deposit(Bitcoin(10))  
  
        assertBalance(t, wallet, Bitcoin(10))  
    })  
  
    t.Run("withdraw with funds", func(t *testing.T) {  
        wallet := Wallet{Bitcoin(20)}  
        err := wallet.Withdraw(Bitcoin(10))  
  
        assertNoError(t, err)  
        assertBalance(t, wallet, Bitcoin(10))  
    })  
  
    t.Run("withdraw insufficient funds", func(t *testing.T) {
```

```
wallet := Wallet{Bitcoin(20)}
err := wallet.Withdraw(Bitcoin(100))

assertError(t, err, ErrInsufficientFunds)
assertBalance(t, wallet, Bitcoin(20))
}
}

func assertBalance(t testing.TB, wallet Wallet, want Bitcoin) {
t.Helper()
got := wallet.Balance()

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

func assertNoError(t testing.TB, got error) {
t.Helper()
if got != nil {
    t.Fatal("got an error but didn't want one")
}
}

func assertError(t testing.TB, got error, want error) {
t.Helper()
if got == nil {
    t.Fatal("didn't get an error but wanted one")
}

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

Wrapping up

Pointers

- Go copies values when you pass them to functions/methods, so if you're writing a function that needs to mutate state you'll need it to take a pointer to the thing you want to change.
- The fact that Go takes a copy of values is useful a lot of the time but sometimes you won't want your system to make a copy of something, in which case you need to pass a reference. Examples include referencing very large data structures or things

where only one instance is necessary (like database connection pools).

nil

- Pointers can be nil
- When a function returns a pointer to something, you need to make sure you check if it's nil or you might raise a runtime exception - the compiler won't help you here.
- Useful for when you want to describe a value that could be missing

Errors

- Errors are the way to signify failure when calling a function/method.
- By listening to our tests we concluded that checking for a string in an error would result in a flaky test. So we refactored our implementation to use a meaningful value instead and this resulted in easier to test code and concluded this would be easier for users of our API too.
- This is not the end of the story with error handling, you can do more sophisticated things but this is just an intro. Later sections will cover more strategies.
- **Don't just check errors, handle them gracefully**

Create new types from existing ones

- Useful for adding more domain specific meaning to values
- Can let you implement interfaces

Pointers and errors are a big part of writing Go that you need to get comfortable with. Thankfully the compiler will usually help you out if you do something wrong, just take your time and read the error.

Maps

You can find all the code for this chapter [here](#)

In [arrays & slices](#), you saw how to store values in order. Now, we will look at a way to store items by a key and look them up quickly.

Maps allow you to store items in a manner similar to a dictionary. You can think of the key as the word and the value as the definition. And what better way is there to learn about Maps than to build our own dictionary?

First, assuming we already have some words with their definitions in the dictionary, if we search for a word, it should return the definition of it.

Write the test first

In dictionary_test.go

```
package main

import "testing"

func TestSearch(t *testing.T) {
    dictionary := map[string]string{"test": "this is just a test"}

    got := Search(dictionary, "test")
    want := "this is just a test"

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

Declaring a Map is somewhat similar to an array. Except, it starts with the map keyword and requires two types. The first is the key type, which is written inside the []. The second is the value type, which goes right after the [].

The key type is special. It can only be a comparable type because without the ability to tell if 2 keys are equal, we have no way to ensure that we are getting the correct value. Comparable types are explained in depth in the [language spec](#).

The value type, on the other hand, can be any type you want. It can even be another map.

Everything else in this test should be familiar.

Try to run the test

By running go test the compiler will fail with ./dictionary_test.go:8:9: undefined: Search.

Write the minimal amount of code for the test to run and check the output

In dictionary.go

```
package main

func Search(dictionary map[string]string, word string) string {
    return ""
}
```

Your test should now fail with a clear error message
dictionary_test.go:12: got " want 'this is just a test' given, 'test'.

Write enough code to make it pass

```
func Search(dictionary map[string]string, word string) string {
    return dictionary[word]
}
```

Getting a value out of a Map is the same as getting a value out of Array map[key].

Refactor

```
func TestSearch(t *testing.T) {
    dictionary := map[string]string{"test": "this is just a test"}

    got := Search(dictionary, "test")
    want := "this is just a test"

    assertStrings(t, got, want)
}

func assertStrings(t testing.TB, got, want string) {
    t.Helper()

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

I decided to create an assertStrings helper to make the implementation more general.

Using a custom type

We can improve our dictionary's usage by creating a new type around map and making Search a method.

In dictionary_test.go:

```
func TestSearch(t *testing.T) {
    dictionary := Dictionary{"test": "this is just a test"}

    got := dictionary.Search("test")
    want := "this is just a test"

    assertStrings(t, got, want)
}
```

We started using the Dictionary type, which we have not defined yet. Then called Search on the Dictionary instance.

We did not need to change assertStrings.

In dictionary.go:

```
type Dictionary map[string]string

func (d Dictionary) Search(word string) string {
    return d[word]
}
```

Here we created a Dictionary type which acts as a thin wrapper around map. With the custom type defined, we can create the Search method.

Write the test first

The basic search was very easy to implement, but what will happen if we supply a word that's not in our dictionary?

We actually get nothing back. This is good because the program can continue to run, but there is a better approach. The function can report that the word is not in the dictionary. This way, the user isn't left wondering if the word doesn't exist or if there is just no definition (this might not seem very useful for a dictionary). However, it's a scenario that could be key in other usecases).

```
func TestSearch(t *testing.T) {
    dictionary := Dictionary{"test": "this is just a test"}

    t.Run("known word", func(t *testing.T) {
        got, _ := dictionary.Search("test")
        want := "this is just a test"

        assertStrings(t, got, want)
    })

    t.Run("unknown word", func(t *testing.T) {
```

```
_ , err := dictionary.Search("unknown")
want := "could not find the word you were looking for"

if err == nil {
    t.Fatal("expected to get an error.")
}

assertStrings(t, err.Error(), want)
}
}
```

The way to handle this scenario in Go is to return a second argument which is an Error type.

Errors can be converted to a string with the .Error() method, which we do when passing it to the assertion. We are also protecting assertStrings with if to ensure we don't call .Error() on nil.

Try and run the test

This does not compile

```
./dictionary_test.go:18:10: assignment mismatch: 2 variables but 1 values
```

Write the minimal amount of code for the test to run and check the output

```
func (d Dictionary) Search(word string) (string, error) {
    return d[word], nil
}
```

Your test should now fail with a much clearer error message.

```
dictionary_test.go:22: expected to get an error.
```

Write enough code to make it pass

```
func (d Dictionary) Search(word string) (string, error) {
    definition, ok := d[word]
    if !ok {
        return "", errors.New("could not find the word you were looking for")
    }

    return definition, nil
}
```

In order to make this pass, we are using an interesting property of the map lookup. It can return 2 values. The second value is a boolean which indicates if the key was found successfully.

This property allows us to differentiate between a word that doesn't exist and a word that just doesn't have a definition.

Refactor

```
var ErrNotFound = errors.New("could not find the word you were looking for")

func (d Dictionary) Search(word string) (string, error) {
    definition, ok := d[word]
    if !ok {
        return "", ErrNotFound
    }

    return definition, nil
}
```

We can get rid of the magic error in our Search function by extracting it into a variable. This will also allow us to have a better test.

```
t.Run("unknown word", func(t *testing.T) {
    _, got := dictionary.Search("unknown")

    assertError(t, got, ErrNotFound)
})

func assertError(t testing.TB, got, want error) {
    t.Helper()

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

By creating a new helper we were able to simplify our test, and start using our ErrNotFound variable so our test doesn't fail if we change the error text in the future.

Write the test first

We have a great way to search the dictionary. However, we have no way to add new words to our dictionary.

```
func TestAdd(t *testing.T) {
    dictionary := Dictionary{}
```

```
dictionary.Add("test", "this is just a test")

want := "this is just a test"
got, err := dictionary.Search("test")
if err != nil {
    t.Fatal("should find added word:", err)
}

assertStrings(t, got, want)
}
```

In this test, we are utilizing our Search function to make the validation of the dictionary a little easier.

Write the minimal amount of code for the test to run and check output

In dictionary.go

```
func (d Dictionary) Add(word, definition string) {  
}
```

Your test should now fail

```
dictionary_test.go:31: should find added word: could not find the word you were looking for
```

Write enough code to make it pass

```
func (d Dictionary) Add(word, definition string) {  
    d[word] = definition  
}
```

Adding to a map is also similar to an array. You just need to specify a key and set it equal to a value.

Pointers, copies, et al

An interesting property of maps is that you can modify them without passing as an address to it (e.g &myMap)

This may make them feel like a "reference type", [but as Dave Cheney describes](#) they are not.

A map value is a pointer to a runtime.hmap structure.

So when you pass a map to a function/method, you are indeed copying it, but just the pointer part, not the underlying data structure that contains the data.

A gotcha with maps is that they can be a nil value. A nil map behaves like an empty map when reading, but attempts to write to a nil map will cause a runtime panic. You can read more about maps [here](#).

Therefore, you should never initialize an empty map variable:

```
var m map[string]string
```

Instead, you can initialize an empty map like we were doing above, or use the make keyword to create a map for you:

```
var dictionary = map[string]string{}
```

// OR

```
var dictionary = make(map[string]string)
```

Both approaches create an empty hash map and point dictionary at it. Which ensures that you will never get a runtime panic.

Refactor

There isn't much to refactor in our implementation but the test could use a little simplification.

```
func TestAdd(t *testing.T) {
    dictionary := Dictionary{}
    word := "test"
    definition := "this is just a test"

    dictionary.Add(word, definition)

    assertDefinition(t, dictionary, word, definition)
}

func assertDefinition(t testing.TB, dictionary Dictionary, word, definition string) {
    t.Helper()

    got, err := dictionary.Search(word)
    if err != nil {
        t.Fatal("should find added word:", err)
    }
    assertStrings(t, got, definition)
}
```

We made variables for word and definition, and moved the definition assertion into its own helper function.

Our Add is looking good. Except, we didn't consider what happens when the value we are trying to add already exists!

Map will not throw an error if the value already exists. Instead, they will go ahead and overwrite the value with the newly provided value. This can be convenient in practice, but makes our function name less than accurate. Add should not modify existing values. It should only add new words to our dictionary.

Write the test first

```
func TestAdd(t *testing.T) {
    t.Run("new word", func(t *testing.T) {
        dictionary := Dictionary{}
        word := "test"
        definition := "this is just a test"

        err := dictionary.Add(word, definition)

        assertError(t, err, nil)
        assertDefinition(t, dictionary, word, definition)
    })

    t.Run("existing word", func(t *testing.T) {
        word := "test"
        definition := "this is just a test"
        dictionary := Dictionary{word: definition}
        err := dictionary.Add(word, "new test")

        assertError(t, err, ErrWordExists)
        assertDefinition(t, dictionary, word, definition)
    })
}

func assertError(t testing.TB, got, want error) {
    t.Helper()
    if got != want {
        t.Errorf("got %q want %q", got, want)
    }
}
```

For this test, we modified Add to return an error, which we are validating against a new error variable, ErrWordExists. We also modified the previous test to check for a nil error, as well as the assertError function.

Try to run test

The compiler will fail because we are not returning a value for Add.

```
./dictionary_test.go:30:13: dictionary.Add(word, definition) used as value  
./dictionary_test.go:41:13: dictionary.Add(word, "new test") used as value
```

Write the minimal amount of code for the test to run and check the output

In dictionary.go

```
var (  
    ErrNotFound = errors.New("could not find the word you were looking for")  
    ErrWordExists = errors.New("cannot add word because it already exists")  
)  
  
func (d Dictionary) Add(word, definition string) error {  
    d[word] = definition  
    return nil  
}
```

Now we get two more errors. We are still modifying the value, and returning a nil error.

```
dictionary_test.go:43: got error '%!q(<nil>)' want 'cannot add word because it already exists'  
dictionary_test.go:44: got 'new test' want 'this is just a test'
```

Write enough code to make it pass

```
func (d Dictionary) Add(word, definition string) error {  
    _, err := d.Search(word)  
  
    switch err {  
        case ErrNotFound:  
            d[word] = definition  
        case nil:  
            return ErrWordExists  
        default:  
            return err  
    }  
  
    return nil  
}
```

Here we are using a switch statement to match on the error. Having a switch like this provides an extra safety net, in case Search returns an error other than ErrNotFound.

Refactor

We don't have too much to refactor, but as our error usage grows we can make a few modifications.

```
const (
    ErrNotFound = DictionaryErr("could not find the word you were looking for")
    ErrWordExists = DictionaryErr("cannot add word because it already exists")
)

type DictionaryErr string

func (e DictionaryErr) Error() string {
    return string(e)
}
```

We made the errors constant; this required us to create our own `DictionaryErr` type which implements the error interface. You can read more about the details in [this excellent article by Dave Cheney](#). Simply put, it makes the errors more reusable and immutable.

Next, let's create a function to Update the definition of a word.

Write the test first

```
func TestUpdate(t *testing.T) {
    word := "test"
    definition := "this is just a test"
    dictionary := Dictionary{word: definition}
    newDefinition := "new definition"

    dictionary.Update(word, newDefinition)

    assertDefinition(t, dictionary, word, newDefinition)
}
```

Update is very closely related to Add and will be our next implementation.

Try and run the test

```
./dictionary_test.go:53:2: dictionary.Update undefined (type Dictionary has no field or method U
```

Write minimal amount of code for the test to run and check the failing test output

We already know how to deal with an error like this. We need to define our function.

```
func (d Dictionary) Update(word, definition string) {}
```

With that in place, we are able to see that we need to change the definition of the word.

```
dictionary_test.go:55: got 'this is just a test' want 'new definition'
```

Write enough code to make it pass

We already saw how to do this when we fixed the issue with Add. So let's implement something really similar to Add.

```
func (d Dictionary) Update(word, definition string) {  
    d[word] = definition  
}
```

There is no refactoring we need to do on this since it was a simple change. However, we now have the same issue as with Add. If we pass in a new word, Update will add it to the dictionary.

Write the test first

```
t.Run("existing word", func(t *testing.T) {  
    word := "test"  
    definition := "this is just a test"  
    dictionary := Dictionary{word: definition}  
    newDefinition := "new definition"  
  
    err := dictionary.Update(word, newDefinition)  
  
    assertError(t, err, nil)  
    assertDefinition(t, dictionary, word, newDefinition)  
})  
  
t.Run("new word", func(t *testing.T) {  
    word := "test"  
    definition := "this is just a test"  
    dictionary := Dictionary{}  
  
    err := dictionary.Update(word, definition)
```

```
    assertError(t, err, ErrWordDoesNotExist)
})
```

We added yet another error type for when the word does not exist. We also modified Update to return an error value.

Try and run the test

```
./dictionary_test.go:53:16: dictionary.Update(word, newDefinition) used as value
./dictionary_test.go:64:16: dictionary.Update(word, definition) used as value
./dictionary_test.go:66:23: undefined: ErrWordDoesNotExist
```

We get 3 errors this time, but we know how to deal with these.

Write the minimal amount of code for the test to run and check the failing test output

```
const (
    ErrNotFound     = DictionaryErr("could not find the word you were looking for")
    ErrWordExists   = DictionaryErr("cannot add word because it already exists")
    ErrWordDoesNotExist = DictionaryErr("cannot update word because it does not exist")
)

func (d Dictionary) Update(word, definition string) error {
    d[word] = definition
    return nil
}
```

We added our own error type and are returning a nil error.

With these changes, we now get a very clear error:

```
dictionary_test.go:66: got error '%!q(<nil>) want 'cannot update word because it does not exist'
```

Write enough code to make it pass

```
func (d Dictionary) Update(word, definition string) error {
    _, err := d.Search(word)

    switch err {
    case ErrNotFound:
        return ErrWordDoesNotExist
    case nil:
        d[word] = definition
    default:
        return err
    }
```

```
    return nil
}
```

This function looks almost identical to Add except we switched when we update the dictionary and when we return an error.

Note on declaring a new error for Update

We could reuse ErrNotFound and not add a new error. However, it is often better to have a precise error for when an update fails.

Having specific errors gives you more information about what went wrong. Here is an example in a web app:

You can redirect the user when ErrNotFound is encountered, but display an error message when ErrWordDoesNotExist is encountered.

Next, let's create a function to Delete a word in the dictionary.

Write the test first

```
func TestDelete(t *testing.T) {
    word := "test"
    dictionary := Dictionary{word: "test definition"}

    dictionary.Delete(word)

    _, err := dictionary.Search(word)
    if err != ErrNotFound {
        t.Errorf("Expected %q to be deleted", word)
    }
}
```

Our test creates a Dictionary with a word and then checks if the word has been removed.

Try to run the test

By running go test we get:

```
./dictionary_test.go:74:6: dictionary.Delete undefined (type Dictionary has no field or method D
```

Write the minimal amount of code for the test to run and check the failing test output

```
func (d Dictionary) Delete(word string) {  
}
```

After we add this, the test tells us we are not deleting the word.

```
dictionary_test.go:78: Expected 'test' to be deleted
```

Write enough code to make it pass

```
func (d Dictionary) Delete(word string) {  
    delete(d, word)  
}
```

Go has a built-in function `delete` that works on maps. It takes two arguments. The first is the map and the second is the key to be removed.

The `delete` function returns nothing, and we based our `Delete` method on the same notion. Since deleting a value that's not there has no effect, unlike our `Update` and `Add` methods, we don't need to complicate the API with errors.

Wrapping up

In this section, we covered a lot. We made a full CRUD (Create, Read, Update and Delete) API for our dictionary. Throughout the process we learned how to:

- Create maps
- Search for items in maps
- Add new items to maps
- Update items in maps
- Delete items from a map
- Learned more about errors
 - How to create errors that are constants
 - Writing error wrappers

Dependency Injection

You can find all the code for this chapter here

It is assumed that you have read the [structs section](#) before as some understanding of interfaces will be needed for this.

There are a lot of misunderstandings around dependency injection around the programming community. Hopefully, this guide will show you how

- You don't need a framework
- It does not overcomplicate your design
- It facilitates testing
- It allows you to write great, general-purpose functions.

We want to write a function that greets someone, just like we did in the hello-world chapter but this time we are going to be testing the actual printing.

Just to recap, here is what that function could look like

```
func Greet(name string) {
    fmt.Printf("Hello, %s", name)
}
```

But how can we test this? Calling `fmt.Printf` prints to `stdout`, which is pretty hard for us to capture using the testing framework.

What we need to do is to be able to **inject** (which is just a fancy word for pass in) the dependency of printing.

Our function doesn't need to care where or how the printing happens, so we should accept an interface rather than a concrete type.

If we do that, we can then change the implementation to print to something we control so that we can test it. In "real life" you would inject in something that writes to `stdout`.

If you look at the source code of `fmt.Printf` you can see a way for us to hook in

```
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

Interesting! Under the hood `Printf` just calls `Fprintf` passing in `os.Stdout`.

What exactly is an `os.Stdout`? What does `Fprintf` expect to get passed to it for the 1st argument?

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
```

```
    return
```

```
}
```

An io.Writer

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

From this we can infer that os.Stdout implements io.Writer; Printf passes os.Stdout to Fprintf which expects an io.Writer.

As you write more Go code you will find this interface popping up a lot because it's a great general purpose interface for "put this data somewhere".

So we know under the covers we're ultimately using Writer to send our greeting somewhere. Let's use this existing abstraction to make our code testable and more reusable.

Write the test first

```
func TestGreet(t *testing.T) {
    buffer := bytes.Buffer{}
    Greet(&buffer, "Chris")

    got := buffer.String()
    want := "Hello, Chris"

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

The Buffer type from the bytes package implements the Writer interface, because it has the method Write(p []byte) (n int, err error).

So we'll use it in our test to send in as our Writer and then we can check what was written to it after we invoke Greet

Try and run the test

The test will not compile

```
./di_test.go:10:7: too many arguments in call to Greet
    have (*bytes.Buffer, string)
    want (string)
```

Write the minimal amount of code for the test to run and check the failing test output

Listen to the compiler and fix the problem.

```
func Greet(writer *bytes.Buffer, name string) {
    fmt.Printf("Hello, %s", name)
}
```

Hello, Chris di_test.go:16: got " want 'Hello, Chris'

The test fails. Notice that the name is getting printed out, but it's going to stdout.

Write enough code to make it pass

Use the writer to send the greeting to the buffer in our test. Remember fmt.Fprintf is like fmt.Printf but instead takes a Writer to send the string to, whereas fmt.Printf defaults to stdout.

```
func Greet(writer *bytes.Buffer, name string) {
    fmt.Fprintf(writer, "Hello, %s", name)
}
```

The test now passes.

Refactor

Earlier the compiler told us to pass in a pointer to a bytes.Buffer. This is technically correct but not very useful.

To demonstrate this, try wiring up the Greet function into a Go application where we want it to print to stdout.

```
func main() {
    Greet(os.Stdout, "Elodie")
}
```

./di.go:14:7: cannot use os.Stdout (type *os.File) as type *bytes.Buffer in argument to Greet

As discussed earlier fmt.Fprintf allows you to pass in an io.Writer which we know both os.Stdout and bytes.Buffer implement.

If we change our code to use the more general purpose interface we can now use it in both tests and in our application.

```
package main
```

```
import (
    "fmt"
```

```
    "io"
    "os"
)

func Greet(writer io.Writer, name string) {
    fmt.Fprintf(writer, "Hello, %s", name)
}

func main() {
    Greet(os.Stdout, "Elodie")
}
```

More on io.Writer

What other places can we write data to using io.Writer? Just how general purpose is our Greet function?

The Internet

Run the following

```
package main

import (
    "fmt"
    "io"
    "log"
    "net/http"
)

func Greet(writer io.Writer, name string) {
    fmt.Fprintf(writer, "Hello, %s", name)
}

func MyGreeterHandler(w http.ResponseWriter, r *http.Request) {
    Greet(w, "world")
}

func main() {
    log.Fatal(http.ListenAndServe(":5001", http.HandlerFunc(MyGreeterHandler)))
}
```

Run the program and go to <http://localhost:5001>. You'll see your greeting function being used.

HTTP servers will be covered in a later chapter so don't worry too much about the details.

When you write an HTTP handler, you are given an `http.ResponseWriter` and the `http.Request` that was used to make the request. When you implement your server you write your response using the writer.

You can probably guess that `http.ResponseWriter` also implements `io.Writer` so this is why we could re-use our `Greet` function inside our handler.

Wrapping up

Our first round of code was not easy to test because it wrote data to somewhere we couldn't control.

Motivated by our tests we refactored the code so we could control where the data was written by **injecting a dependency** which allowed us to:

- **Test our code** If you can't test a function easily, it's usually because of dependencies hard-wired into a function or global state. If you have a global database connection pool for instance that is used by some kind of service layer, it is likely going to be difficult to test and they will be slow to run. DI will motivate you to inject in a database dependency (via an interface) which you can then mock out with something you can control in your tests.
- **Separate our concerns**, decoupling where the data goes from how to generate it. If you ever feel like a method/function has too many responsibilities (generating data and writing to a db? handling HTTP requests and doing domain level logic?) DI is probably going to be the tool you need.
- **Allow our code to be re-used in different contexts** The first "new" context our code can be used in is inside tests. But further on if someone wants to try something new with your function they can inject their own dependencies.

What about mocking? I hear you need that for DI and also it's evil

Mocking will be covered in detail later (and it's not evil). You use mocking to replace real things you inject with a pretend version that you can control and inspect in your tests. In our case though, the standard library had something ready for us to use.

The Go standard library is really good, take time to study it

By having some familiarity with the `io.Writer` interface we are able to use `bytes.Buffer` in our test as our Writer and then we can use other

Writers from the standard library to use our function in a command line app or in web server.

The more familiar you are with the standard library the more you'll see these general purpose interfaces which you can then re-use in your own code to make your software reusable in a number of contexts.

This example is heavily influenced by a chapter in [The Go Programming language](#), so if you enjoyed this, go buy it!

Mocking

[You can find all the code for this chapter here](#)

You have been asked to write a program which counts down from 3, printing each number on a new line (with a 1-second pause) and when it reaches zero it will print "Go!" and exit.

```
3  
2  
1  
Go!
```

We'll tackle this by writing a function called Countdown which we will then put inside a main program so it looks something like this:

```
package main  
  
func main() {  
    Countdown()  
}
```

While this is a pretty trivial program, to test it fully we will need as always to take an iterative, test-driven approach.

What do I mean by iterative? We make sure we take the smallest steps we can to have useful software.

We don't want to spend a long time with code that will theoretically work after some hacking because that's often how developers fall down rabbit holes. **It's an important skill to be able to slice up requirements as small as you can so you can have working software.**

Here's how we can divide our work up and iterate on it:

- Print 3
- Print 3, 2, 1 and Go!
- Wait a second between each line

Write the test first

Our software needs to print to stdout and we saw how we could use Dependency Injection (DI) to facilitate testing this in the DI section.

```
func TestCountdown(t *testing.T) {
    buffer := &bytes.Buffer{}

    Countdown(buffer)

    got := buffer.String()
    want := "3"

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

If anything like buffer is unfamiliar to you, re-read [the previous section](#).

We know we want our Countdown function to write data somewhere and io.Writer is the de-facto way of capturing that as an interface in Go.

- In main we will send to os.Stdout so our users see the countdown printed to the terminal.
- In test we will send to bytes.Buffer so our tests can capture what data is being generated.

Try and run the test

```
./countdown_test.go:11:2: undefined: Countdown
```

Write the minimal amount of code for the test to run and check the failing test output

Define Countdown

```
func Countdown() {}
```

Try again

```
./countdown_test.go:11:11: too many arguments in call to Countdown
    have (*bytes.Buffer)
    want ()
```

The compiler is telling you what your function signature could be, so update it.

```
func Countdown(out *bytes.Buffer) {}  
countdown_test.go:17: got " want '3'  
Perfect!
```

Write enough code to make it pass

```
func Countdown(out *bytes.Buffer) {  
    fmt.Fprint(out, "3")  
}
```

We're using `fmt.Fprint` which takes an `io.Writer` (like `*bytes.Buffer`) and sends a string to it. The test should pass.

Refactor

We know that while `*bytes.Buffer` works, it would be better to use a general purpose interface instead.

```
func Countdown(out io.Writer) {  
    fmt.Fprint(out, "3")  
}
```

Re-run the tests and they should be passing.

To complete matters, let's now wire up our function into a main so we have some working software to reassure ourselves we're making progress.

```
package main
```

```
import (  
    "fmt"  
    "io"  
    "os"  
)  
  
func Countdown(out io.Writer) {  
    fmt.Fprint(out, "3")  
}  
  
func main() {  
    Countdown(os.Stdout)  
}
```

Try and run the program and be amazed at your handywork.

Yes this seems trivial but this approach is what I would recommend for any project. **Take a thin slice of functionality and make it work end-to-end, backed by tests.**

Next we can make it print 2,1 and then "Go!".

Write the test first

By investing in getting the overall plumbing working right, we can iterate on our solution safely and easily. We will no longer need to stop and re-run the program to be confident of it working as all the logic is tested.

```
func TestCountdown(t *testing.T) {
    buffer := &bytes.Buffer{}

    Countdown(buffer)

    got := buffer.String()
    want := `3
2
1
Go!`  

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

The backtick syntax is another way of creating a string but lets you include things like newlines, which is perfect for our test.

Try and run the test

```
countdown_test.go:21: got '3' want '3
2
1
Go!'
```

Write enough code to make it pass

```
func Countdown(out io.Writer) {
    for i := 3; i > 0; i-- {
        fmt.Fprintln(out, i)
    }
}
```

```
    fmt.Fprint(out, "Go!")
}
```

Use a for loop counting backwards with i-- and use fmt.Fprintln to print to out with our number followed by a newline character. Finally use fmt.Fprint to send "Go!" afterward.

Refactor

There's not much to refactor other than refactoring some magic values into named constants.

```
const finalWord = "Go!"
const countdownStart = 3
```

```
func Countdown(out io.Writer) {
    for i := countdownStart; i > 0; i-- {
        fmt.Fprintln(out, i)
    }
    fmt.Fprint(out, finalWord)
}
```

If you run the program now, you should get the desired output but we don't have it as a dramatic countdown with the 1-second pauses.

Go lets you achieve this with time.Sleep. Try adding it in to our code.

```
func Countdown(out io.Writer) {
    for i := countdownStart; i > 0; i-- {
        fmt.Fprintln(out, i)
        time.Sleep(1 * time.Second)
    }
    fmt.Fprint(out, finalWord)
}
```

If you run the program it works as we want it to.

Mocking

The tests still pass and the software works as intended but we have some problems:

- Our tests take 3 seconds to run.
 - Every forward-thinking post about software development emphasises the importance of quick feedback loops.
 - **Slow tests ruin developer productivity.**

-
- Imagine if the requirements get more sophisticated warranting more tests. Are we happy with 3s added to the test run for every new test of Countdown?
 - We have not tested an important property of our function.

We have a dependency on Sleeping which we need to extract so we can then control it in our tests.

If we can mock time.Sleep we can use dependency injection to use it instead of a "real" time.Sleep and then we can **spy on the calls** to make assertions on them.

Write the test first

Let's define our dependency as an interface. This lets us then use a real Sleeper in main and a spy sleeper in our tests. By using an interface our Countdown function is oblivious to this and adds some flexibility for the caller.

```
type Sleeper interface {
    Sleep()
}
```

I made a design decision that our Countdown function would not be responsible for how long the sleep is. This simplifies our code a little for now at least and means a user of our function can configure that sleepiness however they like.

Now we need to make a mock of it for our tests to use.

```
type SpySleeper struct {
    Calls int
}

func (s *SpySleeper) Sleep() {
    s.Calls++
}
```

Spies are a kind of mock which can record how a dependency is used. They can record the arguments sent in, how many times it has been called, etc. In our case, we're keeping track of how many times Sleep() is called so we can check it in our test.

Update the tests to inject a dependency on our Spy and assert that the sleep has been called 3 times.

```
func TestCountdown(t *testing.T) {
    buffer := &bytes.Buffer{}
    spySleeper := &SpySleeper{}
```

```
Countdown(buffer, spySleeper)

got := buffer.String()
want := `3
2
1
Go!` 

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

if spySleeper.Calls != 3 {
    t.Errorf("not enough calls to sleeper, want 3 got %d", spySleeper.Calls)
}
}
```

Try and run the test

```
too many arguments in call to Countdown
have (*bytes.Buffer, *SpySleeper)
want (io.Writer)
```

Write the minimal amount of code for the test to run and check the failing test output

We need to update Countdown to accept our Sleeper

```
func Countdown(out io.Writer, sleeper Sleeper) {
    for i := countdownStart; i > 0; i-- {
        fmt.Fprintln(out, i)
        time.Sleep(1 * time.Second)
    }

    fmt.Fprint(out, finalWord)
}
```

If you try again, your main will no longer compile for the same reason

```
./main.go:26:11: not enough arguments in call to Countdown
have (*os.File)
want (io.Writer, Sleeper)
```

Let's create a real sleeper which implements the interface we need

```
type DefaultSleeper struct{}
```

```
func (d *DefaultSleeper) Sleep() {
    time.Sleep(1 * time.Second)
}
```

We can then use it in our real application like so

```
func main() {
    sleeper := &DefaultSleeper{}
    Countdown(os.Stdout, sleeper)
}
```

Write enough code to make it pass

The test is now compiling but not passing because we're still calling the `time.Sleep` rather than the injected dependency. Let's fix that.

```
func Countdown(out io.Writer, sleeper Sleeper) {
    for i := countdownStart; i > 0; i-- {
        fmt.Fprintln(out, i)
        sleeper.Sleep()
    }

    fmt.Fprint(out, finalWord)
}
```

The test should pass and no longer take 3 seconds.

Still some problems

There's still another important property we haven't tested.

`Countdown` should sleep before each next print, e.g:

- Print N
- Sleep
- Print N-1
- Sleep
- Print Go!
- etc

Our latest change only asserts that it has slept 3 times, but those sleeps could occur out of sequence.

When writing tests if you're not confident that your tests are giving you sufficient confidence, just break it! (make sure you have committed your changes to source control first though). Change the code to the following

```
func Countdown(out io.Writer, sleeper Sleeper) {
    for i := countdownStart; i > 0; i-- {
```

```
    sleeper.Sleep()
}

for i := countdownStart; i > 0; i-- {
    fmt.Fprintln(out, i)
}

fmt.Fprint(out, finalWord)
}
```

If you run your tests they should still be passing even though the implementation is wrong.

Let's use spying again with a new test to check the order of operations is correct.

We have two different dependencies and we want to record all of their operations into one list. So we'll create one spy for them both.

```
type SpyCountdownOperations struct {
    Calls []string
}

func (s *SpyCountdownOperations) Sleep() {
    s.Calls = append(s.Calls, sleep)
}

func (s *SpyCountdownOperations) Write(p []byte) (n int, err error) {
    s.Calls = append(s.Calls, write)
    return
}

const write = "write"
const sleep = "sleep"
```

Our `SpyCountdownOperations` implements both `io.Writer` and `Sleeper`, recording every call into one slice. In this test we're only concerned about the order of operations, so just recording them as list of named operations is sufficient.

We can now add a sub-test into our test suite which verifies our sleeps and prints operate in the order we hope

```
t.Run("sleep before every print", func(t *testing.T) {
    spySleepPrinter := &SpyCountdownOperations{}
    Countdown(spySleepPrinter, spySleepPrinter)

    want := []string{
        write,
```

```
    sleep,
    write,
    sleep,
    write,
    sleep,
    write,
}
}

if !reflect.DeepEqual(want, spySleepPrinter.Calls) {
    t.Errorf("wanted calls %v got %v", want, spySleepPrinter.Calls)
}
})
```

This test should now fail. Revert Countdown back to how it was to fix the test.

We now have two tests spying on the Sleeper so we can now refactor our test so one is testing what is being printed and the other one is ensuring we're sleeping between the prints. Finally, we can delete our first spy as it's not used anymore.

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

    t.Run("prints 3 to Go!", func(t *testing.T) {
        buffer := &bytes.Buffer{}
        Countdown(buffer, &SpyCountdownOperations{})

        got := buffer.String()
        want := `3
2
1
Go!`


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

    t.Run("sleep before every print", func(t *testing.T) {
        spySleepPrinter := &SpyCountdownOperations{}
        Countdown(spySleepPrinter, spySleepPrinter)

        want := []string{
            "write",
            "sleep",
            "write",
            "sleep",
        }
```

```
        write,
        sleep,
        write,
    }

    if !reflect.DeepEqual(want, spySleepPrinter.Calls) {
        t.Errorf("wanted calls %v got %v", want, spySleepPrinter.Calls)
    }
}
}
```

We now have our function and its 2 important properties properly tested.

Extending Sleeper to be configurable

A nice feature would be for the Sleeper to be configurable. This means that we can adjust the sleep time in our main program.

Write the test first

Let's first create a new type for ConfigurableSleeper that accepts what we need for configuration and testing.

```
type ConfigurableSleeper struct {
    duration time.Duration
    sleep    func(time.Duration)
}
```

We are using duration to configure the time slept and sleep as a way to pass in a sleep function. The signature of sleep is the same as for time.Sleep allowing us to use time.Sleep in our real implementation and the following spy in our tests:

```
type SpyTime struct {
    durationSlept time.Duration
}

func (s *SpyTime) Sleep(duration time.Duration) {
    s.durationSlept = duration
}
```

With our spy in place, we can create a new test for the configurable sleeper.

```
func TestConfigurableSleeper(t *testing.T) {
    sleepTime := 5 * time.Second
```

```
spyTime := &SpyTime{}
sleeper := ConfigurableSleeper{sleepTime, spyTime.Sleep}
sleeper.Sleep()

if spyTime.durationSlept != sleepTime {
    t.Errorf("should have slept for %v but slept for %v", sleepTime, spyTime.durationSlept)
}
```

There should be nothing new in this test and it is set up very similar to the previous mock tests.

Try and run the test

sleeper.Sleep undefined (type ConfigurableSleeper has no field or method Sleep, but does have)

You should see a very clear error message indicating that we do not have a Sleep method created on our ConfigurableSleeper.

Write the minimal amount of code for the test to run and check failing test output

```
func (c *ConfigurableSleeper) Sleep() {
```

With our new Sleep function implemented we have a failing test.

```
countdown_test.go:56: should have slept for 5s but slept for 0s
```

Write enough code to make it pass

All we need to do now is implement the Sleep function for ConfigurableSleeper.

```
func (c *ConfigurableSleeper) Sleep() {
    c.sleep(c.duration)
}
```

With this change all of the tests should be passing again and you might wonder why all the hassle as the main program didn't change at all. Hopefully it becomes clear after the following section.

Cleanup and refactor

The last thing we need to do is to actually use our ConfigurableSleeper in the main function.

```
func main() {
    sleeper := &ConfigurableSleeper{1 * time.Second, time.Sleep}
    Countdown(os.Stdout, sleeper)
}
```

If we run the tests and the program manually, we can see that all the behavior remains the same.

Since we are using the ConfigurableSleeper, it is now safe to delete the DefaultSleeper implementation. Wrapping up our program and having a more **generic** Sleeper with arbitrary long countdowns.

But isn't mocking evil?

You may have heard mocking is evil. Just like anything in software development it can be used for evil, just like **DRY**.

People normally get in to a bad state when they don't listen to their tests and are not respecting the refactoring stage.

If your mocking code is becoming complicated or you are having to mock out lots of things to test something, you should listen to that bad feeling and think about your code. Usually it is a sign of

- The thing you are testing is having to do too many things (because it has too many dependencies to mock)
 - Break the module apart so it does less
- Its dependencies are too fine-grained
 - Think about how you can consolidate some of these dependencies into one meaningful module
- Your test is too concerned with implementation details
 - Favour testing expected behaviour rather than the implementation

Normally a lot of mocking points to bad abstraction in your code.

What people see here is a weakness in TDD but it is actually a strength, more often than not poor test code is a result of bad design or put more nicely, well-designed code is easy to test.

But mocks and tests are still making my life hard!

Ever run into this situation?

- You want to do some refactoring
- To do this you end up changing lots of tests
- You question TDD and make a post on Medium titled "Mocking considered harmful"

This is usually a sign of you testing too much implementation detail. Try to make it so your tests are testing useful behaviour unless the implementation is really important to how the system runs.

It is sometimes hard to know what level to test exactly but here are some thought processes and rules I try to follow:

- **The definition of refactoring is that the code changes but the behaviour stays the same.** If you have decided to do some refactoring in theory you should be able to make the commit without any test changes. So when writing a test ask yourself
 - Am I testing the behaviour I want, or the implementation details?
 - If I were to refactor this code, would I have to make lots of changes to the tests?
- Although Go lets you test private functions, I would avoid it as private functions are implementation detail to support public behaviour. Test the public behaviour. Sandi Metz describes private functions as being "less stable" and you don't want to couple your tests to them.
- I feel like if a test is working with **more than 3 mocks then it is a red flag** - time for a rethink on the design
- Use spies with caution. Spies let you see the insides of the algorithm you are writing which can be very useful but that means a tighter coupling between your test code and the implementation. **Be sure you actually care about these details if you're going to spy on them**

Can't I just use a mocking framework? Mocking requires no magic and is relatively simple; using a framework can make mocking seem more complicated than it is. We don't use automocking in this chapter so that we get:

- a better understanding of how to mock
- practice implementing interfaces

In collaborative projects there is value in auto-generating mocks. In a team, a mock generation tool codifies consistency around the test doubles. This will avoid inconsistently written test doubles which can translate to inconsistently written tests.

You should only use a mock generator that generates test doubles against an interface. Any tool that overly dictates how tests are written, or that use lots of 'magic', can get in the sea.

Wrapping up

More on TDD approach

- When faced with less trivial examples, break the problem down into "thin vertical slices". Try to get to a point where you have working software backed by tests as soon as you can, to avoid getting in rabbit holes and taking a "big bang" approach.
- Once you have some working software it should be easier to iterate with small steps until you arrive at the software you need.

"When to use iterative development? You should use iterative development only on projects that you want to succeed."

Martin Fowler.

Mocking

- **Without mocking important areas of your code will be untested.** In our case we would not be able to test that our code paused between each print but there are countless other examples. Calling a service that can fail? Wanting to test your system in a particular state? It is very hard to test these scenarios without mocking.
- Without mocks you may have to set up databases and other third parties things just to test simple business rules. You're likely to have slow tests, resulting in **slow feedback loops**.
- By having to spin up a database or a webservice to test something you're likely to have **fragile tests** due to the unreliability of such services.

Once a developer learns about mocking it becomes very easy to over-test every single facet of a system in terms of the way it works rather than what it does. Always be mindful about **the value of your tests** and what impact they would have in future refactoring.

In this post about mocking we have only covered **Spies**, which are a kind of mock. Mocks are a type of "test double."

Test Double is a generic term for any case where you replace a production object for testing purposes.

Under test doubles, there are various types like stubs, spies and indeed mocks! Check out [Martin Fowler's post](#) for more detail.

Concurrency

You can find all the code for this chapter here

Here's the setup: a colleague has written a function, CheckWebsites, that checks the status of a list of URLs.

```
package concurrency

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)

    for _, url := range urls {
        results[url] = wc(url)
    }

    return results
}
```

It returns a map of each URL checked to a boolean value: true for a good response; false for a bad response.

You also have to pass in a WebsiteChecker which takes a single URL and returns a boolean. This is used by the function to check all the websites.

Using dependency injection has allowed them to test the function without making real HTTP calls, making it reliable and fast.

Here's the test they've written:

```
package concurrency

import (
    "reflect"
    "testing"
)

func mockWebsiteChecker(url string) bool {
    if url == "waat://furhurterwe.geds" {
        return false
    }
    return true
}

func TestCheckWebsites(t *testing.T) {
    websites := []string{
```

```
    "http://google.com",
    "http://blog.gypsydave5.com",
    "waat://furhruterwe.geds",
}

want := map[string]bool{
    "http://google.com":      true,
    "http://blog.gypsydave5.com": true,
    "waat://furhruterwe.geds": false,
}

got := CheckWebsites(mockWebsiteChecker, websites)

if !reflect.DeepEqual(want, got) {
    t.Fatalf("wanted %v, got %v", want, got)
}
}
```

The function is in production and being used to check hundreds of websites. But your colleague has started to get complaints that it's slow, so they've asked you to help speed it up.

Write a test

Let's use a benchmark to test the speed of `CheckWebsites` so that we can see the effect of our changes.

```
package concurrency

import (
    "testing"
    "time"
)

func slowStubWebsiteChecker(_ string) bool {
    time.Sleep(20 * time.Millisecond)
    return true
}

func BenchmarkCheckWebsites(b *testing.B) {
    urls := make([]string, 100)
    for i := 0; i < len(urls); i++ {
        urls[i] = "a url"
    }
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
```

```
    CheckWebsites(slowStubWebsiteChecker, urls)
}
}
```

The benchmark tests CheckWebsites using a slice of one hundred urls and uses a new fake implementation of WebsiteChecker. slowStubWebsiteChecker is deliberately slow. It uses time.Sleep to wait exactly twenty milliseconds and then it returns true. We use b.ResetTimer() in this test to reset the time of our test before it actually runs

When we run the benchmark using go test -bench=. (or if you're in Windows Powershell go test -bench="."):

```
pkg: github.com/gypsydave5/learn-go-with-tests/concurrency/v0
BenchmarkCheckWebsites-4           1      2249228637 ns/op
PASS
ok    github.com/gypsydave5/learn-go-with-tests/concurrency/v0      2.268s
CheckWebsites has been benchmarked at 2249228637 nanoseconds
- about two and a quarter seconds.
```

Let's try and make this faster.

Write enough code to make it pass

Now we can finally talk about concurrency which, for the purposes of the following, means "having more than one thing in progress." This is something that we do naturally everyday.

For instance, this morning I made a cup of tea. I put the kettle on and then, while I was waiting for it to boil, I got the milk out of the fridge, got the tea out of the cupboard, found my favourite mug, put the teabag into the cup and then, when the kettle had boiled, I put the water in the cup.

What I didn't do was put the kettle on and then stand there blankly staring at the kettle until it boiled, then do everything else once the kettle had boiled.

If you can understand why it's faster to make tea the first way, then you can understand how we will make CheckWebsites faster. Instead of waiting for a website to respond before sending a request to the next website, we will tell our computer to make the next request while it is waiting.

Normally in Go when we call a function doSomething() we wait for it to return (even if it has no value to return, we still wait for it to finish). We say that this operation is blocking - it makes us wait for it to finish. An operation that does not block in Go will run in a separate process

called a goroutine. Think of a process as reading down the page of Go code from top to bottom, going 'inside' each function when it gets called to read what it does. When a separate process starts, it's like another reader begins reading inside the function, leaving the original reader to carry on going down the page.

To tell Go to start a new goroutine we turn a function call into a go statement by putting the keyword go in front of it: go doSomething().

```
package concurrency
```

```
type WebsiteChecker func(string) bool
```

```
func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)

    for _, url := range urls {
        go func() {
            results[url] = wc(url)
        }()
    }

    return results
}
```

Because the only way to start a goroutine is to put go in front of a function call, we often use anonymous functions when we want to start a goroutine. An anonymous function literal looks just the same as a normal function declaration, but without a name (unsurprisingly). You can see one above in the body of the for loop.

Anonymous functions have a number of features which make them useful, two of which we're using above. Firstly, they can be executed at the same time that they're declared - this is what the () at the end of the anonymous function is doing. Secondly they maintain access to the lexical scope in which they are defined - all the variables that are available at the point when you declare the anonymous function are also available in the body of the function.

The body of the anonymous function above is just the same as the loop body was before. The only difference is that each iteration of the loop will start a new goroutine, concurrent with the current process (the WebsiteChecker function). Each goroutine will add its result to the results map.

But when we run go test:

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

```
    CheckWebsites_test.go:31: Wanted map[http://google.com:true http://blog.gypsydave5.co
```

```
FAIL
exit status 1
FAIL  github.com/gypsydave5/learn-go-with-tests/concurrency/v1      0.010s
```

A quick aside into a parallel(ism) universe...

You might not get this result. You might get a panic message that we're going to talk about in a bit. Don't worry if you got that, just keep running the test until you do get the result above. Or pretend that you did. Up to you. Welcome to concurrency: when it's not handled correctly it's hard to predict what's going to happen. Don't worry - that's why we're writing tests, to help us know when we're handling concurrency predictably.

... and we're back.

We are caught by the original test CheckWebsites, it's now returning an empty map. What went wrong?

None of the goroutines that our for loop started had enough time to add their result to the results map; the WebsiteChecker function is too fast for them, and it returns the still empty map.

To fix this we can just wait while all the goroutines do their work, and then return. Two seconds ought to do it, right?

```
package concurrency

import "time"

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)

    for _, url := range urls {
        go func() {
            results[url] = wc(url)
        }()
    }

    time.Sleep(2 * time.Second)

    return results
}
```

Now when we run the tests you get (or don't get - see above):

```
--- FAIL: TestCheckWebsites (0.00s)
    CheckWebsites_test.go:31: Wanted map[http://google.com:true http://blog.gypsydave5.co
FAIL
exit status 1
FAIL  github.com/gypsydave5/learn-go-with-tests/concurrency/v1      0.010s
```

This isn't great - why only one result? We might try and fix this by increasing the time we wait - try it if you like. It won't work. The problem here is that the variable url is reused for each iteration of the for loop - it takes a new value from urls each time. But each of our goroutines have a reference to the url variable - they don't have their own independent copy. So they're all writing the value that url has at the end of the iteration - the last url. Which is why the one result we have is the last url.

To fix this:

```
package concurrency

import (
    "time"
)

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)

    for _, url := range urls {
        go func(u string) {
            results[u] = wc(u)
        }(url)
    }

    time.Sleep(2 * time.Second)

    return results
}
```

By giving each anonymous function a parameter for the url - u - and then calling the anonymous function with the url as the argument, we make sure that the value of u is fixed as the value of url for the iteration of the loop that we're launching the goroutine in. u is a copy of the value of url, and so can't be changed.

Now if you're lucky you'll get:

```
PASS
ok  github.com/gypsydave5/learn-go-with-tests/concurrency/v1      2.012s
```

But if you're unlucky (this is more likely if you run them with the benchmark as you'll get more tries)

fatal error: concurrent map writes

```
goroutine 8 [running]:  
runtime.throw(0x12c5895, 0x15)  
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95 fp=0xc420037700 sp=0  
runtime.mapassign_faststr(0x1271d80, 0xc42007acf0, 0x12c6634, 0x17, 0x0)  
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:783 +0x4f5 fp=0xc4200377  
github.com/gpsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1(0xc42007act  
    /Users/gpsydave5/go/src/github.com/gpsydave5/learn-go-with-tests/concurrency/v3/web  
runtime.goexit()  
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1 fp=0xc4200377c8 s  
created by github.com/gpsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker  
    /Users/gpsydave5/go/src/github.com/gpsydave5/learn-go-with-tests/concurrency/v3/web  
... many more scary lines of text ...
```

This is long and scary, but all we need to do is take a breath and read the stacktrace: fatal error: concurrent map writes. Sometimes, when we run our tests, two of the goroutines write to the results map at exactly the same time. Maps in Go don't like it when more than one thing tries to write to them at once, and so fatal error.

This is a race condition, a bug that occurs when the output of our software is dependent on the timing and sequence of events that we have no control over. Because we cannot control exactly when each goroutine writes to the results map, we are vulnerable to two goroutines writing to it at the same time.

Go can help us to spot race conditions with its built in [race detector](#). To enable this feature, run the tests with the race flag: go test -race.

You should get some output that looks like this:

```
=====  
WARNING: DATA RACE  
Write at 0x00c420084d20 by goroutine 8:  
    runtime.mapassign_faststr()  
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:774 +0x0  
github.com/gpsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1()  
    /Users/gpsydave5/go/src/github.com/gpsydave5/learn-go-with-tests/concurrency/v3/webs
```

Previous write at 0x00c420084d20 by goroutine 7:

```
    runtime.mapassign_faststr()  
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:774 +0x0  
github.com/gpsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1()  
    /Users/gpsydave5/go/src/github.com/gpsydave5/learn-go-with-tests/concurrency/v3/webs
```

Goroutine 8 (**running**) created at:

```
github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker()
  /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/webs
github.com/gypsydave5/learn-go-with-tests/concurrency/v3.TestWebsiteChecker()
  /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/webs
testing.tRunner()
  /usr/local/Cellar/go/1.9.3/libexec/src/testing/testing.go:746 +0x16c
```

Goroutine 7 (**finished**) created at:

```
github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker()
  /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/webs
github.com/gypsydave5/learn-go-with-tests/concurrency/v3.TestWebsiteChecker()
  /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/webs
testing.tRunner()
  /usr/local/Cellar/go/1.9.3/libexec/src/testing/testing.go:746 +0x16c
=====
```

The details are, again, hard to read - but WARNING: DATA RACE is pretty unambiguous. Reading into the body of the error we can see two different goroutines performing writes on a map:

Write at 0x00c420084d20 by goroutine 8:

is writing to the same block of memory as

Previous write at 0x00c420084d20 by goroutine 7:

On top of that, we can see the line of code where the write is happening:

```
/Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/websiteC
```

and the line of code where goroutines 7 an 8 are started:

```
/Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/websiteC
```

Everything you need to know is printed to your terminal - all you have to do is be patient enough to read it.

Channels

We can solve this data race by coordinating our goroutines using channels. Channels are a Go data structure that can both receive and send values. These operations, along with their details, allow communication between different processes.

In this case we want to think about the communication between the parent process and each of the goroutines that it makes to do the work of running the WebsiteChecker function with the url.

```
package concurrency

type WebsiteChecker func(string) bool
type result struct {
    string
    bool
}

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
    results := make(map[string]bool)
    resultChannel := make(chan result)

    for _, url := range urls {
        go func(u string) {
            resultChannel <- result{u, wc(u)}
        }(url)
    }

    for i := 0; i < len(urls); i++ {
        r := <-resultChannel
        results[r.string] = r.bool
    }

    return results
}
```

Alongside the results map we now have a resultChannel, which we make in the same way. chan result is the type of the channel - a channel of result. The new type, result has been made to associate the return value of the WebsiteChecker with the url being checked - it's a struct of string and bool. As we don't need either value to be named, each of them is anonymous within the struct; this can be useful in when it's hard to know what to name a value.

Now when we iterate over the urls, instead of writing to the map directly we're sending a result struct for each call to wc to the resultChannel with a send statement. This uses the <- operator, taking a channel on the left and a value on the right:

```
// Send statement
resultChannel <- result{u, wc(u)}
```

The next for loop iterates once for each of the urls. Inside we're using a receive expression, which assigns a value received from a channel to a variable. This also uses the <- operator, but with the two operands now reversed: the channel is now on the right and the variable that we're assigning to is on the left:

```
// Receive expression
r := <-resultChannel
```

We then use the result received to update the map.

By sending the results into a channel, we can control the timing of each write into the results map, ensuring that it happens one at a time. Although each of the calls of `wc`, and each send to the result channel, is happening in parallel inside its own process, each of the results is being dealt with one at a time as we take values out of the result channel with the receive expression.

We have parallelized the part of the code that we wanted to make faster, while making sure that the part that cannot happen in parallel still happens linearly. And we have communicated across the multiple processes involved by using channels.

When we run the benchmark:

```
pkg: github.com/gypsydave5/learn-go-with-tests/concurrency/v2
BenchmarkCheckWebsites-8          100      23406615 ns/op
PASS
ok    github.com/gypsydave5/learn-go-with-tests/concurrency/v2      2.377s
23406615 nanoseconds - 0.023 seconds, about one hundred times as
fast as original function. A great success.
```

Wrapping up

This exercise has been a little lighter on the TDD than usual. In a way we've been taking part in one long refactoring of the `CheckWebsites` function; the inputs and outputs never changed, it just got faster. But the tests we had in place, as well as the benchmark we wrote, allowed us to refactor `CheckWebsites` in a way that maintained confidence that the software was still working, while demonstrating that it had actually become faster.

In making it faster we learned about

- goroutines, the basic unit of concurrency in Go, which let us check more than one website at the same time.
- anonymous functions, which we used to start each of the concurrent processes that check websites.
- channels, to help organize and control the communication between the different processes, allowing us to avoid a race condition bug.
- the race detector which helped us debug problems with concurrent code

Make it fast

One formulation of an agile way of building software, often misattributed to Kent Beck, is:

Make it work, make it right, make it fast

Where 'work' is making the tests pass, 'right' is refactoring the code, and 'fast' is optimizing the code to make it, for example, run quickly. We can only 'make it fast' once we've made it work and made it right. We were lucky that the code we were given was already demonstrated to be working, and didn't need to be refactored. We should never try to 'make it fast' before the other two steps have been performed because

Premature optimization is the root of all evil -- Donald Knuth

Select

You can find all the code for this chapter here

You have been asked to make a function called WebsiteRacer which takes two URLs and "races" them by hitting them with an HTTP GET and returning the URL which returned first. If none of them return within 10 seconds then it should return an error.

For this, we will be using:

- net/http to make the HTTP calls.
- net/http/httptest to help us test them.
- goroutines.
- select to synchronise processes.

Write the test first

Let's start with something naive to get us going.

```
func TestRacer(t *testing.T) {
    slowURL := "http://www.facebook.com"
    fastURL := "http://www.quii.dev"

    want := fastURL
    got := Racer(slowURL, fastURL)

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

We know this isn't perfect and has problems, but it's a start. It's important not to get too hung-up on getting things perfect first time.

Try to run the test

```
./racer_test.go:14:9: undefined: Racer
```

Write the minimal amount of code for the test to run and check the failing test output

```
func Racer(a, b string) (winner string) {
    return
}
racer_test.go:25: got "", want 'http://www.quii.dev'
```

Write enough code to make it pass

```
func Racer(a, b string) (winner string) {
    startA := time.Now()
    http.Get(a)
    aDuration := time.Since(startA)

    startB := time.Now()
    http.Get(b)
    bDuration := time.Since(startB)

    if aDuration < bDuration {
        return a
    }

    return b
}
```

For each URL:

1. We use `time.Now()` to record just before we try and get the URL.
2. Then we use `http.Get` to try and perform an HTTP GET request against the URL. This function returns an `http.Response` and an error but so far we are not interested in these values.
3. `time.Since` takes the start time and returns a `time.Duration` of the difference.

Once we have done this we simply compare the durations to see which is the quickest.

Problems

This may or may not make the test pass for you. The problem is we're reaching out to real websites to test our own logic.

Testing code that uses HTTP is so common that Go has tools in the standard library to help you test it.

In the mocking and dependency injection chapters, we covered how ideally we don't want to be relying on external services to test our code because they can be

- Slow
- Flaky
- Can't test edge cases

In the standard library, there is a package called [net/http/httpptest](#) which enables users to easily create a mock HTTP server.

Let's change our tests to use mocks so we have reliable servers to test against that we can control.

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

    slowServer := httpptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(20 * time.Millisecond)
        w.WriteHeader(http.StatusOK)
    }))

    fastServer := httpptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
    }))

    slowURL := slowServer.URL
    fastURL := fastServer.URL

    want := fastURL
    got := Racer(slowURL, fastURL)

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

    slowServer.Close()
    fastServer.Close()
}
```

The syntax may look a bit busy but just take your time.

httptest.NewServer takes an http.HandlerFunc which we are sending in via an anonymous function.

http.HandlerFunc is a type that looks like this: type HandlerFunc func(ResponseWriter, *Request).

All it's really saying is it needs a function that takes a ResponseWriter and a Request, which is not too surprising for an HTTP server.

It turns out there's really no extra magic here, **this is also how you would write a real HTTP server in Go**. The only difference is we are wrapping it in an htttpertest.NewServer which makes it easier to use with testing, as it finds an open port to listen on and then you can close it when you're done with your test.

Inside our two servers, we make the slow one have a short time.Sleep when we get a request to make it slower than the other one. Both servers then write an OK response with w.WriteHeader(http.StatusOK) back to the caller.

If you re-run the test it will definitely pass now and should be faster. Play with these sleeps to deliberately break the test.

Refactor

We have some duplication in both our production code and test code.

```
func Racer(a, b string) (winner string) {
    aDuration := measureResponseTime(a)
    bDuration := measureResponseTime(b)

    if aDuration < bDuration {
        return a
    }

    return b
}

func measureResponseTime(url string) time.Duration {
    start := time.Now()
    http.Get(url)
    return time.Since(start)
}
```

This DRY-ing up makes our Racer code a lot easier to read.

```
func TestRacer(t *testing.T) {
    slowServer := makeDelayedServer(20 * time.Millisecond)
```

```
fastServer := makeDelayedServer(0 * time.Millisecond)

defer slowServer.Close()
defer fastServer.Close()

slowURL := slowServer.URL
fastURL := fastServer.URL

want := fastURL
got := Racer(slowURL, fastURL)

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

func makeDelayedServer(delay time.Duration) *httptest.Server {
    return httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(delay)
        w.WriteHeader(http.StatusOK)
    }))
}
```

We've refactored creating our fake servers into a function called `makeDelayedServer` to move some uninteresting code out of the test and reduce repetition.

defer

By prefixing a function call with `defer` it will now call that function at the end of the containing function.

Sometimes you will need to clean up resources, such as closing a file or in our case closing a server so that it does not continue to listen to a port.

You want this to execute at the end of the function, but keep the instruction near where you created the server for the benefit of future readers of the code.

Our refactoring is an improvement and is a reasonable solution given the Go features covered so far, but we can make the solution simpler.

Synchronising processes

- Why are we testing the speeds of the websites one after another when Go is great at concurrency? We should be able to check

both at the same time.

- We don't really care about the exact response times of the requests, we just want to know which one comes back first.

To do this, we're going to introduce a new construct called select which helps us synchronise processes really easily and clearly.

```
func Racer(a, b string) (winner string) {
    select {
        case <-ping(a):
            return a
        case <-ping(b):
            return b
    }
}

func ping(url string) chan struct{} {
    ch := make(chan struct{})
    go func() {
        http.Get(url)
        close(ch)
    }()
    return ch
}
```

ping We have defined a function ping which creates a chan struct{} and returns it.

In our case, we don't care what type is sent to the channel, we just want to signal we are done and closing the channel works perfectly!

Why struct{} and not another type like a bool? Well, a chan struct{} is the smallest data type available from a memory perspective so we get no allocation versus a bool. Since we are closing and not sending anything on the chan, why allocate anything?

Inside the same function, we start a goroutine which will send a signal into that channel once we have completed http.Get(url).

Always make channels Notice how we have to use make when creating a channel; rather than say var ch chan struct{}. When you use var the variable will be initialised with the "zero" value of the type. So for string it is "", int it is 0, etc.

For channels the zero value is nil and if you try and send to it with <- it will block forever because you cannot send to nil channels

[You can see this in action in The Go Playground](#)

select You'll recall from the concurrency chapter that you can wait for values to be sent to a channel with `myVar := <-ch`. This is a blocking call, as you're waiting for a value.

`select` allows you to wait on multiple channels. The first one to send a value "wins" and the code underneath the case is executed.

We use `ping` in our `select` to set up two channels, one for each of our URLs. Whichever one writes to its channel first will have its code executed in the `select`, which results in its URL being returned (and being the winner).

After these changes, the intent behind our code is very clear and the implementation is actually simpler.

Timeouts

Our final requirement was to return an error if Racer takes longer than 10 seconds.

Write the test first

```
func TestRacer(t *testing.T) {
    t.Run("compares speeds of servers, returning the url of the fastest one", func(t *testing.T) {
        slowServer := makeDelayedServer(20 * time.Millisecond)
        fastServer := makeDelayedServer(0 * time.Millisecond)

        defer slowServer.Close()
        defer fastServer.Close()

        slowURL := slowServer.URL
        fastURL := fastServer.URL

        want := fastURL
        got, err := Racer(slowURL, fastURL)

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

    t.Run("returns an error if a server doesn't respond within 10s", func(t *testing.T) {
        serverA := makeDelayedServer(11 * time.Second)
        serverB := makeDelayedServer(12 * time.Second)

        defer serverA.Close()
        defer serverB.Close()
    })
}
```

```
_ , err := Racer(serverA.URL, serverB.URL)

if err == nil {
    t.Error("expected an error but didn't get one")
}
})
}
```

We've made our test servers take longer than 10s to return to exercise this scenario and we are expecting Racer to return two values now, the winning URL (which we ignore in this test with `_`) and an error.

Note that we've also handled the error return in our original test, we're using `_` for now to ensure the tests will run.

Try to run the test

```
./racer_test.go:37:10: assignment mismatch: 2 variables but Racer
returns 1 value
```

Write the minimal amount of code for the test to run and check the failing test output

```
func Racer(a, b string) (winner string, error error) {
    select {
        case <-ping(a):
            return a, nil
        case <-ping(b):
            return b, nil
    }
}
```

Change the signature of Racer to return the winner and an error. Return nil for our happy cases.

The compiler will complain about your first test only looking for one value so change that line to `got, err := Racer(slowURL, fastURL)`, knowing that we should check we don't get an error in our happy scenario.

If you run it now after 11 seconds it will fail.

```
--- FAIL: TestRacer (12.00s)
--- FAIL: TestRacer/returns_an_error_if_a_server_doesn't_respond_within_10s (12.00s)
racer_test.go:40: expected an error but didn't get one
```

Write enough code to make it pass

```
func Racer(a, b string) (winner string, error error) {
    select {
        case <-ping(a):
            return a, nil
        case <-ping(b):
            return b, nil
        case <-time.After(10 * time.Second):
            return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
    }
}
```

time.After is a very handy function when using select. Although it didn't happen in our case you can potentially write code that blocks forever if the channels you're listening on never return a value. time.After returns a chan (like ping) and will send a signal down it after the amount of time you define.

For us this is perfect; if a or b manage to return they win, but if we get to 10 seconds then our time.After will send a signal and we'll return an error.

Slow tests

The problem we have is that this test takes 10 seconds to run. For such a simple bit of logic, this doesn't feel great.

What we can do is make the timeout configurable. So in our test, we can have a very short timeout and then when the code is used in the real world it can be set to 10 seconds.

```
func Racer(a, b string, timeout time.Duration) (winner string, error error) {
    select {
        case <-ping(a):
            return a, nil
        case <-ping(b):
            return b, nil
        case <-time.After(timeout):
            return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
    }
}
```

Our tests now won't compile because we're not supplying a timeout.

Before rushing in to add this default value to both our tests let's listen to them.

- Do we care about the timeout in the "happy" test?

-
- The requirements were explicit about the timeout.

Given this knowledge, let's do a little refactoring to be sympathetic to both our tests and the users of our code.

```
var tenSecondTimeout = 10 * time.Second

func Racer(a, b string) (winner string, error error) {
    return ConfigurableRacer(a, b, tenSecondTimeout)
}

func ConfigurableRacer(a, b string, timeout time.Duration) (winner string, error error) {
    select {
        case <-ping(a):
            return a, nil
        case <-ping(b):
            return b, nil
        case <-time.After(timeout):
            return "", fmt.Errorf("timed out waiting for %s and %s", a, b)
    }
}
```

Our users and our first test can use Racer (which uses ConfigurableRacer under the hood) and our sad path test can use ConfigurableRacer.

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

    t.Run("compares speeds of servers, returning the url of the fastest one", func(t *testing.T) {
        slowServer := makeDelayedServer(20 * time.Millisecond)
        fastServer := makeDelayedServer(0 * time.Millisecond)

        defer slowServer.Close()
        defer fastServer.Close()

        slowURL := slowServer.URL
        fastURL := fastServer.URL

        want := fastURL
        got, err := Racer(slowURL, fastURL)

        if err != nil {
            t.Fatalf("did not expect an error but got one %v", err)
        }

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

```
t.Run("returns an error if a server doesn't respond within the specified time", func(t *testing.T) {
    server := makeDelayedServer(25 * time.Millisecond)

    defer server.Close()

    _, err := ConfigurableRacer(server.URL, server.URL, 20*time.Millisecond)

    if err == nil {
        t.Error("expected an error but didn't get one")
    }
})
})
```

I added one final check on the first test to verify we don't get an error.

Wrapping up

select

- Helps you wait on multiple channels.
- Sometimes you'll want to include time.After in one of your cases to prevent your system blocking forever.

httptest

- A convenient way of creating test servers so you can have reliable and controllable tests.
- Uses the same interfaces as the "real" net/http servers which is consistent and less for you to learn.

Reflection

You can find all the code for this chapter here

From Twitter

golang challenge: write a function walk(x interface{}, fn func(string)) which takes a struct x and calls fn for all strings fields found inside. difficulty level: recursively.

To do this we will need to use reflection.

Reflection in computing is the ability of a program to examine its own structure, particularly through types; it's a form of metaprogramming. It's also a great source of confusion.

From [The Go Blog: Reflection](#)

What is interface{ }?

We have enjoyed the type-safety that Go has offered us in terms of functions that work with known types, such as string, int and our own types like BankAccount.

This means that we get some documentation for free and the compiler will complain if you try and pass the wrong type to a function.

You may come across scenarios though where you want to write a function where you don't know the type at compile time.

Go lets us get around this with the type interface{} which you can think of as just any type (in fact, in Go any is an [alias](#) for interface{}).

So walk(x interface{}, fn func(string)) will accept any value for x.

So why not use interface{} for everything and have really flexible functions?

- As a user of a function that takes interface{} you lose type safety. What if you meant to pass Herd.species of type string into a function but instead did Herd.count which is an int? The compiler won't be able to inform you of your mistake. You also have no idea what you're allowed to pass to a function. Knowing that a function takes a UserService for instance is very useful.
- As a writer of such a function, you have to be able to inspect anything that has been passed to you and try and figure out what the type is and what you can do with it. This is done using reflection. This can be quite clumsy and difficult to read and is generally less performant (as you have to do checks at runtime).

In short only use reflection if you really need to.

If you want polymorphic functions, consider if you could design it around an interface (not interface{}, confusingly) so that users can use your function with multiple types if they implement whatever methods you need for your function to work.

Our function will need to be able to work with lots of different things. As always we'll take an iterative approach, writing tests for each new thing we want to support and refactoring along the way until we're done.

Write the test first

We'll want to call our function with a struct that has a string field in it (x). Then we can spy on the function (fn) passed in to see if it is called.

```
func TestWalk(t *testing.T) {
    expected := "Chris"
    var got []string

    x := struct {
        Name string
    }{expected}

    walk(x, func(input string) {
        got = append(got, input)
    })

    if len(got) != 1 {
        t.Errorf("wrong number of function calls, got %d want %d", len(got), 1)
    }
}
```

- We want to store a slice of strings (got) which stores which strings were passed into fn by walk. Often in previous chapters, we have made dedicated types for this to spy on function/method invocations but in this case, we can just pass in an anonymous function for fn that closes over got.
- We use an anonymous struct with a Name field of type string to go for the simplest "happy" path.
- Finally, call walk with x and the spy and for now just check the length of got, we'll be more specific with our assertions once we've got something very basic working.

Try to run the test

```
./reflection_test.go:21:2: undefined: walk
```

Write the minimal amount of code for the test to run and check the failing test output

We need to define walk

```
func walk(x interface{}, fn func(input string)) {
}
```

Try and run the test again

```
==== RUN TestWalk
--- FAIL: TestWalk (0.00s)
    reflection_test.go:19: wrong number of function calls, got 0 want 1
FAIL
```

Write enough code to make it pass

We can call the spy with any string to make this pass.

```
func walk(x interface{}, fn func(input string)) {
    fn("I still can't believe South Korea beat Germany 2-0 to put them last in their group")
}
```

The test should now be passing. The next thing we'll need to do is make a more specific assertion on what our fn is being called with.

Write the test first

Add the following to the existing test to check the string passed to fn is correct

```
if got[0] != expected {
    t.Errorf("got %q, want %q", got[0], expected)
}
```

Try to run the test

```
==== RUN TestWalk
--- FAIL: TestWalk (0.00s)
    reflection_test.go:23: got 'I still can't believe South Korea beat Germany 2-0 to put them last
FAIL
```

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
    val := reflect.ValueOf(x)
    field := val.Field(0)
    fn(field.String())
}
```

This code is very unsafe and very naive, but remember: our goal when we are in "red" (the tests failing) is to write the smallest amount of code possible. We then write more tests to address our concerns.

We need to use reflection to have a look at x and try and look at its properties.

The [reflect package](#) has a function `ValueOf` which returns us a `Value` of a given variable. This has ways for us to inspect a value, including its fields which we use on the next line.

We then make some very optimistic assumptions about the value passed in:

- We look at the first and only field. However, there may be no fields at all, which would cause a panic.
- We then call `String()`, which returns the underlying value as a string. However, this would be wrong if the field was something other than a string.

Refactor

Our code is passing for the simple case but we know our code has a lot of shortcomings.

We're going to be writing a number of tests where we pass in different values and checking the array of strings that `fn` was called with.

We should refactor our test into a table based test to make this easier to continue testing new scenarios.

```
func TestWalk(t *testing.T) {
    cases := []struct {
        Name      string
        Input     interface{}
        ExpectedCalls []string
    }{
        {
            "struct with one string field",
            struct {
                Name string
            }{"Chris"},
            []string{"Chris"},
        },
    }

    for _, test := range cases {
        t.Run(test.Name, func(t *testing.T) {
            var got []string
            walk(test.Input, func(input string) {
                got = append(got, input)
            })
            if !reflect.DeepEqual(got, test.ExpectedCalls) {
                t.Errorf("got %v, want %v", got, test.ExpectedCalls)
            }
        })
    }
}
```

```
    })

    if !reflect.DeepEqual(got, test.ExpectedCalls) {
        t.Errorf("got %v, want %v", got, test.ExpectedCalls)
    }
}

}
```

Now we can easily add a scenario to see what happens if we have more than one string field.

Write the test first

Add the following scenario to the cases.

```
{
    "struct with two string fields",
    struct {
        Name string
        City string
    }{"Chris", "London"},
    []string{"Chris", "London"},
}
```

Try to run the test

```
==== RUN TestWalk/struct_with_two_string_fields
--- FAIL: TestWalk/struct_with_two_string_fields (0.00s)
    reflection_test.go:40: got [Chris], want [Chris London]
```

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
    val := reflect.ValueOf(x)

    for i := 0; i < val.NumField(); i++ {
        field := val.Field(i)
        fn(field.String())
    }
}
```

val has a method NumField which returns the number of fields in the value. This lets us iterate over the fields and call fn which passes our test.

Refactor

It doesn't look like there's any obvious refactors here that would improve the code so let's press on.

The next shortcoming in walk is that it assumes every field is a string. Let's write a test for this scenario.

Write the test first

Add the following case

```
{  
    "struct with non string field",  
    struct {  
        Name string  
        Age int  
    }{"Chris", 33},  
    []string{"Chris"},  
},
```

Try to run the test

```
==== RUN TestWalk/struct_with_non_string_field  
--- FAIL: TestWalk/struct_with_non_string_field (0.00s)  
    reflection_test.go:46: got [Chris <int Value>], want [Chris]
```

Write enough code to make it pass

We need to check that the type of the field is a string.

```
func walk(x interface{}, fn func(input string)) {  
    val := reflect.ValueOf(x)  
  
    for i := 0; i < val.NumField(); i++ {  
        field := val.Field(i)  
  
        if field.Kind() == reflect.String {  
            fn(field.String())  
        }  
    }  
}
```

We can do that by checking its [Kind](#).

Refactor

Again it looks like the code is reasonable enough for now.

The next scenario is what if it isn't a "flat" struct? In other words, what happens if we have a struct with some nested fields?

Write the test first

We have been using the anonymous struct syntax to declare types ad-hocly for our tests so we could continue to do that like so

```
{  
    "nested fields",  
    struct {  
        Name string  
        Profile struct {  
            Age int  
            City string  
        }  
    }{"Chris", struct {  
        Age int  
        City string  
    }{33, "London"}},  
    []string{"Chris", "London"},  
},
```

But we can see that when you get inner anonymous structs the syntax gets a little messy. [There is a proposal to make it so the syntax would be nicer.](#)

Let's just refactor this by making a known type for this scenario and reference it in the test. There is a little indirection in that some of the code for our test is outside the test but readers should be able to infer the structure of the struct by looking at the initialisation.

Add the following type declarations somewhere in your test file

```
type Person struct {  
    Name string  
    Profile Profile  
}  
  
type Profile struct {  
    Age int  
    City string  
}
```

Now we can add this to our cases which reads a lot clearer than before

```
{  
    "nested fields",  
    Person{  
        "Chris",  
        Profile{33, "London"},  
    },  
    []string{"Chris", "London"},  
},
```

Try to run the test

```
==== RUN TestWalk/Nested_fields  
--- FAIL: TestWalk/nested_fields (0.00s)  
    reflection_test.go:54: got [Chris], want [Chris London]
```

The problem is we're only iterating on the fields on the first level of the type's hierarchy.

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {  
    val := reflect.ValueOf(x)  
  
    for i := 0; i < val.NumField(); i++ {  
        field := val.Field(i)  
  
        if field.Kind() == reflect.String {  
            fn(field.String())  
        }  
  
        if field.Kind() == reflect.Struct {  
            walk(field.Interface(), fn)  
        }  
    }  
}
```

The solution is quite simple, we again inspect its Kind and if it happens to be a struct we just call walk again on that inner struct.

Refactor

```
func walk(x interface{}, fn func(input string)) {  
    val := reflect.ValueOf(x)  
  
    for i := 0; i < val.NumField(); i++ {  
        field := val.Field(i)
```

```
switch field.Kind() {
    case reflect.String:
        fn(field.String())
    case reflect.Struct:
        walk(field.Interface(), fn)
    }
}
```

When you're doing a comparison on the same value more than once generally refactoring into a switch will improve readability and make your code easier to extend.

What if the value of the struct passed in is a pointer?

Write the test first

Add this case

```
{
    "pointers to things",
    &Person{
        "Chris",
        Profile{33, "London"},
    },
    []string{"Chris", "London"},
},
```

Try to run the test

```
==== RUN TestWalk/pointers_to_things
panic: reflect: call of reflect.Value.NumField on ptr Value [recovered]
    panic: reflect: call of reflect.Value.NumField on ptr Value
```

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
    val := reflect.ValueOf(x)

    if val.Kind() == reflect.Pointer {
        val = val.Elem()
    }

    for i := 0; i < val.NumField(); i++ {
        field := val.Field(i)
```

```
switch field.Kind() {
    case reflect.String:
        fn(field.String())
    case reflect.Struct:
        walk(field.Interface(), fn)
    }
}
```

You can't use NumField on a pointer Value, we need to extract the underlying value before we can do that by using Elem().

Refactor

Let's encapsulate the responsibility of extracting the reflect.Value from a given interface{} into a function.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    for i := 0; i < val.NumField(); i++ {
        field := val.Field(i)

        switch field.Kind() {
            case reflect.String:
                fn(field.String())
            case reflect.Struct:
                walk(field.Interface(), fn)
        }
    }
}

func getValue(x interface{}) reflect.Value {
    val := reflect.ValueOf(x)

    if val.Kind() == reflect.Pointer {
        val = val.Elem()
    }

    return val
}
```

This actually adds more code but I feel the abstraction level is right.

- Get the reflect.Value of x so I can inspect it, I don't care how.

-
- Iterate over the fields, doing whatever needs to be done depending on its type.

Next, we need to cover slices.

Write the test first

```
{  
    "slices",  
    []Profile {  
        {33, "London"},  
        {34, "Reykjavík"},  
    },  
    []string{"London", "Reykjavík"},  
},
```

Try to run the test

```
==== RUN TestWalk/slices  
panic: reflect: call of reflect.Value.NumField on slice Value [recovered]  
    panic: reflect: call of reflect.Value.NumField on slice Value
```

Write the minimal amount of code for the test to run and check the failing test output

This is similar to the pointer scenario before, we are trying to call NumField on our reflect.Value but it doesn't have one as it's not a struct.

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {  
    val := getValue(x)  
  
    if val.Kind() == reflect.Slice {  
        for i := 0; i < val.Len(); i++ {  
            walk(val.Index(i).Interface(), fn)  
        }  
        return  
    }  
  
    for i := 0; i < val.NumField(); i++ {  
        field := val.Field(i)  
  
        switch field.Kind() {
```

```
    case reflect.String:
        fn(field.String())
    case reflect.Struct:
        walk(field.Interface(), fn)
    }
}
}
```

Refactor

This works but it's yucky. No worries, we have working code backed by tests so we are free to tinker all we like.

If you think a little abstractly, we want to call walk on either

- Each field in a struct
- Each thing in a slice

Our code at the moment does this but doesn't reflect it very well. We just have a check at the start to see if it's a slice (with a return to stop the rest of the code executing) and if it's not we just assume it's a struct.

Let's rework the code so instead we check the type first and then do our work.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    switch val.Kind() {
    case reflect.Struct:
        for i := 0; i < val.NumField(); i++ {
            walk(val.Field(i).Interface(), fn)
        }
    case reflect.Slice:
        for i := 0; i < val.Len(); i++ {
            walk(val.Index(i).Interface(), fn)
        }
    case reflect.String:
        fn(val.String())
    }
}
```

Looking much better! If it's a struct or a slice we iterate over its values calling walk on each one. Otherwise, if it's a reflect.String we can call fn.

Still, to me it feels like it could be better. There's repetition of the

operation of iterating over fields/values and then calling walk but conceptually they're the same.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    numberOfValues := 0
    var getField func(int) reflect.Value

    switch val.Kind() {
        case reflect.String:
            fn(val.String())
        case reflect.Struct:
            numberOfValues = val.NumField()
            getField = val.Field
        case reflect.Slice:
            numberOfValues = val.Len()
            getField = val.Index
    }

    for i := 0; i < numberOfValues; i++ {
        walk(getField(i).Interface(), fn)
    }
}
```

If the value is a `reflect.String` then we just call `fn` like normal.

Otherwise, our switch will extract out two things depending on the type

- How many fields there are
- How to extract the Value (Field or Index)

Once we've determined those things we can iterate through `numberOfValues` calling `walk` with the result of the `getField` function.

Now we've done this, handling arrays should be trivial.

Write the test first

Add to the cases

```
{
    "arrays",
    [2]Profile {
        {33, "London"},
        {34, "Reykjavík"},
    },
    []string{"London", "Reykjavík"},
```

```
},
```

Try to run the test

```
==== RUN TestWalk/arrays
--- FAIL: TestWalk/arrays (0.00s)
    reflection_test.go:78: got [], want [London Reykjavík]
```

Write enough code to make it pass

Arrays can be handled the same way as slices, so just add it to the case with a comma

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    numberOfValues := 0
    var getField func(int) reflect.Value

    switch val.Kind() {
        case reflect.String:
            fn(val.String())
        case reflect.Struct:
            numberOfValues = val.NumField()
            getField = val.Field
        case reflect.Slice, reflect.Array:
            numberOfValues = val.Len()
            getField = val.Index
    }

    for i := 0; i < numberOfValues; i++ {
        walk(getField(i).Interface(), fn)
    }
}
```

The next type we want to handle is map.

Write the test first

```
{
    "maps",
    map[string]string{
        "Cow": "Moo",
        "Sheep": "Baa",
    },
    []string{"Moo", "Baa"},
```

},

Try to run the test

```
==== RUN TestWalk/maps
--- FAIL: TestWalk/maps (0.00s)
    reflection_test.go:86: got [], want [Moo Baa]
```

Write enough code to make it pass

Again if you think a little abstractly you can see that map is very similar to struct, it's just the keys are unknown at compile time.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    numberOfValues := 0
    var getField func(int) reflect.Value

    switch val.Kind() {
        case reflect.String:
            fn(val.String())
        case reflect.Struct:
            numberOfValues = val.NumField()
            getField = val.Field
        case reflect.Slice, reflect.Array:
            numberOfValues = val.Len()
            getField = val.Index
        case reflect.Map:
            for _, key := range val.MapKeys() {
                walk(val.MapIndex(key).Interface(), fn)
            }
    }

    for i := 0; i < numberOfValues; i++ {
        walk(getField(i).Interface(), fn)
    }
}
```

However, by design you cannot get values out of a map by index. It's only done by key, so that breaks our abstraction, darn.

Refactor

How do you feel right now? It felt like maybe a nice abstraction at the time but now the code feels a little wonky.

This is OK! Refactoring is a journey and sometimes we will make mistakes. A major point of TDD is it gives us the freedom to try these things out.

By taking small steps backed by tests this is in no way an irreversible situation. Let's just put it back to how it was before the refactor.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    walkValue := func(value reflect.Value) {
        walk(value.Interface(), fn)
    }

    switch val.Kind() {
        case reflect.String:
            fn(val.String())
        case reflect.Struct:
            for i := 0; i < val.NumField(); i++ {
                walkValue(val.Field(i))
            }
        case reflect.Slice, reflect.Array:
            for i := 0; i < val.Len(); i++ {
                walkValue(val.Index(i))
            }
        case reflect.Map:
            for _, key := range val.MapKeys() {
                walkValue(val.MapIndex(key))
            }
    }
}
```

We've introduced `walkValue` which DRYs up the calls to `walk` inside our switch so that they only have to extract out the `reflect.Values` from `val`.

One final problem

Remember that maps in Go do not guarantee order. So your tests will sometimes fail because we assert that the calls to `fn` are done in a particular order.

To fix this, we'll need to move our assertion with the maps to a new test where we do not care about the order.

```
t.Run("with maps", func(t *testing.T) {
    aMap := map[string]string{
        "Cow": "Moo",
```

```
        "Sheep": "Baa",
    }

var got []string
walk(aMap, func(input string) {
    got = append(got, input)
})

assertContains(t, got, "Moo")
assertContains(t, got, "Baa")
})
```

Here is how assertContains is defined

```
func assertContains(t testing.TB, haystack []string, needle string) {
    t.Helper()
    contains := false
    for _, x := range haystack {
        if x == needle {
            contains = true
        }
    }
    if !contains {
        t.Errorf("expected %v to contain %q but it didn't", haystack, needle)
    }
}
```

Since we have extracted maps into a new test, we haven't seen the failure message. Intentionally break the with maps test here so that you can check the error message, then fix it again so all tests are passing.

The next type we want to handle is chan.

Write the test first

```
t.Run("with channels", func(t *testing.T) {
    aChannel := make(chan Profile)

    go func() {
        aChannel <- Profile{33, "Berlin"}
        aChannel <- Profile{34, "Katowice"}
        close(aChannel)
    }()
}

var got []string
want := []string{"Berlin", "Katowice"}
```

```
    walk(aChannel, func(input string) {
        got = append(got, input)
    })

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

Try to run the test

```
--- FAIL: TestWalk (0.00s)
--- FAIL: TestWalk/with_channels (0.00s)
    reflection_test.go:115: got [], want [Berlin Katowice]
```

Write enough code to make it pass

We can iterate through all values sent through channel until it was closed with Recv()

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    walkValue := func(value reflect.Value) {
        walk(value.Interface(), fn)
    }

    switch val.Kind() {
        case reflect.String:
            fn(val.String())
        case reflect.Struct:
            for i := 0; i < val.NumField(); i++ {
                walkValue(val.Field(i))
            }
        case reflect.Slice, reflect.Array:
            for i := 0; i < val.Len(); i++ {
                walkValue(val.Index(i))
            }
        case reflect.Map:
            for _, key := range val.MapKeys() {
                walkValue(val.MapIndex(key))
            }
        case reflect.Chan:
            for v, ok := val.Recv(); ok; v, ok = val.Recv() {
```

```
        walkValue(v)
    }
}
}
```

The next type we want to handle is func.

Write the test first

```
t.Run("with function", func(t *testing.T) {
    aFunction := func() (Profile, Profile) {
        return Profile{33, "Berlin"}, Profile{34, "Katowice"}
    }

    var got []string
    want := []string{"Berlin", "Katowice"}

    walk(aFunction, func(input string) {
        got = append(got, input)
    })

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

Try to run the test

```
--- FAIL: TestWalk (0.00s)
--- FAIL: TestWalk/with_function (0.00s)
    reflection_test.go:132: got [], want [Berlin Katowice]
```

Write enough code to make it pass

Non zero-argument functions do not seem to make a lot of sense in this scenario. But we should allow for arbitrary return values.

```
func walk(x interface{}, fn func(input string)) {
    val := getValue(x)

    walkValue := func(value reflect.Value) {
        walk(value.Interface(), fn)
    }

    switch val.Kind() {
        case reflect.String:
```

```
fn(val.String())
case reflect.Struct:
    for i := 0; i < val.NumField(); i++ {
        walkValue(val.Field(i))
    }
case reflect.Slice, reflect.Array:
    for i := 0; i < val.Len(); i++ {
        walkValue(val.Index(i))
    }
case reflect.Map:
    for _, key := range val.MapKeys() {
        walkValue(val.MapIndex(key))
    }
case reflect.Chan:
    for v, ok := val.Recv(); ok; v, ok = val.Recv() {
        walkValue(v)
    }
case reflect.Func:
    valFnResult := val.Call(nil)
    for _, res := range valFnResult {
        walkValue(res)
    }
}
```

Wrapping up

- Introduced some concepts from the reflect package.
- Used recursion to traverse arbitrary data structures.
- Did an in retrospect bad refactor but didn't get too upset about it. By working iteratively with tests it's not such a big deal.
- This only covered a small aspect of reflection. [The Go blog has an excellent post covering more details.](#)
- Now that you know about reflection, do your best to avoid using it.

Sync

[You can find all the code for this chapter here](#)

We want to make a counter which is safe to use concurrently.

We'll start with an unsafe counter and verify its behaviour works in a single-threaded environment.

Then we'll exercise it's unsafeness, with multiple goroutines trying to use the counter via a test, and fix it.

Write the test first

We want our API to give us a method to increment the counter and then retrieve its value.

```
func TestCounter(t *testing.T) {
    t.Run("incrementing the counter 3 times leaves it at 3", func(t *testing.T) {
        counter := Counter{}
        counter.Inc()
        counter.Inc()
        counter.Inc()

        if counter.Value() != 3 {
            t.Errorf("got %d, want %d", counter.Value(), 3)
        }
    })
}
```

Try to run the test

```
./sync_test.go:9:14: undefined: Counter
```

Write the minimal amount of code for the test to run and check the failing test output

Let's define Counter.

```
type Counter struct {
}
```

Try again and it fails with the following

```
./sync_test.go:14:10: counter.Inc undefined (type Counter has no field or method Inc)
./sync_test.go:18:13: counter.Value undefined (type Counter has no field or method Value)
```

So to finally make the test run we can define those methods

```
func (c *Counter) Inc() {
}

func (c *Counter) Value() int {
    return 0
}
```

It should now run and fail

```
==== RUN TestCounter
==== RUN TestCounter/incrementing_the_counter_3_times_leaves_it_at_3
--- FAIL: TestCounter (0.00s)
    --- FAIL: TestCounter/incrementing_the_counter_3_times_leaves_it_at_3 (0.00s)
        sync_test.go:27: got 0, want 3
```

Write enough code to make it pass

This should be trivial for Go experts like us. We need to keep some state for the counter in our datatype and then increment it on every Inc call

```
type Counter struct {
    value int
}

func (c *Counter) Inc() {
    c.value++
}

func (c *Counter) Value() int {
    return c.value
}
```

Refactor

There's not a lot to refactor but given we're going to write more tests around Counter we'll write a small assertion function assertCount so the test reads a bit clearer.

```
t.Run("incrementing the counter 3 times leaves it at 3", func(t *testing.T) {
    counter := Counter{}
    counter.Inc()
    counter.Inc()
    counter.Inc()

    assertCounter(t, counter, 3)
})

func assertCounter(t testing.TB, got Counter, want int) {
    t.Helper()
    if got.Value() != want {
        t.Errorf("got %d, want %d", got.Value(), want)
    }
}
```

Next steps

That was easy enough but now we have a requirement that it must be safe to use in a concurrent environment. We will need to write a failing test to exercise this.

Write the test first

```
t.Run("it runs safely concurrently", func(t *testing.T) {
    wantedCount := 1000
    counter := Counter{}

    var wg sync.WaitGroup
    wg.Add(wantedCount)

    for i := 0; i < wantedCount; i++ {
        go func() {
            counter.Inc()
            wg.Done()
        }()
    }
    wg.Wait()

    assertCounter(t, counter, wantedCount)
})
```

This will loop through our wantedCount and fire a goroutine to call counter.Inc().

We are using `sync.WaitGroup` which is a convenient way of synchronising concurrent processes.

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

By waiting for wg.Wait() to finish before making our assertions we can be sure all of our goroutines have attempted to Inc the Counter.

Try to run the test

```
==== RUN TestCounter/it_runs_safely_in_a_concurrent_enivornment
--- FAIL: TestCounter (0.00s)
    --- FAIL: TestCounter/it_runs_safely_in_a_concurrent_enivornment (0.00s)
        sync_test.go:26: got 939, want 1000
```

FAIL

The test will probably fail with a different number, but nonetheless it demonstrates it does not work when multiple goroutines are trying to mutate the value of the counter at the same time.

Write enough code to make it pass

A simple solution is to add a lock to our Counter, ensuring only one goroutine can increment the counter at a time. Go's [Mutex](#) provides such a lock:

A Mutex is a mutual exclusion lock. The zero value for a Mutex is an unlocked mutex.

```
type Counter struct {
    mu sync.Mutex
    value int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}
```

What this means is any goroutine calling Inc will acquire the lock on Counter if they are first. All the other goroutines will have to wait for it to be Unlocked before getting access.

If you now re-run the test it should now pass because each goroutine has to wait its turn before making a change.

I've seen other examples where the sync.Mutex is embedded into the struct.

You may see examples like this

```
type Counter struct {
    sync.Mutex
    value int
}
```

It can be argued that it can make the code a bit more elegant.

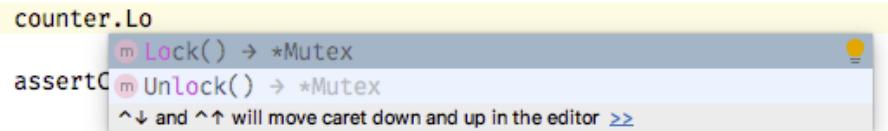
```
func (c *Counter) Inc() {
    c.Lock()
    defer c.Unlock()
```

```
    c.value++
}
```

This looks nice but while programming is a hugely subjective discipline, this is **bad and wrong**.

Sometimes people forget that embedding types means the methods of that type becomes part of the public interface; and you often will not want that. Remember that we should be very careful with our public APIs, the moment we make something public is the moment other code can couple themselves to it. We always want to avoid unnecessary coupling.

Exposing Lock and Unlock is at best confusing but at worst potentially very harmful to your software if callers of your type start calling these methods.



This seems like a really bad idea

Copying mutexes

Our test passes but our code is still a bit dangerous

If you run go vet on your code you should get an error like the following

```
sync/v2/sync_test.go:16: call of assertCounter copies lock value: v1.Counter contains sync.Mutex  
sync/v2/sync_test.go:39: assertCounter passes lock by value: v1.Counter contains sync.Mutex
```

A look at the documentation of [sync.Mutex](#) tells us why

A Mutex must not be copied after first use.

When we pass our Counter (by value) to assertCounter it will try and create a copy of the mutex.

To solve this we should pass in a pointer to our Counter instead, so change the signature of assertCounter

```
func assertCounter(t testing.TB, got *Counter, want int)
```

Our tests will no longer compile because we are trying to pass in a Counter rather than a *Counter. To solve this I prefer to create a constructor which shows readers of your API that it would be better to not initialise the type yourself.

```
func NewCounter() *Counter {
    return &Counter{}
}
```

Use this function in your tests when initialising Counter.

Wrapping up

We've covered a few things from the [sync package](#)

- Mutex allows us to add locks to our data
- WaitGroup is a means of waiting for goroutines to finish jobs

When to use locks over channels and goroutines?

We've previously covered goroutines in the first concurrency chapter which let us write safe concurrent code so why would you use locks? The go wiki has a page dedicated to this topic; [Mutex Or Channel](#)

A common Go newbie mistake is to over-use channels and goroutines just because it's possible, and/or because it's fun. Don't be afraid to use a sync.Mutex if that fits your problem best. Go is pragmatic in letting you use the tools that solve your problem best and not forcing you into one style of code.

Paraphrasing:

- **Use channels when passing ownership of data**
- **Use mutexes for managing state**

go vet

Remember to use go vet in your build scripts as it can alert you to some subtle bugs in your code before they hit your poor users.

Don't use embedding because it's convenient

- Think about the effect embedding has on your public API.
- Do you really want to expose these methods and have people coupling their own code to them?
- With respect to mutexes, this could be potentially disastrous in very unpredictable and weird ways, imagine some nefarious code unlocking a mutex when it shouldn't be; this would cause some very strange bugs that will be hard to track down.

Context

[You can find all the code for this chapter here](#)

Software often kicks off long-running, resource-intensive processes (often in goroutines). If the action that caused this gets cancelled or fails for some reason you need to stop these processes in a consistent way through your application.

If you don't manage this your snappy Go application that you're so proud of could start having difficult to debug performance problems.

In this chapter we'll use the package context to help us manage long-running processes.

We're going to start with a classic example of a web server that when hit kicks off a potentially long-running process to fetch some data for it to return in the response.

We will exercise a scenario where a user cancels the request before the data can be retrieved and we'll make sure the process is told to give up.

I've set up some code on the happy path to get us started. Here is our server code.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, store.Fetch())
    }
}
```

The function Server takes a Store and returns us a http.HandlerFunc. Store is defined as:

```
type Store interface {
    Fetch() string
}
```

The returned function calls the store's Fetch method to get the data and writes it to the response.

We have a corresponding stub for Store which we use in a test.

```
type StubStore struct {
    response string
}

func (s *StubStore) Fetch() string {
    return s.response
}
```

```
func TestServer(t *testing.T) {
    data := "hello, world"
    svr := Server(&StubStore{data})

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
    }
}
```

Now that we have a happy path, we want to make a more realistic scenario where the Store can't finish aFetch before the user cancels the request.

Write the test first

Our handler will need a way of telling the Store to cancel the work so update the interface.

```
type Store interface {
    Fetch() string
    Cancel()
}
```

We will need to adjust our spy so it takes some time to return data and a way of knowing it has been told to cancel. We'll also rename it to SpyStore as we are now observing the way it is called. It'll have to add Cancel as a method to implement the Store interface.

```
type SpyStore struct {
    response string
    cancelled bool
}

func (s *SpyStore) Fetch() string {
    time.Sleep(100 * time.Millisecond)
    return s.response
}

func (s *SpyStore) Cancel() {
    s.cancelled = true
}
```

Let's add a new test where we cancel the request before 100 milliseconds and check the store to see if it gets cancelled.

```
t.Run("tells store to cancel work if request is cancelled", func(t *testing.T) {
    data := "hello, world"
    store := &SpyStore{response: data}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)

    cancellingCtx, cancel := context.WithCancel(request.Context())
    time.AfterFunc(5*time.Millisecond, cancel)
    request = request.WithContext(cancellingCtx)

    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if !store.cancelled {
        t.Error("store was not told to cancel")
    }
})
```

From the [Go Blog: Context](#)

The context package provides functions to derive new Context values from existing ones. These values form a tree: when a Context is canceled, all Contexts derived from it are also canceled.

It's important that you derive your contexts so that cancellations are propagated throughout the call stack for a given request.

What we do is derive a new cancellingCtx from our request which returns us a cancel function. We then schedule that function to be called in 5 milliseconds by using time.AfterFunc. Finally we use this new context in our request by calling request.WithContext.

Try to run the test

The test fails as we'd expect.

```
--- FAIL: TestServer (0.00s)
--- FAIL: TestServer/tells_store_to_cancel_work_if_request_is_cancelled (0.00s)
    context_test.go:62: store was not told to cancel
```

Write enough code to make it pass

Remember to be disciplined with TDD. Write the minimal amount of code to make our test pass.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        store.Cancel()
        fmt.Fprint(w, store.Fetch())
    }
}
```

This makes this test pass but it doesn't feel good does it! We surely shouldn't be cancelling Store before we fetch on every request.

By being disciplined it highlighted a flaw in our tests, this is a good thing!

We'll need to update our happy path test to assert that it does not get cancelled.

```
t.Run("returns data from store", func(t *testing.T) {
    data := "hello, world"
    store := &SpyStore{response: data}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
    }

    if store.cancelled {
        t.Error("it should not have cancelled the store")
    }
})
```

Run both tests and the happy path test should now be failing and now we're forced to do a more sensible implementation.

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        data := make(chan string, 1)
```

```
go func() {
    data <- store.Fetch()
}()

select {
case d := <-data:
    fmt.Fprint(w, d)
case <-ctx.Done():
    store.Cancel()
}
}
```

What have we done here?

context has a method Done() which returns a channel which gets sent a signal when the context is "done" or "cancelled". We want to listen to that signal and call store.Cancel if we get it but we want to ignore it if our Store manages to Fetch before it.

To manage this we run Fetch in a goroutine and it will write the result into a new channel data. We then use select to effectively race to the two asynchronous processes and then we either write a response or Cancel.

Refactor

We can refactor our test code a bit by making assertion methods on our spy

```
type SpyStore struct {
    response string
    cancelled bool
    t        *testing.T
}

func (s *SpyStore) assertWasCancelled() {
    s.t.Helper()
    if !s.cancelled {
        s.t.Error("store was not told to cancel")
    }
}

func (s *SpyStore) assertWasNotCancelled() {
    s.t.Helper()
    if s.cancelled {
        s.t.Error("store was told to cancel")
    }
}
```

```
    }
}
```

Remember to pass in the `*testing.T` when creating the spy.

```
func TestServer(t *testing.T) {
    data := "hello, world"

    t.Run("returns data from store", func(t *testing.T) {
        store := &SpyStore{response: data, t: t}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)
        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if response.Body.String() != data {
            t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
        }

        store.assertWasNotCancelled()
    })

    t.Run("tells store to cancel work if request is cancelled", func(t *testing.T) {
        store := &SpyStore{response: data, t: t}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)

        cancellingCtx, cancel := context.WithCancel(request.Context())
        time.AfterFunc(5*time.Millisecond, cancel)
        request = request.WithContext(cancellingCtx)

        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        store.assertWasCancelled()
    })
})
```

This approach is ok, but is it idiomatic?

Does it make sense for our web server to be concerned with manually cancelling Store? What if Store also happens to depend on other slow-running processes? We'll have to make sure that `Store.Cancel` correctly propagates the cancellation to all of its dependants.

One of the main points of context is that it is a consistent way of offering cancellation.

[From the go doc](#)

Incoming requests to a server should create a Context, and outgoing calls to servers should accept a Context. The chain of function calls between them must propagate the Context, optionally replacing it with a derived Context created using WithCancel, WithDeadline, WithTimeout, or WithValue. When a Context is canceled, all Contexts derived from it are also canceled.

From the [Go Blog: Context](#) again:

At Google, we require that Go programmers pass a Context parameter as the first argument to every function on the call path between incoming and outgoing requests. This allows Go code developed by many different teams to interoperate well. It provides simple control over timeouts and cancelation and ensures that critical values like security credentials transit Go programs properly.

(Pause for a moment and think of the ramifications of every function having to send in a context, and the ergonomics of that.)

Feeling a bit uneasy? Good. Let's try and follow that approach though and instead pass through the context to our Store and let it be responsible. That way it can also pass the context through to its dependants and they too can be responsible for stopping themselves.

Write the test first

We'll have to change our existing tests as their responsibilities are changing. The only thing our handler is responsible for now is making sure it sends a context through to the downstream Store and that it handles the error that will come from the Store when it is cancelled.

Let's update our Store interface to show the new responsibilities.

```
type Store interface {
    Fetch(ctx context.Context) (string, error)
}
```

Delete the code inside our handler for now

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
    }
}
```

Update our SpyStore

```
type SpyStore struct {
    response string
    t        *testing.T
}

func (s *SpyStore) Fetch(ctx context.Context) (string, error) {
    data := make(chan string, 1)

    go func() {
        var result string
        for _, c := range s.response {
            select {
            case <-ctx.Done():
                log.Println("spy store got cancelled")
                return
            default:
                time.Sleep(10 * time.Millisecond)
                result += string(c)
            }
        }
        data <- result
    }()

    select {
    case <-ctx.Done():
        return "", ctx.Err()
    case res := <-data:
        return res, nil
    }
}
```

We have to make our spy act like a real method that works with context.

We are simulating a slow process where we build the result slowly by appending the string, character by character in a goroutine. When the goroutine finishes its work it writes the string to the data channel. The goroutine listens for the ctx.Done and will stop the work if a signal is sent in that channel.

Finally the code uses another select to wait for that goroutine to finish its work or for the cancellation to occur.

It's similar to our approach from before, we use Go's concurrency primitives to make two asynchronous processes race each other to determine what we return.

You'll take a similar approach when writing your own functions and methods that accept a context so make sure you understand what's going on.

Finally we can update our tests. Comment out our cancellation test so we can fix the happy path test first.

```
t.Run("returns data from store", func(t *testing.T) {
    data := "hello, world"
    store := &SpyStore{response: data, t: t}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)
    response := httptest.NewRecorder()

    svr.ServeHTTP(response, request)

    if response.Body.String() != data {
        t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
    }
})
```

Try to run the test

```
==== RUN TestServer/returns_data_from_store
--- FAIL: TestServer (0.00s)
    --- FAIL: TestServer/returns_data_from_store (0.00s)
        context_test.go:22: got "", want "hello, world"
```

Write enough code to make it pass

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        data, _ := store.Fetch(r.Context())
        fmt.Fprint(w, data)
    }
}
```

Our happy path should be... happy. Now we can fix the other test.

Write the test first

We need to test that we do not write any kind of response on the error case. Sadly `httptest.ResponseRecorder` doesn't have a way of figuring this out so we'll have to roll our own spy to test for this.

```
type SpyResponseWriter struct {
    written bool
}

func (s *SpyResponseWriter) Header() http.Header {
    s.written = true
    return nil
}

func (s *SpyResponseWriter) Write([]byte) (int, error) {
    s.written = true
    return 0, errors.New("not implemented")
}

func (s *SpyResponseWriter) WriteHeader(statusCode int) {
    s.written = true
}
```

Our SpyResponseWriter implements http.ResponseWriter so we can use it in the test.

```
t.Run("tells store to cancel work if request is cancelled", func(t *testing.T) {
    data := "hello, world"
    store := &SpyStore{response: data, t: t}
    svr := Server(store)

    request := httptest.NewRequest(http.MethodGet, "/", nil)

    cancellingCtx, cancel := context.WithCancel(context.Background())
    time.AfterFunc(5*time.Millisecond, cancel)
    request = request.WithContext(cancellingCtx)

    response := &SpyResponseWriter{}

    svr.ServeHTTP(response, request)

    if response.written {
        t.Error("a response should not have been written")
    }
})
```

Try to run the test

```
==== RUN TestServer
==== RUN TestServer/tells_store_to_cancel_work_if_request_is_cancelled
--- FAIL: TestServer (0.01s)
```

```
--- FAIL: TestServer/tells_store_to_cancel_work_if_request_is_cancelled (0.01s)
context_test.go:47: a response should not have been written
```

Write enough code to make it pass

```
func Server(store Store) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        data, err := store.Fetch(r.Context())

        if err != nil {
            return // todo: log error however you like
        }

        fmt.Fprint(w, data)
    }
}
```

We can see after this that the server code has become simplified as it's no longer explicitly responsible for cancellation, it simply passes through context and relies on the downstream functions to respect any cancellations that may occur.

Wrapping up

What we've covered

- How to test a HTTP handler that has had the request cancelled by the client.
- How to use context to manage cancellation.
- How to write a function that accepts context and uses it to cancel itself by using goroutines, select and channels.
- Follow Google's guidelines as to how to manage cancellation by propagating request scoped context through your call-stack.
- How to roll your own spy for http.ResponseWriter if you need it.

What about context.Value ?

Michal Štrba and I have a similar opinion.

If you use ctx.Value in my (non-existent) company, you're fired

Some engineers have advocated passing values through context as it feels convenient.

Convenience is often the cause of bad code.

The problem with `context.Values` is that it's just an untyped map so you have no type-safety and you have to handle it not actually containing your value. You have to create a coupling of map keys from one module to another and if someone changes something things start breaking.

In short, **if a function needs some values, put them as typed parameters rather than trying to fetch them from `context.Value`.** This makes it statically checked and documented for everyone to see.

But... On other hand, it can be helpful to include information that is orthogonal to a request in a context, such as a trace id. Potentially this information would not be needed by every function in your call-stack and would make your functional signatures very messy.

Jack Lindamood says **Context.Value should inform, not control**

The content of `context.Value` is for maintainers not users. It should never be required input for documented or expected results.

Additional material

- I really enjoyed reading [Context should go away for Go 2](#) by [Michal Štrba](#). His argument is that having to pass context everywhere is a smell, that it's pointing to a deficiency in the language in respect to cancellation. He says it would better if this was somehow solved at the language level, rather than at a library level. Until that happens, you will need context if you want to manage long running processes.
- The [Go blog further describes the motivation for working with context and has some examples](#)

Roman Numerals

You can find all the code for this chapter here

Some companies will ask you to do the [Roman Numeral Kata](#) as part of the interview process. This chapter will show how you can tackle it with TDD.

We are going to write a function which converts an [Arabic number](#) (numbers 0 to 9) to a Roman Numeral.

If you haven't heard of [Roman Numerals](#) they are how the Romans wrote down numbers.

You build them by sticking symbols together and those symbols represent numbers

So I is "one". III is three.

Seems easy but there's a few interesting rules. V means five, but IV is 4 (not IIII).

MCMLXXXIV is 1984. That looks complicated and it's hard to imagine how we can write code to figure this out right from the start.

As this book stresses, a key skill for software developers is to try and identify "thin vertical slices" of useful functionality and then **iterating**. The TDD workflow helps facilitate iterative development.

So rather than 1984, let's start with 1.

Write the test first

```
func TestRomanNumerals(t *testing.T) {
    got := ConvertToRoman(1)
    want := "|"

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

If you've got this far in the book this is hopefully feeling very boring and routine to you. That's a good thing.

Try to run the test

```
./numeral_test.go:6:9: undefined: ConvertToRoman
```

Let the compiler guide the way

Write the minimal amount of code for the test to run and check the failing test output

Create our function but don't make the test pass yet, always make sure the tests fails how you expect

```
func ConvertToRoman(arabic int) string {
    return ""
}
```

It should run now

```
==== RUN TestRomanNumerals
--- FAIL: TestRomanNumerals (0.00s)
    numeral_test.go:10: got "", want 'I'
FAIL
```

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {
    return "I"
}
```

Refactor

Not much to refactor yet.

I know it feels weird just to hard-code the result but with TDD we want to stay out of "red" for as long as possible. It may feel like we haven't accomplished much but we've defined our API and got a test capturing one of our rules; even if the "real" code is pretty dumb.

Now use that uneasy feeling to write a new test to force us to write slightly less dumb code.

Write the test first

We can use subtests to nicely group our tests

```
func TestRomanNumerals(t *testing.T) {
    t.Run("1 gets converted to I", func(t *testing.T) {
        got := ConvertToRoman(1)
        want := "I"

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

    t.Run("2 gets converted to II", func(t *testing.T) {
        got := ConvertToRoman(2)
        want := "II"

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

Try to run the test

```
==== RUN TestRomanNumerals/2_gets_converted_to_II
--- FAIL: TestRomanNumerals/2_gets_converted_to_II (0.00s)
    numeral_test.go:20: got 'I', want 'II'
```

Not much surprise there

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {
    if arabic == 2 {
        return "II"
    }
    return "I"
}
```

Yup, it still feels like we're not actually tackling the problem. So we need to write more tests to drive us forward.

Refactor

We have some repetition in our tests. When you're testing something which feels like it's a matter of "given input X, we expect Y" you should probably use table based tests.

```
func TestRomanNumerals(t *testing.T) {
    cases := []struct {
        Description string
        Arabic      int
        Want        string
    }{
        {"1 gets converted to I", 1, "I"},
        {"2 gets converted to II", 2, "II"},
    }

    for _, test := range cases {
        t.Run(test.Description, func(t *testing.T) {
            got := ConvertToRoman(test.Arabic)
            if got != test.Want {
                t.Errorf("got %q, want %q", got, test.Want)
            }
        })
    }
}
```

We can now easily add more cases without having to write any more test boilerplate.

Let's push on and go for 3

Write the test first

Add the following to our cases

```
{"3 gets converted to III", 3, "III"},
```

Try to run the test

```
==== RUN TestRomanNumerals/3_getsConvertedToIII
--- FAIL: TestRomanNumerals/3_getsConvertedToIII (0.00s)
    numeral_test.go:20: got 'I', want 'III'
```

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {
    if arabic == 3 {
        return "III"
    }
    if arabic == 2 {
        return "II"
    }
    return "I"
}
```

Refactor

OK so I'm starting to not enjoy these if statements and if you look at the code hard enough you can see that we're building a string of I based on the size of arabic.

We "know" that for more complicated numbers we will be doing some kind of arithmetic and string concatenation.

Let's try a refactor with these thoughts in mind, it might not be suitable for the end solution but that's OK. We can always throw our code away and start afresh with the tests we have to guide us.

```
func ConvertToRoman(arabic int) string {
    var result strings.Builder
    for i := 0; i < arabic; i++ {
```

```
        result.WriteString("I")
    }

return result.String()
}
```

You may not have used `strings.Builder` before

A Builder is used to efficiently build a string using Write methods. It minimizes memory copying.

Normally I wouldn't bother with such optimisations until I have an actual performance problem but the amount of code is not much larger than a "manual" appending on a string so we may as well use the faster approach.

The code looks better to me and describes the domain as we know it right now.

The Romans were into DRY too...

Things start getting more complicated now. The Romans in their wisdom thought repeating characters would become hard to read and count. So a rule with Roman Numerals is you can't have the same character repeated more than 3 times in a row.

Instead you take the next highest symbol and then "subtract" by putting a symbol to the left of it. Not all symbols can be used as subtractors; only I (1), X (10) and C (100).

For example 5 in Roman Numerals is V. To create 4 you do not do IIII, instead you do IV.

Write the test first

```
{"4 gets converted to IV (can't repeat more than 3 times)", 4, "IV"},
```

Try to run the test

```
==== RUN TestRomanNumerals/4_getsConvertedToIV(cant_repeat_more_than_3_times)
--- FAIL: TestRomanNumerals/4_getsConvertedToIV(cant_repeat_more_than_3_times) (0.000s)
    numeral_test.go:24: got 'IIII', want 'IV'
```

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {
```

```
if arabic == 4 {
    return "IV"
}

var result strings.Builder

for i := 0; i < arabic; i++ {
    result.WriteString("I")
}

return result.String()
}
```

Refactor

I don't "like" that we have broken our string building pattern and I want to carry on with it.

```
func ConvertToRoman(arabic int) string {

    var result strings.Builder

    for i := arabic; i > 0; i-- {
        if i == 4 {
            result.WriteString("IV")
            break
        }
        result.WriteString("I")
    }

    return result.String()
}
```

In order for 4 to "fit" with my current thinking I now count down from the Arabic number, adding symbols to our string as we progress. Not sure if this will work in the long run but let's see!

Let's make 5 work

Write the test first

```
{"5 gets converted to V", 5, "V"},
```

Try to run the test

```
==== RUN TestRomanNumerals/5_gets_converted_to_V
--- FAIL: TestRomanNumerals/5_gets_converted_to_V (0.00s)
    numeral_test.go:25: got 'IIV', want 'V'
```

Write enough code to make it pass

Just copy the approach we did for 4

```
func ConvertToRoman(arabic int) string {
    var result strings.Builder

    for i := arabic; i > 0; i-- {
        if i == 5 {
            result.WriteString("V")
            break
        }
        if i == 4 {
            result.WriteString("IV")
            break
        }
        result.WriteString("I")
    }

    return result.String()
}
```

Refactor

Repetition in loops like this are usually a sign of an abstraction waiting to be called out. Short-circuiting loops can be an effective tool for readability but it could also be telling you something else.

We are looping over our Arabic number and if we hit certain symbols we are calling break but what we are really doing is subtracting over i in a ham-fisted manner.

```
func ConvertToRoman(arabic int) string {
    var result strings.Builder

    for arabic > 0 {
        switch {
        case arabic > 4:
            result.WriteString("V")
```

```
arabic -= 5
case arabic > 3:
    result.WriteString("IV")
    arabic -= 4
default:
    result.WriteString("I")
    arabic--
}
}

return result.String()
}
```

- Given the signals I'm reading from our code, driven from our tests of some very basic scenarios I can see that to build a Roman Numeral I need to subtract from arabic as I apply symbols
- The for loop no longer relies on an i and instead we will keep building our string until we have subtracted enough symbols away from arabic.

I'm pretty sure this approach will be valid for 6 (VI), 7 (VII) and 8 (VIII) too. Nonetheless add the cases in to our test suite and check (I won't include the code for brevity, check the github for samples if you're unsure).

9 follows the same rule as 4 in that we should subtract I from the representation of the following number. 10 is represented in Roman Numerals with X; so therefore 9 should be IX.

Write the test first

```
{"9 gets converted to IX", 9, "IX"},
```

Try to run the test

```
==== RUN TestRomanNumerals/9_getsConvertedToIX
--- FAIL: TestRomanNumerals/9_getsConvertedToIX (0.00s)
    numeral_test.go:29: got 'VIV', want 'IX'
```

Write enough code to make it pass

We should be able to adopt the same approach as before

```
case arabic > 8:
    result.WriteString("IX")
    arabic -= 9
```

Refactor

It feels like the code is still telling us there's a refactor somewhere but it's not totally obvious to me, so let's keep going.

I'll skip the code for this too, but add to your test cases a test for 10 which should be X and make it pass before reading on.

Here are a few tests I added as I'm confident up to 39 our code should work

```
{"10 gets converted to X", 10, "X"},  
 {"14 gets converted to XIV", 14, "XIV"},  
 {"18 gets converted to XVIII", 18, "XVIII"},  
 {"20 gets converted to XX", 20, "XX"},  
 {"39 gets converted to XXXIX", 39, "XXXIX"},
```

If you've ever done OO programming, you'll know that you should view switch statements with a bit of suspicion. Usually you are capturing a concept or data inside some imperative code when in fact it could be captured in a class structure instead.

Go isn't strictly OO but that doesn't mean we ignore the lessons OO offers entirely (as much as some would like to tell you).

Our switch statement is describing some truths about Roman Numerals along with behaviour.

We can refactor this by decoupling the data from the behaviour.

```
type RomanNumeral struct {  
    Value int  
    Symbol string  
}  
  
var allRomanNumerals = []RomanNumeral{  
    {10, "X"},  
    {9, "IX"},  
    {5, "V"},  
    {4, "IV"},  
    {1, "I"},  
}  
  
func ConvertToRoman(arabic int) string {  
    var result strings.Builder  
  
    for _, numeral := range allRomanNumerals {  
        for arabic >= numeral.Value {  
            result.WriteString(numeral.Symbol)
```

```

        arabic -= numeral.Value
    }
}

return result.String()
}

```

This feels much better. We've declared some rules around the numerals as data rather than hidden in an algorithm and we can see how we just work through the Arabic number, trying to add symbols to our result if they fit.

Does this abstraction work for bigger numbers? Extend the test suite so it works for the Roman number for 50 which is L.

Here are some test cases, try and make them pass.

```
{"40 gets converted to XL", 40, "XL"},  
 {"47 gets converted to XLVII", 47, "XLVII"},  
 {"49 gets converted to XLIX", 49, "XLIX"},  
 {"50 gets converted to L", 50, "L"},
```

Need help? You can see what symbols to add in [this gist](#).

And the rest!

Here are the remaining symbols

Arabic	Roman
100	C
500	D
1000	M

Take the same approach for the remaining symbols, it should just be a matter of adding data to both the tests and our array of symbols.

Does your code work for 1984: MCMLXXXIV ?

Here is my final test suite

```
func TestRomanNumerals(t *testing.T) {  
    cases := []struct {  
        Arabic int  
        Roman string  
    }{  
        {Arabic: 1, Roman: "I"},  
        {Arabic: 2, Roman: "II"},  
        {Arabic: 3, Roman: "III"},  
    }
```

```
{Arabic: 4, Roman: "IV"},  
{Arabic: 5, Roman: "V"},  
{Arabic: 6, Roman: "VI"},  
{Arabic: 7, Roman: "VII"},  
{Arabic: 8, Roman: "VIII"},  
{Arabic: 9, Roman: "IX"},  
{Arabic: 10, Roman: "X"},  
{Arabic: 14, Roman: "XIV"},  
{Arabic: 18, Roman: "XVIII"},  
{Arabic: 20, Roman: "XX"},  
{Arabic: 39, Roman: "XXXIX"},  
{Arabic: 40, Roman: "XL"},  
{Arabic: 47, Roman: "XLVII"},  
{Arabic: 49, Roman: "XLIX"},  
{Arabic: 50, Roman: "L"},  
{Arabic: 100, Roman: "C"},  
{Arabic: 90, Roman: "XC"},  
{Arabic: 400, Roman: "CD"},  
{Arabic: 500, Roman: "D"},  
{Arabic: 900, Roman: "CM"},  
{Arabic: 1000, Roman: "M"},  
{Arabic: 1984, Roman: "MCMXXXIV"},  
{Arabic: 3999, Roman: "MMMCMXCIX"},  
{Arabic: 2014, Roman: "MMXIV"},  
{Arabic: 1006, Roman: "MVI"},  
{Arabic: 798, Roman: "DCCXCVIII"},  
}  
  
for _, test := range cases {  
    t.Run(fmt.Sprintf("%d gets converted to %q", test.Arabic, test.Roman), func(t *testing.T) {  
        got := ConvertToRoman(test.Arabic)  
        if got != test.Roman {  
            t.Errorf("got %q, want %q", got, test.Roman)  
        }  
    })  
}
```

- I removed description as I felt the data described enough of the information.
 - I added a few other edge cases I found just to give me a little more confidence. With table based tests this is very cheap to do.

I didn't change the algorithm, all I had to do was update the allRomanNumerals array.

```
var allRomanNumerals = []RomanNumeral{
```

```
{1000, "M"},  
{900, "CM"},  
{500, "D"},  
{400, "CD"},  
{100, "C"},  
{90, "XC"},  
{50, "L"},  
{40, "XL"},  
{10, "X"},  
{9, "IX"},  
{5, "V"},  
{4, "IV"},  
{1, "I"},  
}
```

Parsing Roman Numerals

We're not done yet. Next we're going to write a function that converts from a Roman Numeral to an int

Write the test first

We can re-use our test cases here with a little refactoring

Move the cases variable outside of the test as a package variable in a var block.

```
func TestConvertingToArabic(t *testing.T) {  
    for _, test := range cases[:1] {  
        t.Run(fmt.Sprintf("%q gets converted to %d", test.Roman, test.Arabic), func(t *testing.T) {  
            got := ConvertToArabic(test.Roman)  
            if got != test.Arabic {  
                t.Errorf("got %d, want %d", got, test.Arabic)  
            }  
        })  
    }  
}
```

Notice I am using the slice functionality to just run one of the tests for now (cases[:1]) as trying to make all of those tests pass all at once is too big a leap

Try to run the test

```
./numeral_test.go:60:11: undefined: ConvertToArabic
```

Write the minimal amount of code for the test to run and check the failing test output

Add our new function definition

```
func ConvertToArabic(roman string) int {  
    return 0  
}
```

The test should now run and fail

```
--- FAIL: TestConvertingToArabic (0.00s)  
    --- FAIL: TestConvertingToArabic/'I'_getsConvertedTo_1 (0.00s)  
        numeral_test.go:62: got 0, want 1
```

Write enough code to make it pass

You know what to do

```
func ConvertToArabic(roman string) int {  
    return 1  
}
```

Next, change the slice index in our test to move to the next test case (e.g. cases[:2]). Make it pass yourself with the dumbest code you can think of, continue writing dumb code (best book ever right?) for the third case too. Here's my dumb code.

```
func ConvertToArabic(roman string) int {  
    if roman == "III" {  
        return 3  
    }  
    if roman == "II" {  
        return 2  
    }  
    return 1  
}
```

Through the dumbness of real code that works we can start to see a pattern like before. We need to iterate through the input and build something, in this case a total.

```
func ConvertToArabic(roman string) int {  
    total := 0  
    for range roman {  
        total++  
    }  
    return total  
}
```

Write the test first

Next we move to cases[:4] (IV) which now fails because it gets 2 back as that's the length of the string.

Write enough code to make it pass

```
// earlier..  
type RomanNumerals []RomanNumeral  
  
func (r RomanNumerals) ValueOf(symbol string) int {  
    for _, s := range r {  
        if s.Symbol == symbol {  
            return s.Value  
        }  
    }  
  
    return 0  
}  
  
var allRomanNumerals = RomanNumerals{  
    {1000, "M"},  
    {900, "CM"},  
    {500, "D"},  
    {400, "CD"},  
    {100, "C"},  
    {90, "XC"},  
    {50, "L"},  
    {40, "XL"},  
    {10, "X"},  
    {9, "IX"},  
    {5, "V"},  
    {4, "IV"},  
    {1, "I"},  
}  
  
// later..  
func ConvertToArabic(roman string) int {  
    total := 0  
  
    for i := 0; i < len(roman); i++ {  
        symbol := roman[i]  
  
        // look ahead to next symbol if we can and, the current symbol is base 10 (only valid subtraction)  
        if i+1 < len(roman) && symbol == 'I' {  
            nextSymbol := roman[i+1]
```

```

// build the two character string
potentialNumber := string([]byte{symbol, nextSymbol})

// get the value of the two character string
value := allRomanNumerals.ValueOf(potentialNumber)

if value != 0 {
    total += value
    i++ // move past this character too for the next loop
} else {
    total++
}
} else {
    total++
}
}

return total
}

```

This is horrible but it does work. It's so bad I felt the need to add comments.

- I wanted to be able to look up an integer value for a given roman numeral so I made a type from our array of RomanNumerals and then added a method to it, ValueOf
- Next in our loop we need to look ahead if the string is big enough and the current symbol is a valid subtractor. At the moment it's just I (1) but can also be X (10) or C (100).
 - If it satisfies both of these conditions we need to lookup the value and add it to the total if it is one of the special subtractors, otherwise ignore it
 - Then we need to further increment i so we don't count this symbol twice

Refactor

I'm not entirely convinced this will be the long-term approach and there's potentially some interesting refactors we could do, but I'll resist that in case our approach is totally wrong. I'd rather make a few more tests pass first and see. For the meantime I made the first if statement slightly less horrible.

```

func ConvertToArabic(roman string) int {
    total := 0

    for i := 0; i < len(roman); i++ {

```

```
symbol := roman[i]

if couldBeSubtractive(i, symbol, roman) {
    nextSymbol := roman[i+1]

    // build the two character string
    potentialNumber := string([]byte{symbol, nextSymbol})

    // get the value of the two character string
    value := allRomanNumerals.ValueOf(potentialNumber)

    if value != 0 {
        total += value
        i++ // move past this character too for the next loop
    } else {
        total++
    }
} else {
    total++
}
}

return total
}

func couldBeSubtractive(index int, currentSymbol uint8, roman string) bool {
    return index+1 < len(roman) && currentSymbol == 'I'
}
```

Write the test first

Let's move on to cases[:5]

```
==== RUN TestConvertingToArabic/'V'_getsConvertedTo5
--- FAIL: TestConvertingToArabic/'V'_getsConvertedTo5 (0.00s)
    numeral_test.go:62: got 1, want 5
```

Write enough code to make it pass

Apart from when it is subtractive our code assumes that every character is a I which is why the value is 1. We should be able to re-use our ValueOf method to fix this.

```
func ConvertToArabic(roman string) int {
    total := 0

    for i := 0; i < len(roman); i++ {
```

```
symbol := roman[i]

// look ahead to next symbol if we can and, the current symbol is base 10 (only valid subtractive)
if couldBeSubtractive(i, symbol, roman) {
    nextSymbol := roman[i+1]

    // build the two character string
    potentialNumber := string([]byte{symbol, nextSymbol})

    if value := allRomanNumerals.ValueOf(potentialNumber); value != 0 {
        total += value
        i++ // move past this character too for the next loop
    } else {
        total++ // this is fishy...
    }
} else {
    total += allRomanNumerals.ValueOf(string([]byte{symbol}))
}
}

return total
}
```

Refactor

When you index strings in Go, you get a byte. This is why when we build up the string again we have to do stuff like `string([]byte{symbol})`. It's repeated a couple of times, let's just move that functionality so that `ValueOf` takes some bytes instead.

```
func (r RomanNumerals) ValueOf(symbols ...byte) int {
    symbol := string(symbols)
    for _, s := range r {
        if s.Symbol == symbol {
            return s.Value
        }
    }

    return 0
}
```

Then we can just pass in the bytes as is, to our function

```
func ConvertToArabic(roman string) int {
    total := 0

    for i := 0; i < len(roman); i++ {
        symbol := roman[i]
```

```

if couldBeSubtractive(i, symbol, roman) {
    if value := allRomanNumerals.ValueOf(symbol, roman[i+1]); value != 0 {
        total += value
        i++ // move past this character too for the next loop
    } else {
        total++ // this is fishy...
    }
} else {
    total += allRomanNumerals.ValueOf(symbol)
}
}
return total
}

```

It's still pretty nasty, but it's getting there.

If you start moving our cases[:xx] number through you'll see that quite a few are passing now. Remove the slice operator entirely and see which ones fail, here's some examples from my suite

```

==== RUN TestConvertingToArabic/'XL'_getsConvertedTo_40
--- FAIL: TestConvertingToArabic/'XL'_getsConvertedTo_40 (0.00s)
    numeral_test.go:62: got 60, want 40
==== RUN TestConvertingToArabic/'XLVII'_getsConvertedTo_47
--- FAIL: TestConvertingToArabic/'XLVII'_getsConvertedTo_47 (0.00s)
    numeral_test.go:62: got 67, want 47
==== RUN TestConvertingToArabic/'XLIX'_getsConvertedTo_49
--- FAIL: TestConvertingToArabic/'XLIX'_getsConvertedTo_49 (0.00s)
    numeral_test.go:62: got 69, want 49

```

I think all we're missing is an update to couldBeSubtractive so that it accounts for the other kinds of subtractive symbols

```

func couldBeSubtractive(index int, currentSymbol uint8, roman string) bool {
    isSubtractiveSymbol := currentSymbol == 'I' || currentSymbol == 'X' || currentSymbol == 'C'
    return index+1 < len(roman) && isSubtractiveSymbol
}

```

Try again, they still fail. However we left a comment earlier...

```
total++ // this is fishy...
```

We should never be just incrementing total as that implies every symbol is a I. Replace it with:

```
total += allRomanNumerals.ValueOf(symbol)
```

And all the tests pass! Now that we have fully working software we can indulge ourselves in some refactoring, with confidence.

Refactor

Here is all the code I finished up with. I had a few failed attempts but as I keep emphasising, that's fine and the tests help me play around with the code freely.

```
import "strings"

func ConvertToArabic(roman string) (total int) {
    for _, symbols := range windowedRoman(roman).Symbols() {
        total += allRomanNumerals.ValueOf(symbols...)
    }
    return
}

func ConvertToRoman(arabic int) string {
    var result strings.Builder

    for _, numeral := range allRomanNumerals {
        for arabic >= numeral.Value {
            result.WriteString(numeral.Symbol)
            arabic -= numeral.Value
        }
    }

    return result.String()
}

type romanNumeral struct {
    Value int
    Symbol string
}

type romanNumerals []romanNumeral

func (r romanNumerals) ValueOf(symbols ...byte) int {
    symbol := string(symbols)
    for _, s := range r {
        if s.Symbol == symbol {
            return s.Value
        }
    }

    return 0
}
```

```

func (r romanNumerals) Exists(symbols ...byte) bool {
    symbol := string(symbols)
    for _, s := range r {
        if s.Symbol == symbol {
            return true
        }
    }
    return false
}

var allRomanNumerals = romanNumerals{
    {1000, "M"},
    {900, "CM"},
    {500, "D"},
    {400, "CD"},
    {100, "C"},
    {90, "XC"},
    {50, "L"},
    {40, "XL"},
    {10, "X"},
    {9, "IX"},
    {5, "V"},
    {4, "IV"},
    {1, "I"},
}
}

type windowedRoman string

func (w windowedRoman) Symbols() (symbols [][]byte) {
    for i := 0; i < len(w); i++ {
        symbol := w[i]
        notAtEnd := i+1 < len(w)

        if notAtEnd && isSubtractive(symbol) && allRomanNumerals.Exists(symbol, w[i+1]) {
            symbols = append(symbols, []byte{symbol, w[i+1]})
            i++
        } else {
            symbols = append(symbols, []byte{symbol})
        }
    }
    return
}

func isSubtractive(symbol uint8) bool {
    return symbol == 'I' || symbol == 'X' || symbol == 'C'
}

```

My main problem with the previous code is similar to our refactor from earlier. We had too many concerns coupled together. We wrote an algorithm which was trying to extract Roman Numerals from a string and then find their values.

So I created a new type `windowedRoman` which took care of extracting the numerals, offering a `Symbols` method to retrieve them as a slice. This meant our `ConvertToArabic` function could simply iterate over the symbols and total them.

I broke the code down a bit by extracting some functions, especially around the wonky if statement to figure out if the symbol we are currently dealing with is a two character subtractive symbol.

There's probably a more elegant way but I'm not going to sweat it. The code is there and it works and it is tested. If I (or anyone else) finds a better way they can safely change it - the hard work is done.

An intro to property based tests

There have been a few rules in the domain of Roman Numerals that we have worked with in this chapter

- Can't have more than 3 consecutive symbols
- Only I (1), X (10) and C (100) can be "subtractors"
- Taking the result of `ConvertToRoman(N)` and passing it to `ConvertToArabic` should return us N

The tests we have written so far can be described as "example" based tests where we provide examples for the tooling to verify.

What if we could take these rules that we know about our domain and somehow exercise them against our code?

Property based tests help you do this by throwing random data at your code and verifying the rules you describe always hold true. A lot of people think property based tests are mainly about random data but they would be mistaken. The real challenge about property based tests is having a good understanding of your domain so you can write these properties.

Enough words, let's see some code

```
func TestPropertiesOfConversion(t *testing.T) {
    assertion := func(arabic int) bool {
        roman := ConvertToRoman(arabic)
        fromRoman := ConvertToArabic(roman)
        return fromRoman == arabic
    }
}
```

```
    if err := quick.Check(assertion, nil); err != nil {
        t.Error("failed checks", err)
    }
}
```

Rationale of property

Our first test will check that if we transform a number into Roman, when we use our other function to convert it back to a number that we get what we originally had.

- Given random number (e.g 4).
- Call ConvertToRoman with random number (should return IV if 4).
- Take the result of above and pass it to ConvertToArabic.
- The above should give us our original input (4).

This feels like a good test to build us confidence because it should break if there's a bug in either. The only way it could pass is if they have the same kind of bug; which isn't impossible but feels unlikely.

Technical explanation

We're using the [testing/quick](#) package from the standard library

Reading from the bottom, we provide quick.Check a function that it will run against a number of random inputs, if the function returns false it will be seen as failing the check.

Our assertion function above takes a random number and runs our functions to test the property.

Run our test

Try running it; your computer may hang for a while, so kill it when you're bored :)

What's going on? Try adding the following to the assertion code.

```
assertion := func(arabic int) bool {
    if arabic < 0 || arabic > 3999 {
        log.Println(arabic)
        return true
    }
    roman := ConvertToRoman(arabic)
    fromRoman := ConvertToArabic(roman)
    return fromRoman == arabic
}
```

You should see something like this:

```
==== RUN TestPropertiesOfConversion
2019/07/09 14:41:27 6849766357708982977
2019/07/09 14:41:27 -7028152357875163913
2019/07/09 14:41:27 -6752532134903680693
2019/07/09 14:41:27 4051793897228170080
2019/07/09 14:41:27 -1111868396280600429
2019/07/09 14:41:27 8851967058300421387
2019/07/09 14:41:27 562755830018219185
```

Just running this very simple property has exposed a flaw in our implementation. We used int as our input but:

- You can't do negative numbers with Roman Numerals
- Given our rule of a max of 3 consecutive symbols we can't represent a value greater than 3999 ([well, kinda](#)) and int has a much higher maximum value than 3999.

This is great! We've been forced to think more deeply about our domain which is a real strength of property based tests.

Clearly int is not a great type. What if we tried something a little more appropriate?

uint16

Go has types for unsigned integers, which means they cannot be negative; so that rules out one class of bug in our code immediately. By adding 16, it means it is a 16 bit integer which can store a max of 65535, which is still too big but gets us closer to what we need.

Try updating the code to use uint16 rather than int. I updated assertion in the test to give a bit more visibility.

```
assertion := func(arabic uint16) bool {
    if arabic > 3999 {
        return true
    }
    t.Log("testing", arabic)
    roman := ConvertToRoman(arabic)
    fromRoman := ConvertToArabic(roman)
    return fromRoman == arabic
}
```

Notice that now we are logging the input using the log method from the testing framework. Make sure you run the go test command with the flag -v to print the additional output (go test -v).

If you run the test they now actually run and you can see what is being tested. You can run multiple times to see our code stands up well to the various values! This gives me a lot of confidence that our code is working how we want.

The default number of runs quick.Check performs is 100 but you can change that with a config.

```
if err := quick.Check(assertion, &quick.Config{
    MaxCount: 1000,
}); err != nil {
    t.Error("failed checks", err)
}
```

Further work

- Can you write property tests that check the other properties we described?
- Can you think of a way of making it so it's impossible for someone to call our code with a number greater than 3999?
 - You could return an error
 - Or create a new type that cannot represent > 3999
 - * What do you think is best?

Wrapping up

More TDD practice with iterative development

Did the thought of writing code that converts 1984 into MCMLXXXIV feel intimidating to you at first? It did to me and I've been writing software for quite a long time.

The trick, as always, is to **get started with something simple** and take **small steps**.

At no point in this process did we make any large leaps, do any huge refactorings, or get in a mess.

I can hear someone cynically saying "this is just a kata". I can't argue with that, but I still take this same approach for every project I work on. I never ship a big distributed system in my first step, I find the simplest thing the team could ship (usually a "Hello world" website) and then iterate on small bits of functionality in manageable chunks, just like how we did here.

The skill is knowing how to split work up, and that comes with practice and with some lovely TDD to help you on your way.

Property based tests

- Built into the standard library
- If you can think of ways to describe your domain rules in code, they are an excellent tool for giving you more confidence
- Force you to think about your domain deeply
- Potentially a nice complement to your test suite

Postscript

This book is reliant on valuable feedback from the community. [Dave](#) is an enormous help in practically every chapter. But he had a real rant about my use of 'Arabic numerals' in this chapter so, in the interests of full disclosure, here's what he said.

Just going to write up why a value of type int isn't really an 'arabic numeral'. This might be me being way too precise so I'll completely understand if you tell me to f off.

A digit is a character used in the representation of numbers - from the Latin for 'finger', as we usually have ten of them. In the Arabic (also called Hindu-Arabic) number system there are ten of them. These Arabic digits are:

0 1 2 3 4 5 6 7 8 9

A numeral is the representation of a number using a collection of digits. An Arabic numeral is a number represented by Arabic digits in a base 10 positional number system. We say 'positional' because each digit has a different value based upon its position in the numeral. So

1337

The 1 has a value of one thousand because its the first digit in a four digit numeral.

Roman are built using a reduced number of digits (I, V etc...) mainly as values to produce the numeral. There's a bit of positional stuff but it's mostly I always representing 'one'.

So, given this, is int an 'Arabic number'? The idea of a number is not at all tied to its representation - we can see this if we ask ourselves what the correct representation of this number is:

255
1111111
two-hundred and fifty-five

Yes, this is a trick question. They're all correct. They're the representation of the same number in the decimal, binary, English, hexadecimal and octal number systems respectively.

The representation of a number as a numeral is independent of its properties as a number - and we can see this when we look at integer literals in Go:

`0xFF == 255 // true`

And how we can print integers in a format string:

```
n := 255
fmt.Printf("%b %c %d %o %q %x %X %U", n, n, n, n, n, n, n, n)
// 11111111   255 377  ' ff FF U+00FF
```

We can write the same integer both as a hexadecimal and an Arabic (decimal) numeral.

So when the function signature looks like `ConvertToRoman(arabic int)` string it's making a bit of an assumption about how it's being called. Because sometimes arabic will be written as a decimal integer literal

`ConvertToRoman(255)`

But it could just as well be written

`ConvertToRoman(0xFF)`

Really, we're not 'converting' from an Arabic numeral at all, we're 'printing' - representing - an int as a Roman numeral - and ints are not numerals, Arabic or otherwise; they're just numbers. The `ConvertToRoman` function is more like `strconv.Itoa` in that it's turning an int into a string.

But every other version of the kata doesn't care about this distinction so ☺

Mathematics

You can find all the code for this chapter here

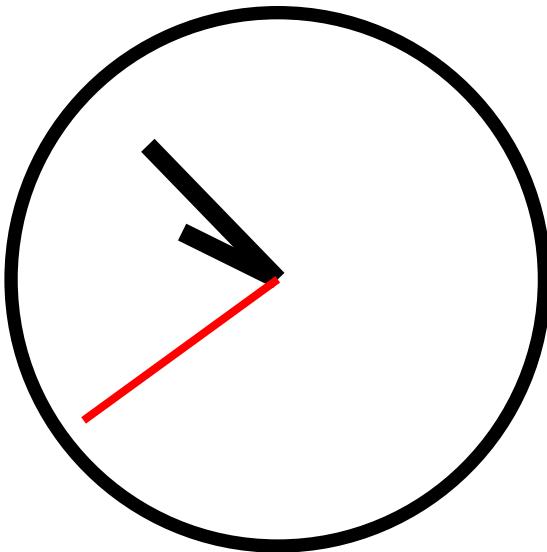
For all the power of modern computers to perform huge sums at lightning speed, the average developer rarely uses any mathematics to do their job. But not today! Today we'll use mathematics to solve a real

problem. And not boring mathematics - we're going to use trigonometry and vectors and all sorts of stuff that you always said you'd never have to use after highschool.

The Problem

You want to make an SVG of a clock. Not a digital clock - no, that would be easy - an analogue clock, with hands. You're not looking for anything fancy, just a nice function that takes a Time from the time package and spits out an SVG of a clock with all the hands - hour, minute and second - pointing in the right direction. How hard can that be?

First we're going to need an SVG of a clock for us to play with. SVGs are a fantastic image format to manipulate programmatically because they're written as a series of shapes, described in XML. So this clock:



is described like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/s
```

```
<svg xmlns="http://www.w3.org/2000/svg"
      width="100%"
      height="100%"
      viewBox="0 0 300 300"
      version="2.0">

    <!-- bezel -->
    <circle cx="150" cy="150" r="100" style="fill:#fff;stroke:#000;stroke-width:5px;" />

    <!-- hour hand -->
    <line x1="150" y1="150" x2="114.150000" y2="132.260000"
          style="fill:none;stroke:#000;stroke-width:7px;" />

    <!-- minute hand -->
    <line x1="150" y1="150" x2="101.290000" y2="99.730000"
          style="fill:none;stroke:#000;stroke-width:7px;" />

    <!-- second hand -->
    <line x1="150" y1="150" x2="77.190000" y2="202.900000"
          style="fill:none;stroke:#f00;stroke-width:3px;" />
</svg>
```

It's a circle with three lines, each of the lines starting in the middle of the circle ($x=150$, $y=150$), and ending some distance away.

So what we're going to do is reconstruct the above somehow, but change the lines so they point in the appropriate directions for a given time.

An Acceptance Test

Before we get too stuck in, lets think about an acceptance test.

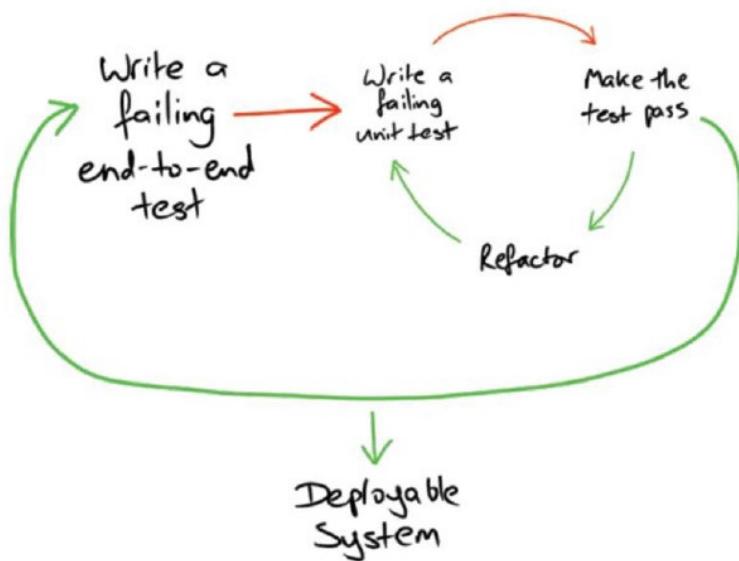
Wait, you don't know what an acceptance test is yet. Look, let me try to explain.

Let me ask you: what does winning look like? How do we know we've finished work? TDD provides a good way of knowing when you've finished: when the test passes. Sometimes it's nice - actually, almost all of the time it's nice - to write a test that tells you when you've finished writing the whole usable feature. Not just a test that tells you that a particular function is working in the way you expect, but a test that tells you that the whole thing you're trying to achieve - the 'feature' - is complete.

These tests are sometimes called 'acceptance tests', sometimes called 'feature tests'. The idea is that you write a really high level test to describe what you're trying to achieve - a user clicks a button

on a website, and they see a complete list of the Pokémons they've caught, for instance. When we've written that test, we can then write more tests - unit tests - that build towards a working system that will pass the acceptance test. So for our example these tests might be about rendering a webpage with a button, testing route handlers on a web server, performing database look ups, etc. All of these things will be TDD'd, and all of them will go towards making the original acceptance test pass.

Something like this classic picture by Nat Pryce and Steve Freeman



Anyway, let's try and write that acceptance test - the one that will let us know when we're done.

We've got an example clock, so let's think about what the important parameters are going to be.

```
<line x1="150" y1="150" x2="114.150000" y2="132.260000"  
style="fill:none;stroke:#000;stroke-width:7px;" />
```

The centre of the clock (the attributes `x1` and `y1` for this line) is the same for each hand of the clock. The numbers that need to change for each hand of the clock - the parameters to whatever builds the SVG - are the `x2` and `y2` attributes. We'll need an X and a Y for each of the hands of the clock.

I could think about more parameters - the radius of the clockface circle, the size of the SVG, the colours of the hands, their shape, etc... but it's better to start off by solving a simple, concrete problem with a simple, concrete solution, and then to start adding parameters to make it generalised.

So we'll say that

- every clock has a centre of (150, 150)
- the hour hand is 50 long
- the minute hand is 80 long
- the second hand is 90 long.

A thing to note about SVGs: the origin - point (0,0) - is at the top left hand corner, not the bottom left as we might expect. It'll be important to remember this when we're working out where what numbers to plug in to our lines.

Finally, I'm not deciding how to construct the SVG - we could use a template from the [text/template](#) package, or we could just send bytes into a bytes.Buffer or a writer. But we know we'll need those numbers, so let's focus on testing something that creates them.

Write the test first

So my first test looks like this:

```
package clockface_test

import (
    "projectpath/clockface"
    "testing"
    "time"
)

func TestSecondHandAtMidnight(t *testing.T) {
    tm := time.Date(1337, time.January, 1, 0, 0, 0, 0, time.UTC)

    want := clockface.Point{X: 150, Y: 150 - 90}
    got := clockface.SecondHand(tm)

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

Remember how SVGs plot their coordinates from the top left hand corner? To place the second hand at midnight we expect that it hasn't

moved from the centre of the clockface on the X axis - still 150 - and the Y axis is the length of the hand 'up' from the centre; 150 minus 90.

Try to run the test

This drives out the expected failures around the missing functions and types:

```
--- FAIL: TestSecondHandAtMidnight (0.00s)
./clockface_test.go:13:10: undefined: clockface.Point
./clockface_test.go:14:9: undefined: clockface.SecondHand
```

So a Point where the tip of the second hand should go, and a function to get it.

Write the minimal amount of code for the test to run and check the failing test output

Let's implement those types to get the code to compile

```
package clockface
```

```
import "time"
```

```
// A Point represents a two-dimensional Cartesian coordinate
type Point struct {
    X float64
    Y float64
}

// SecondHand is the unit vector of the second hand of an analogue clock at time `t`
// represented as a Point.
func SecondHand(t time.Time) Point {
    return Point{}
}
```

and now we get:

```
--- FAIL: TestSecondHandAtMidnight (0.00s)
    clockface_test.go:17: Got {0 0}, wanted {150 60}
FAIL
exit status 1
FAIL  learn-go-with-tests/math/clockface 0.006s
```

Write enough code to make it pass

When we get the expected failure, we can fill in the return value of SecondHand:

```
// SecondHand is the unit vector of the second hand of an analogue clock at time `t`
// represented as a Point.
func SecondHand(t time.Time) Point {
    return Point{150, 60}
}
```

Behold, a passing test.

```
PASS
ok      clockface 0.006s
```

Refactor

No need to refactor yet - there's barely enough code!

Repeat for new requirements

We probably need to do some work here that doesn't just involve returning a clock that shows midnight for every time...

Write the test first

```
func TestSecondHandAt30Seconds(t *testing.T) {
    tm := time.Date(1337, time.January, 1, 0, 0, 30, 0, time.UTC)

    want := clockface.Point{X: 150, Y: 150 + 90}
    got := clockface.SecondHand(tm)

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

Same idea, but now the second hand is pointing downwards so we add the length to the Y axis.

This will compile... but how do we make it pass?

Thinking time

How are we going to solve this problem?

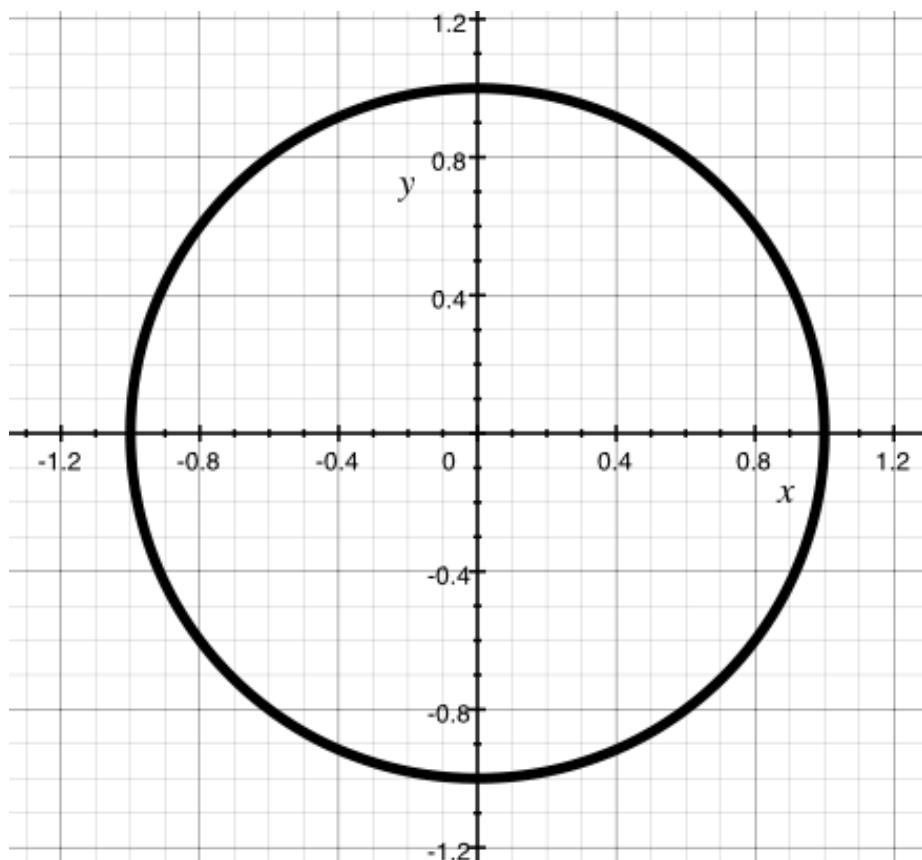
Every minute the second hand goes through the same 60 states, pointing in 60 different directions. When it's 0 seconds it points to the top of the clockface, when it's 30 seconds it points to the bottom of the clockface. Easy enough.

So if I wanted to think about in what direction the second hand was pointing at, say, 37 seconds, I'd want the angle between 12 o'clock and 37/60ths around the circle. In degrees this is $(360 / 60) * 37 = 222$, but it's easier just to remember that it's 37/60 of a complete rotation.

But the angle is only half the story; we need to know the X and Y coordinate that the tip of the second hand is pointing at. How can we work that out?

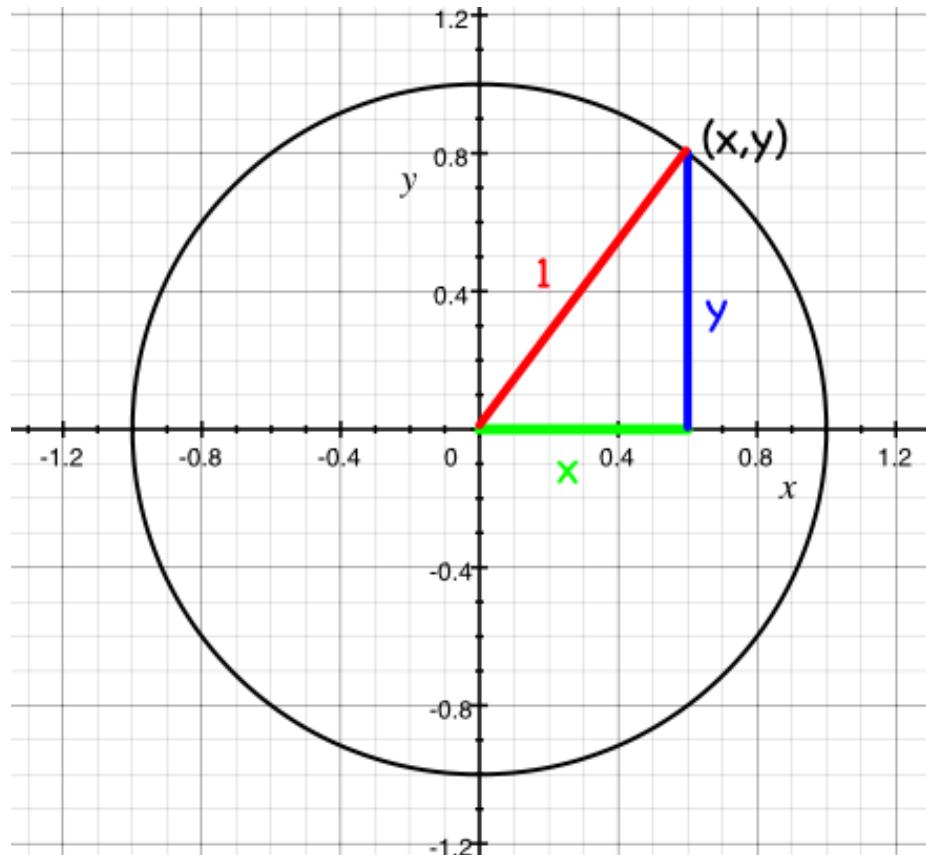
Math

Imagine a circle with a radius of 1 drawn around the origin - the coordinate 0, 0.

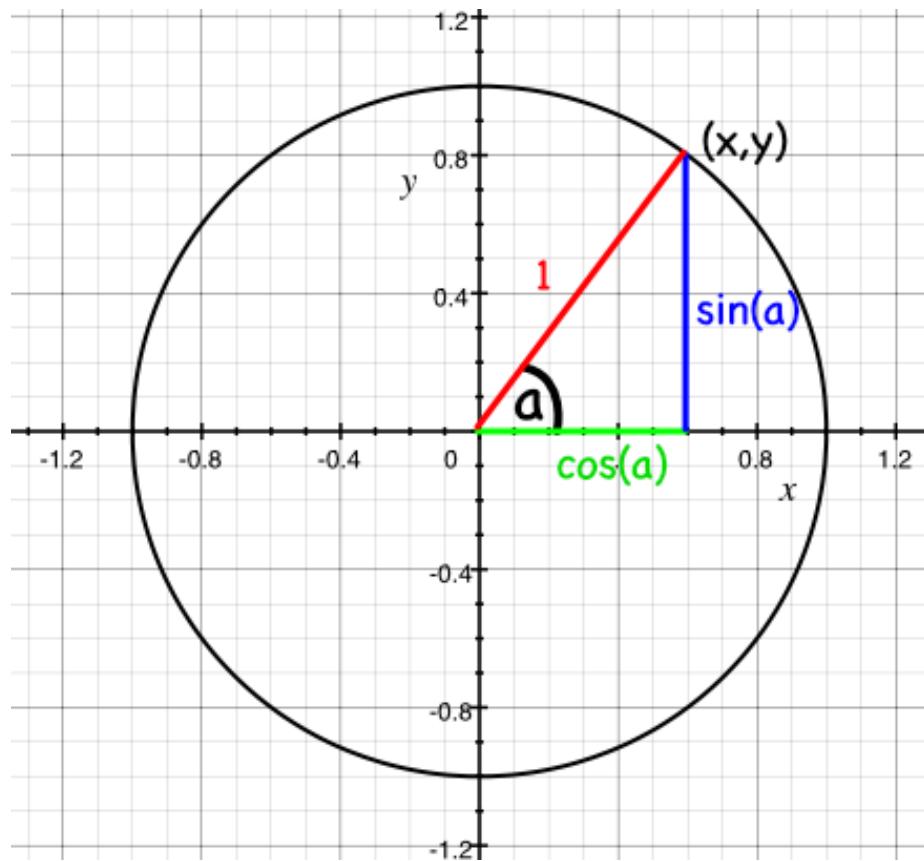


This is called the 'unit circle' because... well, the radius is 1 unit!

The circumference of the circle is made of points on the grid - more coordinates. The x and y components of each of these coordinates form a triangle, the hypotenuse of which is always 1 (i.e. the radius of the circle).

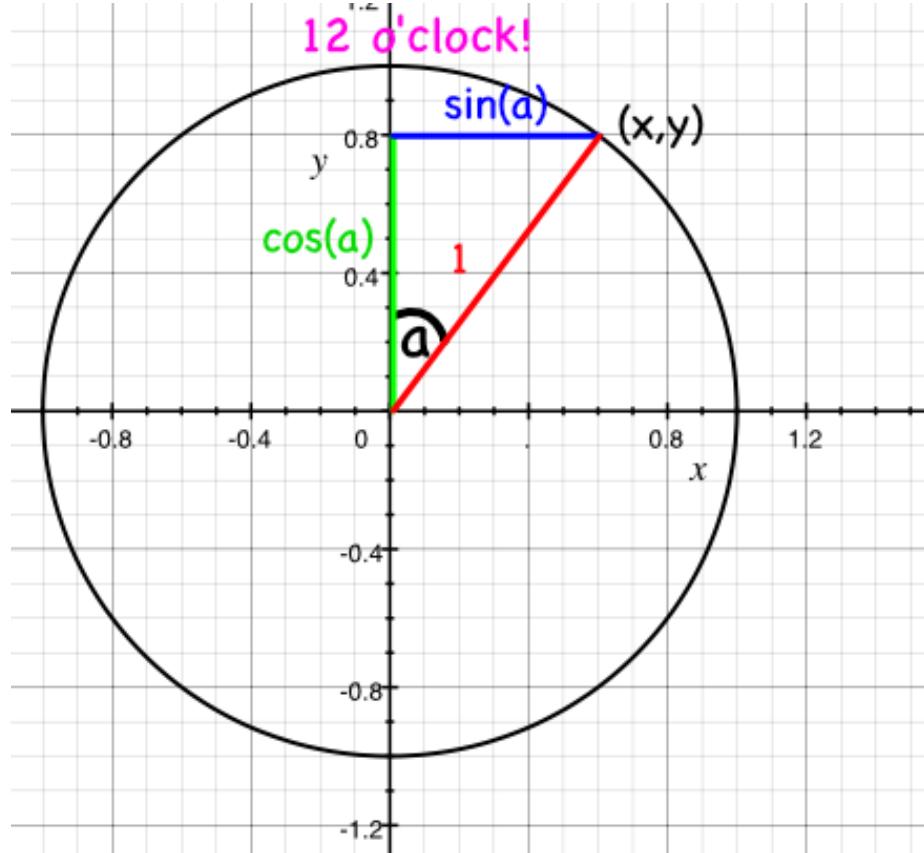


Now, trigonometry will let us work out the lengths of X and Y for each triangle if we know the angle they make with the origin. The X coordinate will be $\cos(a)$, and the Y coordinate will be $\sin(a)$, where a is the angle made between the line and the (positive) x axis.



(If you don't believe this, [go and look at Wikipedia...](#))

One final twist - because we want to measure the angle from 12 o'clock rather than from the X axis (3 o'clock), we need to swap the axis around; now $x = \sin(a)$ and $y = \cos(a)$.



So now we know how to get the angle of the second hand (1/60th of a circle for each second) and the X and Y coordinates. We'll need functions for both sin and cos.

math

Happily the Go math package has both, with one small snag we'll need to get our heads around; if we look at the description of `math.Cos`:

`Cos` returns the cosine of the radian argument `x`.

It wants the angle to be in radians. So what's a radian? Instead of defining the full turn of a circle to be made up of 360 degrees, we define a full turn as being 2π radians. There are good reasons to do this that we won't go into.¹

¹In short it makes it easier to do calculus with circles as π just keeps coming up as an angle if you use normal degrees, so if you count your angles in π s it makes all the equations simpler.

Now that we've done some reading, some learning and some thinking, we can write our next test.

Write the test first

All this maths is hard and confusing. I'm not confident I understand what's going on - so let's write a test! We don't need to solve the whole problem in one go - let's start off with working out the correct angle, in radians, for the second hand at a particular time.

I'm going to comment out the acceptance test that I was working on while I'm working on these tests - I don't want to get distracted by that test while I'm getting this one to pass.

A recap on packages

At the moment, our acceptance tests are in the `clockface_test` package. Our tests can be outside of the `clockface` package - as long as their name ends with `_test.go` they can be run.

I'm going to write these radians tests within the `clockface` package; they may never get exported, and they may get deleted (or moved) once I have a better grip on what's going on. I'll rename my acceptance test file to `clockface_acceptance_test.go`, so that I can create a new file called `clockface_test` to test seconds in radians.

```
package clockface
```

```
import (
    "math"
    "testing"
    "time"
)

func TestSecondsInRadians(t *testing.T) {
    thirtySeconds := time.Date(312, time.October, 28, 0, 0, 30, 0, time.UTC)
    want := math.Pi
    got := secondsInRadians(thirtySeconds)

    if want != got {
        t.Fatalf("Wanted %v radians, but got %v", want, got)
    }
}
```

Here we're testing that 30 seconds past the minute should put the second hand at halfway around the clock. And it's our first use of the `math` package! If a full turn of a circle is 2π radians, we know that

halfway round should just be π radians. `math.Pi` provides us with a value for π .

Try to run the test

```
./clockface_test.go:12:9: undefined: secondsInRadians
```

Write the minimal amount of code for the test to run and check the failing test output

```
func secondsInRadians(t time.Time) float64 {
    return 0
}
```

```
clockface_test.go:15: Wanted 3.141592653589793 radians, but got 0
```

Write enough code to make it pass

```
func secondsInRadians(t time.Time) float64 {
    return math.Pi
}
PASS
ok    clockface  0.011s
```

Refactor

Nothing needs refactoring yet

Repeat for new requirements

Now we can extend the test to cover a few more scenarios. I'm going to skip forward a bit and show some already refactored test code - it should be clear enough how I got where I want to.

```
func TestSecondsInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(0, 0, 30), math.Pi},
        {simpleTime(0, 0, 0), 0},
        {simpleTime(0, 0, 45), (math.Pi / 2) * 3},
        {simpleTime(0, 0, 7), (math.Pi / 30) * 7},
    }
}
```

```
for _, c := range cases {
    t.Run(testName(c.time), func(t *testing.T) {
        got := secondsInRadians(c.time)
        if got != c.angle {
            t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
        }
    })
}
```

I added a couple of helper functions to make writing this table based test a little less tedious. `testName` converts a time into a digital watch format (HH:MM:SS), and `simpleTime` constructs a `time.Time` using only the parts we actually care about (again, hours, minutes and seconds).² Here they are:

```
func simpleTime(hours, minutes, seconds int) time.Time {
    return time.Date(312, time.October, 28, hours, minutes, seconds, 0, time.UTC)
}

func testName(t time.Time) string {
    return t.Format("15:04:05")
}
```

These two functions should help make these tests (and future tests) a little easier to write and maintain.

This gives us some nice test output:

```
clockface_test.go:24: Wanted 0 radians, but got 3.141592653589793
```

```
clockface_test.go:24: Wanted 4.71238898038469 radians, but got 3.141592653589793
```

Time to implement all of that maths stuff we were talking about above:

```
func secondsInRadians(t time.Time) float64 {
    return float64(t.Second()) * (math.Pi / 30)
}
```

One second is $(2\pi / 60)$ radians... cancel out the 2 and we get $\pi/30$ radians. Multiply that by the number of seconds (as a `float64`) and we should now have all the tests passing...

```
clockface_test.go:24: Wanted 3.141592653589793 radians, but got 3.1415926535897936
```

Wait, what?

²This is a lot easier than writing a name out by hand as a string and then having to keep it in sync with the actual time. Believe me you don't want to do that...

Floats are horrible

Floating point arithmetic is [notoriously inaccurate](#). Computers can only really handle integers, and rational numbers to some extent. Decimal numbers start to become inaccurate, especially when we factor them up and down as we are in the secondsInRadians function. By dividing math.Pi by 30 and then by multiplying it by 30 we've ended up with a number that's no longer the same as math.Pi.

There are two ways around this:

1. Live with it
2. Refactor our function by refactoring our equation

Now (1) may not seem all that appealing, but it's often the only way to make floating point equality work. Being inaccurate by some infinitesimal fraction is frankly not going to matter for the purposes of drawing a clockface, so we could write a function that defines a 'close enough' equality for our angles. But there's a simple way we can get the accuracy back: we rearrange the equation so that we're no longer dividing down and then multiplying up. We can do it all by just dividing.

So instead of

`numberOfSeconds * π / 30`

we can write

`π / (30 / numberOfSeconds)`

which is equivalent.

In Go:

```
func secondsInRadians(t time.Time) float64 {  
    return (math.Pi / (30 / (float64(t.Second()))))  
}
```

And we get a pass.

```
PASS  
ok    clockface    0.005s
```

It should all look [something like this](#).

A note on dividing by zero

Computers often don't like dividing by zero because infinity is a bit strange.

In Go if you try to explicitly divide by zero you will get a compilation error.

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(10.0 / 0.0) // fails to compile
}
```

Obviously the compiler can't always predict that you'll divide by zero, such as our t.Second()

Try this

```
func main() {
    fmt.Println(10.0 / zero())
}

func zero() float64 {
    return 0.0
}
```

It will print +Inf (infinity). Dividing by +Inf seems to result in zero and we can see this with the following:

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(secondsinradians())
}

func zero() float64 {
    return 0.0
}

func secondsinradians() float64 {
    return (math.Pi / (30 / (float64(zero())))))
}
```

Repeat for new requirements

So we've got the first part covered here - we know what angle the second hand will be pointing at in radians. Now we need to work out the coordinates.

Again, let's keep this as simple as possible and only work with the unit circle; the circle with a radius of 1. This means that our hands will all have a length of one but, on the bright side, it means that the maths will be easy for us to swallow.

Write the test first

```
func TestSecondHandPoint(t *testing.T) {
    cases := []struct {
        time time.Time
        point Point
    }{
        {simpleTime(0, 0, 30), Point{0, -1}},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := secondHandPoint(c.time)
            if got != c.point {
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)
            }
        })
    }
}
```

Try to run the test

```
./clockface_test.go:40:11: undefined: secondHandPoint
```

Write the minimal amount of code for the test to run and check the failing test output

```
func secondHandPoint(t time.Time) Point {
    return Point{}
}
```

```
clockface_test.go:42: Wanted {0 -1} Point, but got {0 0}
```

Write enough code to make it pass

```
func secondHandPoint(t time.Time) Point {  
    return Point{0, -1}  
}  
  
PASS  
ok    clockface 0.007s
```

Repeat for new requirements

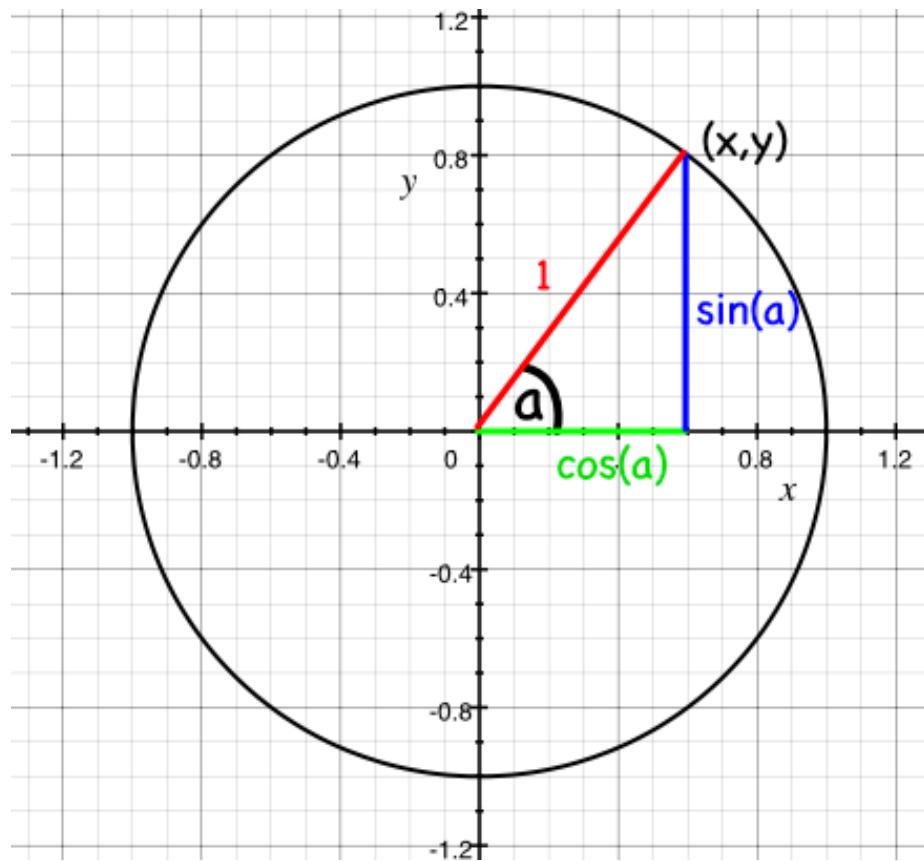
```
func TestSecondHandPoint(t *testing.T) {  
    cases := []struct {  
        time time.Time  
        point Point  
    }{  
        {simpleTime(0, 0, 30), Point{0, -1}},  
        {simpleTime(0, 0, 45), Point{-1, 0}},  
    }  
  
    for _, c := range cases {  
        t.Run(testName(c.time), func(t *testing.T) {  
            got := secondHandPoint(c.time)  
            if got != c.point {  
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)  
            }  
        })  
    }  
}
```

Try to run the test

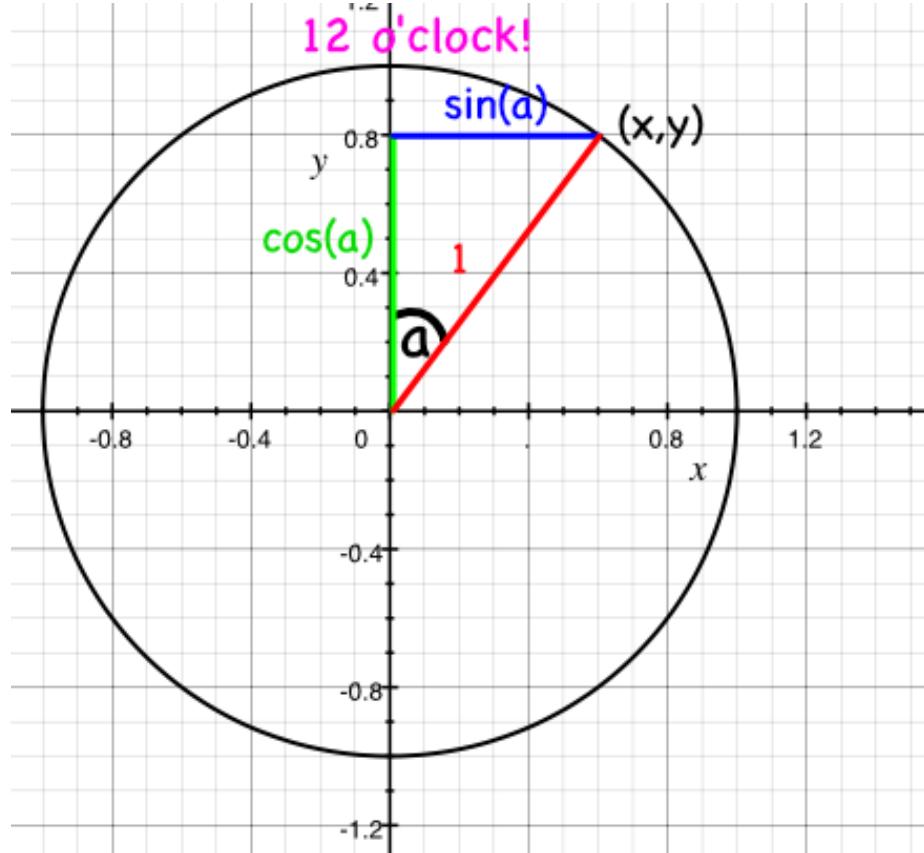
```
clockface_test.go:43: Wanted {-1 0} Point, but got {0 -1}
```

Write enough code to make it pass

Remember our unit circle picture?



Also recall that we want to measure the angle from 12 o'clock which is the Y axis instead of from the X axis which we would like measuring the angle between the second hand and 3 o'clock.



We now want the equation that produces X and Y. Let's write it into seconds:

```
func secondHandPoint(t time.Time) Point {
    angle := secondsInRadians(t)
    x := math.Sin(angle)
    y := math.Cos(angle)

    return Point{x, y}
}
```

Now we get

clockface_test.go:43: Wanted {0 -1} Point, but got {1.2246467991473515e-16 -1}

clockface_test.go:43: Wanted {-1 0} Point, but got {-1 -1.8369701987210272e-16}

Wait, what (again)? Looks like we've been cursed by the floats once more - both of those unexpected numbers are infinitesimal - way down at the 16th decimal place. So again we can either choose to try to

increase precision, or to just say that they're roughly equal and get on with our lives.

One option to increase the accuracy of these angles would be to use the rational type Rat from the math/big package. But given the objective is to draw an SVG and not land on the moon, I think we can live with a bit of fuzziness.

```
func TestSecondHandPoint(t *testing.T) {
    cases := []struct {
        time time.Time
        point Point
    }{
        {simpleTime(0, 0, 30), Point{0, -1}},
        {simpleTime(0, 0, 45), Point{-1, 0}},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := secondHandPoint(c.time)
            if !roughlyEqualPoint(got, c.point) {
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)
            }
        })
    }
}

func roughlyEqualFloat64(a, b float64) bool {
    const equalityThreshold = 1e-7
    return math.Abs(a-b) < equalityThreshold
}

func roughlyEqualPoint(a, b Point) bool {
    return roughlyEqualFloat64(a.X, b.X) &&
        roughlyEqualFloat64(a.Y, b.Y)
}
```

We've defined two functions to define approximate equality between two Points - they'll work if the X and Y elements are within 0.0000001 of each other. That's still pretty accurate.

And now we get:

```
PASS
ok    clockface 0.007s
```

Refactor

I'm still pretty happy with this.

Here's [what it looks like now](#)

Repeat for new requirements

Well, saying new isn't entirely accurate - really what we can do now is get that acceptance test passing! Let's remind ourselves of what it looks like:

```
func TestSecondHandAt30Seconds(t *testing.T) {
    tm := time.Date(1337, time.January, 1, 0, 0, 30, 0, time.UTC)

    want := clockface.Point{X: 150, Y: 150 + 90}
    got := clockface.SecondHand(tm)

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

Try to run the test

```
clockface_acceptance_test.go:28: Got {150 60}, wanted {150 240}
```

Write enough code to make it pass

We need to do three things to convert our unit vector into a point on the SVG:

1. Scale it to the length of the hand
2. Flip it over the X axis to account for the SVG having an origin in the top left hand corner
3. Translate it to the right position (so that it's coming from an origin of (150,150))

Fun times!

```
// SecondHand is the unit vector of the second hand of an analogue clock at time `t`
// represented as a Point.
func SecondHand(t time.Time) Point {
    p := secondHandPoint(t)
    p = Point{p.X * 90, p.Y * 90} // scale
    p = Point{p.X, -p.Y}         // flip
    p = Point{p.X + 150, p.Y + 150} // translate
```

```
    return p
}
```

Scale, flip, and translate in exactly that order. Hooray maths!

PASS

ok clockface 0.007s

Refactor

There's a few magic numbers here that should get pulled out as constants, so let's do that

```
const secondHandLength = 90
const clockCentreX = 150
const clockCentreY = 150
```

```
// SecondHand is the unit vector of the second hand of an analogue clock at time `t`
// represented as a Point.
func SecondHand(t time.Time) Point {
    p := secondHandPoint(t)
    p = Point{p.X * secondHandLength, p.Y * secondHandLength}
    p = Point{p.X, -p.Y}
    p = Point{p.X + clockCentreX, p.Y + clockCentreY} //translate
    return p
}
```

Draw the clock

Well... the second hand anyway...

Let's do this thing - because there's nothing worse than not delivering some value when it's just sitting there waiting to get out into the world to dazzle people. Let's draw a second hand!

We're going to stick a new directory under our main clockface package directory, called (confusingly), clockface. In there we'll put the main package that will create the binary that will build an SVG:

```
-- clockface
|   |-- main.go
|-- clockface.go
|-- clockface_acceptance_test.go
|-- clockface_test.go
```

Inside main.go, you'll start with this code but change the import for the clockface package to point at your own version:

```
package main

import (
    "fmt"
    "io"
    "os"
    "time"

    "learn-go-with-tests/math/clockface" // REPLACE THIS!
)

func main() {
    t := time.Now()
    sh := clockface.SecondHand(t)
    io.WriteString(os.Stdout, svgStart)
    io.WriteString(os.Stdout, bezel)
    io.WriteString(os.Stdout, secondHandTag(sh))
    io.WriteString(os.Stdout, svgEnd)
}

func secondHandTag(p clockface.Point) string {
    return fmt.Sprintf(`<line x1="150" y1="150" x2="%f" y2="%f" style="fill:none;stroke:#f00`+
        `>`)
}

const svgStart = `<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/s
<svg xmlns="http://www.w3.org/2000/svg"
      width="100%"
      height="100%"
      viewBox="0 0 300 300"
      version="2.0">` 

const bezel = `<circle cx="150" cy="150" r="100" style="fill:#fff;stroke:#000;stroke-width:5px` 

const svgEnd = `</svg>` 

Oh boy am I not trying to win any prizes for beautiful code with this
mess - but it does the job. It's writing an SVG out to os.Stdout - one
string at a time.

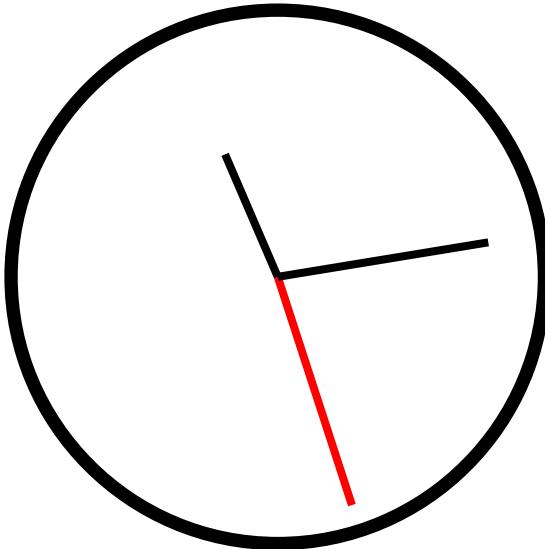
If we build this

go build

and run it, sending the output into a file

./clockface > clock.svg

We should see something like
```



And this is [how the code looks](#).

Refactor

This stinks. Well, it doesn't quite stink stink, but I'm not happy about it.

1. That whole SecondHand function is super tied to being an SVG... without mentioning SVGs or actually producing an SVG...
2. ... while at the same time I'm not testing any of my SVG code.

Yeah, I guess I screwed up. This feels wrong. Let's try to recover with a more SVG-centric test.

What are our options? Well, we could try testing that the characters spewing out of the SVGWriter contain things that look like the sort of SVG tag we're expecting for a particular time. For instance:

```
func TestSVGWriterAtMidnight(t *testing.T) {
    tm := time.Date(1337, time.January, 1, 0, 0, 0, time.UTC)

    var b strings.Builder
```

```
clockface.SVGWriter(&b, tm)
got := b.String()

want := `<line x1="150" y1="150" x2="150" y2="60"`

if !strings.Contains(got, want) {
    t.Errorf("Expected to find the second hand %v, in the SVG output %v", want, got)
}
```

But is this really an improvement?

Not only will it still pass if I don't produce a valid SVG (as it's only testing that a string appears in the output), but it will also fail if I make the smallest, unimportant change to that string - if I add an extra space between the attributes, for instance.

The biggest smell is that I'm testing a data structure - XML - by looking at its representation as a series of characters - as a string. This is never, ever a good idea as it produces problems just like the ones I outline above: a test that's both too fragile and not sensitive enough. A test that's testing the wrong thing!

So the only solution is to test the output as XML. And to do that we'll need to parse it.

Parsing XML

[encoding/xml](#) is the Go package that can handle all things to do with simple XML parsing.

The function [xml.Unmarshal](#) takes a []byte of XML data, and a pointer to a struct for it to get unmarshalled in to.

So we'll need a struct to unmarshall our XML into. We could spend some time working out what the correct names for all of the nodes and attributes, and how to write the correct structure but, happily, someone has written [zek](#) a program that will automate all of that hard work for us. Even better, there's an online version at <https://xml-to-go.github.io/>. Just paste the SVG from the top of the file into one box and - bam - out pops:

```
type Svg struct {
    XMLName xml.Name `xml:"svg"`
    Text    string   `xml:",chardata"`
    XmlNs  string   `xml:"xmlns,attr"`
    Width   string   `xml:"width,attr"`
    Height  string   `xml:"height,attr"`
    ViewBox string   `xml:"viewBox,attr`
```

```
Version string `xml:"version,attr"`
Circle struct {
    Text string `xml:",chardata"`
    Cx string `xml:"cx,attr"`
    Cy string `xml:"cy,attr"`
    R string `xml:"r,attr"`
    Style string `xml:"style,attr"`
} `xml:"circle"`
Line []struct {
    Text string `xml:",chardata"`
    X1 string `xml:"x1,attr"`
    Y1 string `xml:"y1,attr"`
    X2 string `xml:"x2,attr"`
    Y2 string `xml:"y2,attr"`
    Style string `xml:"style,attr"`
} `xml:"line"`
}
```

We could make adjustments to this if we needed to (like changing the name of the struct to SVG) but it's definitely good enough to start us off. Paste the struct into the `clockface_acceptance_test` file and let's write a test with it:

```
func TestSVGWriterAtMidnight(t *testing.T) {
    tm := time.Date(1337, time.January, 1, 0, 0, 0, 0, time.UTC)

    b := bytes.Buffer{}
    clockface.SVGWriter(&b, tm)

    svg := Svg{}
    xml.Unmarshal(b.Bytes(), &svg)

    x2 := "150"
    y2 := "60"

    for _, line := range svg.Line {
        if line.X2 == x2 && line.Y2 == y2 {
            return
        }
    }

    t.Errorf("Expected to find the second hand with x2 of %v and y2 of %v, in the SVG output")
}
```

We write the output of `clockface.SVGWriter` to a `bytes.Buffer` and then Unmarshal it into an `Svg`. We then look at each `Line` in the `Svg` to see if any of them have the expected `X2` and `Y2` values. If we get a match

we return early (passing the test); if not we fail with a (hopefully) informative message.

```
./clockface_acceptance_test.go:41:2: undefined: clockface.SVGWriter  
Looks like we'd better create SVGWriter.go...
```

```
package clockface
```

```
import (  
    "fmt"  
    "io"  
    "time"  
)  
  
const (  
    secondHandLength = 90  
    clockCentreX    = 150  
    clockCentreY    = 150  
)  
  
// SVGWriter writes an SVG representation of an analogue clock, showing the time t, to the writer w.  
func SVGWriter(w io.Writer, t time.Time) {  
    io.WriteString(w, svgStart)  
    io.WriteString(w, bezel)  
    secondHand(w, t)  
    io.WriteString(w, svgEnd)  
}  
  
func secondHand(w io.Writer, t time.Time) {  
    p := secondHandPoint(t)  
    p = Point{p.X * secondHandLength, p.Y * secondHandLength} // scale  
    p = Point{p.X, -p.Y} // flip  
    p = Point{p.X + clockCentreX, p.Y + clockCentreY} // translate  
    fmt.Fprintf(w, `<line x1="150" y1="150" x2="%f" y2="%f" style="fill:none;stroke:#f00;stroke-width:1px;">`)  
}  
  
const svgStart = `<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">  
<svg xmlns="http://www.w3.org/2000/svg"  
      width="100%"  
      height="100%"  
      viewBox="0 0 300 300"  
      version="2.0">`  
  
const bezel = `<circle cx="150" cy="150" r="100" style="fill:#fff;stroke:#000;stroke-width:5px;">`
```

```
const svgEnd = `</svg>`
```

The most beautiful SVG writer? No. But hopefully it'll do the job...

```
clockface_acceptance_test.go:56: Expected to find the second hand with x2 of 150 and y2 of 60
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"
<svg xmlns="http://www.w3.org/2000/svg"
      width="100%"
      height="100%"
      viewBox="0 0 300 300"
      version="2.0"><circle cx="150" cy="150" r="100" style="fill:#fff;stroke:#000;stroke-width:1px">
```

Oooops! The %f format directive is printing our coordinates to the default level of precision - six decimal places. We should be explicit as to what level of precision we're expecting for the coordinates. Let's say three decimal places.

```
fmt.Fprintf(w, `<line x1="150" y1="150" x2=".3f" y2=".3f" style="fill:none;stroke:#f00;stroke-width:1px">
```

And after we update our expectations in the test

```
x2 := "150.000"
y2 := "60.000"
```

We get:

```
PASS
ok    clockface 0.006s
```

We can now shorten our main function:

```
package main

import (
    "os"
    "time"

    "learn-go-with-tests/math/clockface"
)

func main() {
    t := time.Now()
    clockface.SVGWriter(os.Stdout, t)
}
```

This is what [things should look like now](#).

And we can write a test for another time following the same pattern, but not before...

Refactor

Three things stick out:

1. We're not really testing for all of the information we need to ensure is present - what about the x1 values, for instance?
2. Also, those attributes for x1 etc. aren't really strings are they? They're numbers!
3. Do I really care about the style of the hand? Or, for that matter, the empty Text node that's been generated by zak?

We can do better. Let's make a few adjustments to the Svg struct, and the tests, to sharpen everything up.

```
type SVG struct {
    XMLName xml.Name `xml:"svg"`
    Xmlns string `xml:"xmlns,attr"`
    Width string `xml:"width,attr"`
    Height string `xml:"height,attr"`
    viewBox string `xml:"viewBox,attr"`
    Version string `xml:"version,attr"`
    Circle Circle `xml:"circle"`
    Line []Line `xml:"line"`
}

type Circle struct {
    Cx float64 `xml:"cx,attr"`
    Cy float64 `xml:"cy,attr"`
    R float64 `xml:"r,attr"`
}

type Line struct {
    X1 float64 `xml:"x1,attr"`
    Y1 float64 `xml:"y1,attr"`
    X2 float64 `xml:"x2,attr"`
    Y2 float64 `xml:"y2,attr"`
}
```

Here I've

- Made the important parts of the struct named types -- the Line and the Circle
- Turned the numeric attributes into float64s instead of strings.
- Deleted unused attributes like Style and Text
- Renamed Svg to SVG because it's the right thing to do.

This will let us assert more precisely on the line we're looking for:

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

```
tm := time.Date(1337, time.January, 1, 0, 0, 0, 0, time.UTC)
b := bytes.Buffer{}

clockface.SVGWriter(&b, tm)

svg := SVG{}

xml.Unmarshal(b.Bytes(), &svg)

want := Line{150, 150, 150, 60}

for _, line := range svg.Line {
    if line == want {
        return
    }
}

t.Errorf("Expected to find the second hand line %+v, in the SVG lines %+v", want, svg.Line)
```

Finally we can take a leaf out of the unit tests' tables, and we can write a helper function containsLine(line Line, lines []Line) bool to really make these tests shine:

```
func TestSVGWriterSecondHand(t *testing.T) {
    cases := []struct {
        time time.Time
        line Line
    }{
        {
            simpleTime(0, 0, 0),
            Line{150, 150, 150, 60},
        },
        {
            simpleTime(0, 0, 30),
            Line{150, 150, 150, 240},
        },
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            b := bytes.Buffer{}
            clockface.SVGWriter(&b, c.time)

            svg := SVG{}
            xml.Unmarshal(b.Bytes(), &svg)
```

```
        if !containsLine(c.line, svg.Line) {
            t.Errorf("Expected to find the second hand line %+v, in the SVG lines %+v", c.line, sv
        }
    }
}

func containsLine(l Line, ls []Line) bool {
    for _, line := range ls {
        if line == l {
            return true
        }
    }
    return false
}
```

Here's what [it looks like](#)

Now that's what I call an acceptance test!

Write the test first

So that's the second hand done. Now let's get started on the minute hand.

```
func TestSVGWriterMinuteHand(t *testing.T) {
    cases := []struct {
        time time.Time
        line Line
    }{
        {
            simpleTime(0, 0, 0),
            Line{150, 150, 150, 70},
        },
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            b := bytes.Buffer{}
            clockface.SVGWriter(&b, c.time)

            svg := SVG{}
            xml.Unmarshal(b.Bytes(), &svg)

            if !containsLine(c.line, svg.Line) {
                t.Errorf("Expected to find the minute hand line %+v, in the SVG lines %+v", c.line, sv
            }
        })
    }
}
```

```
        })
    }
}
```

Try to run the test

clockface_acceptance_test.go:87: Expected to find the minute hand line {X1:150 Y1:150 X2:150}

We'd better start building some other clock hands. Much in the same way as we produced the tests for the second hand, we can iterate to produce the following set of tests. Again we'll comment out our acceptance test while we get this working:

```
func TestMinutesInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(0, 30, 0), math.Pi},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := minutesInRadians(c.time)
            if got != c.angle {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}
```

Try to run the test

./clockface_test.go:59:11: undefined: minutesInRadians

Write the minimal amount of code for the test to run and check the failing test output

```
func minutesInRadians(t time.Time) float64 {
    return math.Pi
}
```

Repeat for new requirements

Well, OK - now let's make ourselves do some real work. We could model the minute hand as only moving every full minute - so that it

'jumps' from 30 to 31 minutes past without moving in between. But that would look a bit rubbish. What we want it to do is move a tiny little bit every second.

```
func TestMinutesInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(0, 30, 0), math.Pi},
        {simpleTime(0, 0, 7), 7 * (math.Pi / (30 * 60))},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := minutesInRadians(c.time)
            if got != c.angle {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}
```

How much is that tiny little bit? Well...

- Sixty seconds in a minute
- thirty minutes in a half turn of the circle (math.Pi radians)
- so $30 * 60$ seconds in a half turn.
- So if the time is 7 seconds past the hour ...
- ... we're expecting to see the minute hand at $7 * (\text{math.Pi} / (30 * 60))$ radians past the 12.

Try to run the test

```
clockface_test.go:62: Wanted 0.012217304763960306 radians, but got 3.141592653589793
```

Write enough code to make it pass

In the immortal words of Jennifer Aniston: [Here comes the science bit](#)

```
func minutesInRadians(t time.Time) float64 {
    return (secondsInRadians(t) / 60) +
        (math.Pi / (30 / float64(t.Minute())))
}
```

Rather than working out how far to push the minute hand around the clockface for every second from scratch, here we can just leverage

the secondsInRadians function. For every second the minute hand will move 1/60th of the angle the second hand moves.

```
secondsInRadians(t) / 60
```

Then we just add on the movement for the minutes - similar to the movement of the second hand.

```
math.Pi / (30 / float64(t.Minute()))
```

And...

```
PASS  
ok    clockface 0.007s
```

Nice and easy. This is what things [look like now](#)

Repeat for new requirements

Should I add more cases to the minutesInRadians test? At the moment there are only two. How many cases do I need before I move on to the testing the minuteHandPoint function?

One of my favourite TDD quotes, often attributed to Kent Beck,³ is
Write tests until fear is transformed into boredom.

And, frankly, I'm bored of testing that function. I'm confident I know how it works. So it's on to the next one.

Write the test first

```
func TestMinuteHandPoint(t *testing.T) {  
    cases := []struct {  
        time time.Time  
        point Point  
    }{  
        {simpleTime(0, 30, 0), Point{0, -1}},  
    }  
  
    for _, c := range cases {  
        t.Run(testName(c.time), func(t *testing.T) {  
            got := minuteHandPoint(c.time)  
            if !roughlyEqualPoint(got, c.point) {  
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)  
            }  
        })  
    }  
}
```

³Misattributed because, like all great authors, Kent Beck is more quoted than read. Beck himself attributes it to [Phlip](#).

```
    }
}
```

Try to run the test

```
./clockface_test.go:79:11: undefined: minuteHandPoint
```

Write the minimal amount of code for the test to run and check the failing test output

```
func minuteHandPoint(t time.Time) Point {
    return Point{}
}
```

```
clockface_test.go:80: Wanted {0 -1} Point, but got {0 0}
```

Write enough code to make it pass

```
func minuteHandPoint(t time.Time) Point {
    return Point{0, -1}
}
PASS
ok   clockface 0.007s
```

Repeat for new requirements

And now for some actual work

```
func TestMinuteHandPoint(t *testing.T) {
    cases := []struct {
        time time.Time
        point Point
    }{
        {simpleTime(0, 30, 0), Point{0, -1}},
        {simpleTime(0, 45, 0), Point{-1, 0}},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := minuteHandPoint(c.time)
            if !roughlyEqualPoint(got, c.point) {
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)
            }
        })
    }
}
```

```
clockface_test.go:81: Wanted {-1 0} Point, but got {0 -1}
```

Write enough code to make it pass

A quick copy and paste of the secondHandPoint function with some minor changes ought to do it...

```
func minuteHandPoint(t time.Time) Point {
    angle := minutesInRadians(t)
    x := math.Sin(angle)
    y := math.Cos(angle)

    return Point{x, y}
}

PASS
ok    clockface 0.009s
```

Refactor

We've definitely got a bit of repetition in the minuteHandPoint and secondHandPoint - I know because we just copied and pasted one to make the other. Let's DRY it out with a function.

```
func angleToPoint(angle float64) Point {
    x := math.Sin(angle)
    y := math.Cos(angle)

    return Point{x, y}
}
```

and we can rewrite minuteHandPoint and secondHandPoint as one liners:

```
func minuteHandPoint(t time.Time) Point {
    return angleToPoint(minutesInRadians(t))
}

func secondHandPoint(t time.Time) Point {
    return angleToPoint(secondsInRadians(t))
}

PASS
ok    clockface 0.007s
```

Now we can uncomment the acceptance test and get to work drawing the minute hand.

Write enough code to make it pass

The minuteHand function is a copy-and-paste of secondHand with some minor adjustments, such as declaring a minuteHandLength:

```
const minuteHandLength = 80
```

```
//...
```

```
func minuteHand(w io.Writer, t time.Time) {
    p := minuteHandPoint(t)
    p = Point{p.X * minuteHandLength, p.Y * minuteHandLength}
    p = Point{p.X, -p.Y}
    p = Point{p.X + clockCentreX, p.Y + clockCentreY}
    fmt.Fprintf(w, `<line x1="150" y1="150" x2="%f" y2="%f" style="fill:none;stroke:#000;stroke-width:1px">` , p.X, p.Y)
}
```

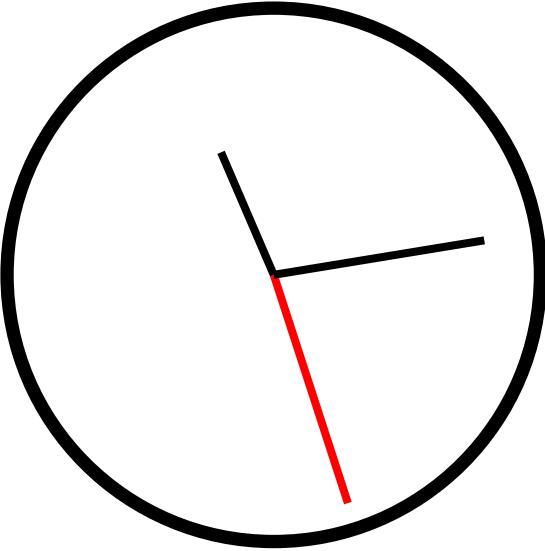
And a call to it in our SVGWriter function:

```
func SVGWriter(w io.Writer, t time.Time) {
    io.WriteString(w, svgStart)
    io.WriteString(w, bezel)
    secondHand(w, t)
    minuteHand(w, t)
    io.WriteString(w, svgEnd)
}
```

Now we should see that TestSVGWriterMinuteHand passes:

```
PASS
ok    clockface 0.006s
```

But the proof of the pudding is in the eating - if we now compile and run our clockface program, we should see something like



Refactor

Let's remove the duplication from the `secondHand` and `minuteHand` functions, putting all of that scale, flip and translate logic all in one place.

```
func secondHand(w io.Writer, t time.Time) {
    p := makeHand(secondHandPoint(t), secondHandLength)
    fmt.Fprintf(w, `<line x1="150" y1="150" x2="% .3f" y2="% .3f" style="fill:none;stroke:#f00;stroke-width:2px">`)
}

func minuteHand(w io.Writer, t time.Time) {
    p := makeHand(minuteHandPoint(t), minuteHandLength)
    fmt.Fprintf(w, `<line x1="150" y1="150" x2="% .3f" y2="% .3f" style="fill:none;stroke:#000;stroke-width:2px">`)
}

func makeHand(p Point, length float64) Point {
    p = Point{p.X * length, p.Y * length}
    p = Point{p.X, -p.Y}
    return Point{p.X + clockCentreX, p.Y + clockCentreY}
```

```
}

PASS
ok    clockface 0.007s

This is where we're up to now.

There... now it's just the hour hand to do!
```

Write the test first

```
func TestSVGWriterHourHand(t *testing.T) {
    cases := []struct {
        time time.Time
        line Line
    }{
        {
            simpleTime(6, 0, 0),
            Line{150, 150, 150, 200},
        },
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            b := bytes.Buffer{}
            clockface.SVGWriter(&b, c.time)

            svg := SVG{}
            xml.Unmarshal(b.Bytes(), &svg)

            if !containsLine(c.line, svg.Line) {
                t.Errorf("Expected to find the hour hand line %v, in the SVG lines %v", c.line, svg.Line)
            }
        })
    }
}
```

Try to run the test

```
clockface_acceptance_test.go:113: Expected to find the hour hand line {X1:150 Y1:150 X2:150 Y2:150}
```

Again, let's comment this one out until we've got the some coverage with the lower level tests:

Write the test first

```
func TestHoursInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(6, 0, 0), math.Pi},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := hoursInRadians(c.time)
            if got != c.angle {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}
```

Try to run the test

```
./clockface_test.go:97:11: undefined: hoursInRadians
```

Write the minimal amount of code for the test to run and check the failing test output

```
func hoursInRadians(t time.Time) float64 {
    return math.Pi
}

PASS
ok    clockface 0.007s
```

Repeat for new requirements

```
func TestHoursInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(6, 0, 0), math.Pi},
        {simpleTime(0, 0, 0), 0},
    }

    for _, c := range cases {
```

```
t.Run(testName(c.time), func(t *testing.T) {
    got := hoursInRadians(c.time)
    if got != c.angle {
        t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
    }
})
}
}
```

Try to run the test

```
clockface_test.go:100: Wanted 0 radians, but got 3.141592653589793
```

Write enough code to make it pass

```
func hoursInRadians(t time.Time) float64 {
    return (math.Pi / (6 / float64(t.Hour())))
}
```

Repeat for new requirements

```
func TestHoursInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(6, 0, 0), math.Pi},
        {simpleTime(0, 0, 0), 0},
        {simpleTime(21, 0, 0), math.Pi * 1.5},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := hoursInRadians(c.time)
            if got != c.angle {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}
```

Try to run the test

```
clockface_test.go:101: Wanted 4.71238898038469 radians, but got 10.995574287564276
```

Write enough code to make it pass

```
func hoursInRadians(t time.Time) float64 {
    return (math.Pi / (6 / (float64(t.Hour() % 12))))
}
```

Remember, this is not a 24-hour clock; we have to use the remainder operator to get the remainder of the current hour divided by 12.

```
PASS
ok    learn-go-with-tests/math/clockface 0.008s
```

Write the test first

Now let's try to move the hour hand around the clockface based on the minutes and the seconds that have passed.

```
func TestHoursInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(6, 0, 0), math.Pi},
        {simpleTime(0, 0, 0), 0},
        {simpleTime(21, 0, 0), math.Pi * 1.5},
        {simpleTime(0, 1, 30), math.Pi / ((6 * 60 * 60) / 90)},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := hoursInRadians(c.time)
            if got != c.angle {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}
```

Try to run the test

```
clockface_test.go:102: Wanted 0.013089969389957472 radians, but got 0
```

Write enough code to make it pass

Again, a bit of thinking is now required. We need to move the hour hand along a little bit for both the minutes and the seconds. Luckily

we have an angle already to hand for the minutes and the seconds - the one returned by minutesInRadians. We can reuse it!

So the only question is by what factor to reduce the size of that angle. One full turn is one hour for the minute hand, but for the hour hand it's twelve hours. So we just divide the angle returned by minutesInRadians by twelve:

```
func hoursInRadians(t time.Time) float64 {
    return (minutesInRadians(t) / 12) +
        (math.Pi / (6 / float64(t.Hour()%12)))
}
```

and behold:

```
clockface_test.go:104: Wanted 0.013089969389957472 radians, but got 0.0130899693899574
```

Floating point arithmetic strikes again.

Let's update our test to use roughlyEqualFloat64 for the comparison of the angles.

```
func TestHoursInRadians(t *testing.T) {
    cases := []struct {
        time time.Time
        angle float64
    }{
        {simpleTime(6, 0, 0), math.Pi},
        {simpleTime(0, 0, 0), 0},
        {simpleTime(21, 0, 0), math.Pi * 1.5},
        {simpleTime(0, 1, 30), math.Pi / ((6 * 60 * 60) / 90)},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := hoursInRadians(c.time)
            if !roughlyEqualFloat64(got, c.angle) {
                t.Fatalf("Wanted %v radians, but got %v", c.angle, got)
            }
        })
    }
}

PASS
ok    clockface 0.007s
```

Refactor

If we're going to use `roughlyEqualFloat64` in one of our radians tests, we should probably use it for all of them. That's a nice and simple refactor, which will leave things [looking like this](#).

Hour Hand Point

Right, it's time to calculate where the hour hand point is going to go by working out the unit vector.

Write the test first

```
func TestHourHandPoint(t *testing.T) {
    cases := []struct {
        time time.Time
        point Point
    }{
        {simpleTime(6, 0, 0), Point{0, -1}},
        {simpleTime(21, 0, 0), Point{-1, 0}},
    }

    for _, c := range cases {
        t.Run(testName(c.time), func(t *testing.T) {
            got := hourHandPoint(c.time)
            if !roughlyEqualPoint(got, c.point) {
                t.Fatalf("Wanted %v Point, but got %v", c.point, got)
            }
        })
    }
}
```

Wait, am I going to write two test cases at once? Isn't this bad TDD?

On TDD Zealotry

Test driven development is not a religion. Some people might act like it is - usually people who don't do TDD but are happy to moan on Twitter or Dev.to that it's only done by zealots and that they're 'being pragmatic' when they don't write tests. But it's not a religion. It's a tool.

I know what the two tests are going to be - I've tested two other clock hands in exactly the same way - and I already know what my implementation is going to be - I wrote a function for the general case of changing an angle into a point in the minute hand iteration.

I'm not going to plough through TDD ceremony for the sake of it. TDD is a technique that helps me understand the code I'm writing - and the code that I'm going to write - better. TDD gives me feedback, knowledge and insight. But if I've already got that knowledge, then I'm not going to plough through the ceremony for no reason. Neither tests nor TDD are an end in themselves.

My confidence has increased, so I feel I can make larger strides forward. I'm going to 'skip' a few steps, because I know where I am, I know where I'm going and I've been down this road before.

But also note: I'm not skipping writing the tests entirely - I'm still writing them first. They're just appearing in less granular chunks.

Try to run the test

```
./clockface_test.go:119:11: undefined: hourHandPoint
```

Write enough code to make it pass

```
func hourHandPoint(t time.Time) Point {
    return angleToPoint(hoursInRadians(t))
}
```

As I said, I know where I am, and I know where I'm going. Why pretend otherwise? The tests will soon tell me if I'm wrong.

```
PASS
ok    learn-go-with-tests/math/clockface 0.009s
```

Draw the hour hand

And finally we get to draw in the hour hand. We can bring in that acceptance test by uncommenting it:

```
func TestSVGWriterHourHand(t *testing.T) {
    cases := []struct {
        time time.Time
        line Line
    }{
        {
            simpleTime(6, 0, 0),
            Line{150, 150, 150, 200},
        },
    }
    for _, c := range cases {
```

```
t.Run(testName(c.time), func(t *testing.T) {
    b := bytes.Buffer{}
    clockface.SVGWriter(&b, c.time)

    svg := SVG{}
    xml.Unmarshal(b.Bytes(), &svg)

    if !containsLine(c.line, svg.Line) {
        t.Errorf("Expected to find the hour hand line %v, in the SVG lines %v", c.line, svg.Lines)
    }
}
})
```

Try to run the test

```
clockface_acceptance_test.go:113: Expected to find the hour hand line {X1:150 Y1:150 X2:150
in the SVG lines [{X1:150 Y1:150 X2:150 Y2:60} {X1:150 Y1:150 X2:150 Y2:70}]
```

Write enough code to make it pass

And we can now make our final adjustments to the SVG writing constants and functions:

```
const (
    secondHandLength = 90
    minuteHandLength = 80
    hourHandLength   = 50
    clockCentreX     = 150
    clockCentreY     = 150
)

// SVGWriter writes an SVG representation of an analogue clock, showing the time t, to the writer w
func SVGWriter(w io.Writer, t time.Time) {
    io.WriteString(w, svgStart)
    io.WriteString(w, bezel)
    secondHand(w, t)
    minuteHand(w, t)
    hourHand(w, t)
    io.WriteString(w, svgEnd)
}

// ...

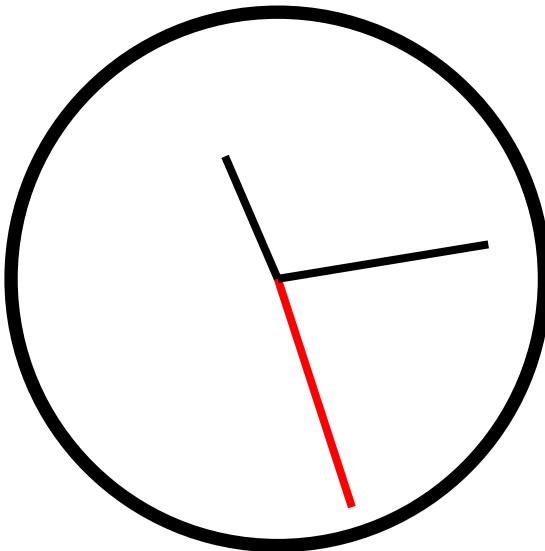
func hourHand(w io.Writer, t time.Time) {
    p := makeHand(hourHandPoint(t), hourHandLength)
```

```
    fmt.Fprintf(w, `<line x1="150" y1="150" x2="%f" y2="%f" style="fill:none;stroke:#000;`  
}
```

And so...

```
ok    clockface 0.007s
```

Let's just check by compiling and running our `clockface` program.



Refactor

Looking at `clockface.go`, there are a few 'magic numbers' floating about. They are all based around how many hours/minutes/seconds there are in a half-turn around a clockface. Let's refactor so that we make explicit their meaning.

```
const (  
    secondsInHalfClock = 30  
    secondsInClock    = 2 * secondsInHalfClock  
    minutesInHalfClock = 30  
    minutesInClock    = 2 * minutesInHalfClock  
    hoursInHalfClock   = 6
```

```
    hoursInClock    = 2 * hoursInHalfClock
)
```

Why do this? Well, it makes explicit what each number means in the equation. If - when - we come back to this code, these names will help us to understand what's going on.

Moreover, should we ever want to make some really, really WEIRD clocks - ones with 4 hours for the hour hand, and 20 seconds for the second hand say - these constants could easily become parameters. We're helping to leave that door open (even if we never go through it).

Wrapping up

Do we need to do anything else?

First, let's pat ourselves on the back - we've written a program that makes an SVG clockface. It works and it's great. It will only ever make one sort of clockface - but that's fine! Maybe you only want one sort of clockface. There's nothing wrong with a program that solves a specific problem and nothing else.

A Program... and a Library

But the code we've written does solve a more general set of problems to do with drawing a clockface. Because we used tests to think about each small part of the problem in isolation, and because we codified that isolation with functions, we've built a very reasonable little API for clockface calculations.

We can work on this project and turn it into something more general - a library for calculating clockface angles and/or vectors.

In fact, providing the library along with the program is a really good idea. It costs us nothing, while increasing the utility of our program and helping to document how it works.

APIs should come with programs, and vice versa. An API that you must write C code to use, which cannot be invoked easily from the command line, is harder to learn and use. And contrariwise, it's a royal pain to have interfaces whose only open, documented form is a program, so you cannot invoke them easily from a C program. -- Henry Spencer, in *The Art of Unix Programming*

In [my final take on this program](#), I've made the unexported functions within `clockface` into a public API for the library, with functions to calculate the angle and unit vector for each of the clock hands. I've also

split the SVG generation part into its own package, `svg`, which is then used by the `clockface` program directly. Naturally I've documented each of the functions and packages.

Talking about SVGs...

The Most Valuable Test

I'm sure you've noticed that the most sophisticated piece of code for handling SVGs isn't in our application code at all; it's in the test code. Should this make us feel uncomfortable? Shouldn't we do something like

- use a template from `text/template`?
- use an XML library (much as we're doing in our test)?
- use an SVG library?

We could refactor our code to do any of these things, and we can do so because it doesn't matter how we produce our SVG, what is important is what we produce - an SVG. As such, the part of our system that needs to know the most about SVGs - that needs to be the strictest about what constitutes an SVG - is the test for the SVG output: it needs to have enough context and knowledge about what an SVG is for us to be confident that we're outputting an SVG. The what of an SVG lives in our tests; the how in the code.

We may have felt odd that we were pouring a lot of time and effort into those SVG tests - importing an XML library, parsing XML, refactoring the structs - but that test code is a valuable part of our codebase - possibly more valuable than the current production code. It will help guarantee that the output is always a valid SVG, no matter what we choose to use to produce it.

Tests are not second class citizens - they are not 'throwaway' code. Good tests will last a lot longer than the version of the code they are testing. You should never feel like you're spending 'too much time' writing your tests. It is an investment.

Reading files

- [You can find all the code for this chapter here](#)
- [Here is a video of me working through the problem and taking questions from the Twitch stream](#)

In this chapter we're going to learn how to read some files, get some data out of them, and do something useful.

Pretend you're working with your friend to create some blog software. The idea is an author will write their posts in markdown, with some metadata at the top of the file. On startup, the web server will read a folder to create some Posts, and then a separate NewHandler function will use those Posts as a datasource for the blog's webserver.

We've been asked to create the package that converts a given folder of blog post files into a collection of Posts.

Example data

hello world.md

Title: Hello, TDD world!

Description: First post on our wonderful blog

Tags: tdd, go

Hello world!

The body of posts starts after the `---`

Expected data

```
type Post struct {
    Title, Description, Body string
    Tags                   []string
}
```

Iterative, test-driven development

We'll take an iterative approach where we're always taking simple, safe steps toward our goal.

This requires us to break up our work, but we should be careful not to fall into the trap of taking a "bottom up" approach.

We should not trust our over-active imaginations when we start work. We could be tempted into making some kind of abstraction that is only validated once we stick everything together, such as some kind of BlogPostFileParser.

This is not iterative and is missing out on the tight feedback loops that TDD is supposed to bring us.

Kent Beck says:

Optimism is an occupational hazard of programming. Feedback is the treatment.

Instead, our approach should strive to be as close to delivering real consumer value as quickly as possible (often called a "happy path"). Once we have delivered a small amount of consumer value end-to-end, further iteration of the rest of the requirements is usually straightforward.

Thinking about the kind of test we want to see

Let's remind ourselves of our mindset and goals when starting:

- **Write the test we want to see.** Think about how we'd like to use the code we're going to write from a consumer's point of view.
- Focus on what and why, but don't get distracted by how.

Our package needs to offer a function that can be pointed at a folder, and return us some posts.

```
var posts []blogposts.Post  
posts = blogposts.NewPostsFromFS("some-folder")
```

To write a test around this, we'd need some kind of test folder with some example posts in it. There's nothing terribly wrong with this, but you are making some trade-offs:

- for each test you may need to create new files to test a particular behaviour
- some behaviour will be challenging to test, such as failing to load files
- the tests will run a little slower because they will need to access the file system

We're also unnecessarily coupling ourselves to a specific implementation of the file system.

File system abstractions introduced in Go 1.16

Go 1.16 introduced an abstraction for file systems; the [io/fs](#) package.

Package `fs` defines basic interfaces to a file system. A file system can be provided by the host operating system but also by other packages.

This lets us loosen our coupling to a specific file system, which will then let us inject different implementations according to our needs.

[On the producer side of the interface, the new embed.FS type implements fs.FS, as does zip.Reader. The new os.DirFS function provides an implementation of fs.FS backed by a tree of operating system files.](#)

If we use this interface, users of our package have a number of options baked-in to the standard library to use. Learning to leverage interfaces defined in Go's standard library (e.g. `io.fs`, `io.Reader`, `io.Writer`), is vital to writing loosely coupled packages. These packages can then be re-used in contexts different to those you imagined, with minimal fuss from your consumers.

In our case, maybe our consumer wants the posts to be embedded into the Go binary rather than files in a "real" filesystem? Either way, our code doesn't need to care.

For our tests, the package `testing/fstest` offers us an implementation of `io/FS` to use, similar to the tools we're familiar with in `net/http/httptest`.

Given this information, the following feels like a better approach,

```
var posts []blogposts.Post  
posts = blogposts.NewPostsFromFS(someFS)
```

Write the test first

We should keep scope as small and useful as possible. If we prove that we can read all the files in a directory, that will be a good start. This will give us confidence in the software we're writing. We can check that the count of `[]Post` returned is the same as the number of files in our fake file system.

Create a new project to work through this chapter.

- `mkdir blogposts`
- `cd blogposts`
- `go mod init github.com/{your-name}/blogposts`
- `touch blogposts_test.go`

```
package blogposts_test  
  
import (  
    "testing"  
    "testing/fstest"  
)  
  
func TestNewBlogPosts(t *testing.T) {  
    fs := fstest.MapFS{  
        "hello world.md": {Data: []byte("hi")},  
        "hello-world2.md": {Data: []byte("hola")},  
    }  
  
    posts := blogposts.NewPostsFromFS(fs)
```

```
if len(posts) != len(fs) {
    t.Errorf("got %d posts, wanted %d posts", len(posts), len(fs))
}
```

Notice that the package of our test is `blogposts_test`. Remember, when TDD is practiced well we take a consumer-driven approach: we don't want to test internal details because consumers don't care about them. By appending `_test` to our intended package name, we only access exported members from our package - just like a real user of our package.

We've imported `testing/fstest` which gives us access to the `fstest.MapFS` type. Our fake file system will pass `fstest.MapFS` to our package.

A MapFS is a simple in-memory file system for use in tests, represented as a map from path names (arguments to `Open`) to information about the files or directories they represent.

This feels simpler than maintaining a folder of test files, and it will execute quicker.

Finally, we codified the usage of our API from a consumer's point of view, then checked if it creates the correct number of posts.

Try to run the test

```
./blogpost_test.go:15:12: undefined: blogposts
```

Write the minimal amount of code for the test to run and check the failing test output

The package doesn't exist. Create a new file `blogposts.go` and put package `blogposts` inside it. You'll need to then import that package into your tests. For me, the imports now look like:

```
import (
    blogposts "github.com/quii/learn-go-with-tests/reading-files"
    "testing"
    "testing/fstest"
)
```

Now the tests won't compile because our new package does not have a `NewPostsFromFS` function, that returns some kind of collection.

```
./blogpost_test.go:16:12: undefined: blogposts.NewPostsFromFS
```

This forces us to make the skeleton of our function to make the test run. Remember not to overthink the code at this point; we're only trying to get a running test, and to make sure it fails as we'd expect. If we skip this step we may skip over assumptions and, write a test which is not useful.

```
package blogposts

import "testing/fstest"

type Post struct {
}

func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
    return nil
}

The test should now correctly fail
==== RUN TestNewBlogPosts
blogposts_test.go:48: got 0 posts, wanted 2 posts
```

Write enough code to make it pass

We could "[slime](#)" this to make it pass:

```
func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
    return []Post{{}, {}}
}
```

But, as Denise Yu wrote:

Sliming is useful for giving a “skeleton” to your object. Designing an interface and executing logic are two concerns, and sliming tests strategically lets you focus on one at a time.

We already have our structure. So, what do we do instead?

As we've cut scope, all we need to do is read the directory and create a post for each file we encounter. We don't have to worry about opening files and parsing them just yet.

```
func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
    dir, _ := fs.ReadDir(fileSystem, ".")
    var posts []Post
    for range dir {
        posts = append(posts, Post{})
    }
}
```

```
    return posts
}
```

`fs.ReadDir` reads a directory inside a given `fs.FS` returning `[]DirEntry`.

Already our idealised view of the world has been foiled because errors can happen, but remember now our focus is making the test pass, not changing design, so we'll ignore the error for now.

The rest of the code is straightforward: iterate over the entries, create a Post for each one and, return the slice.

Refactor

Even though our tests are passing, we can't use our new package outside of this context, because it is coupled to a concrete implementation `fstest.MapFS`. But, it doesn't have to be. Change the argument to our `NewPostsFromFS` function to accept the interface from the standard library.

```
func NewPostsFromFS(fileSystem fs.FS) []Post {
    dir, _ := fs.ReadDir(fileSystem, ".")
    var posts []Post
    for range dir {
        posts = append(posts, Post{})
    }
    return posts
}
```

Re-run the tests: everything should be working.

Error handling

We parked error handling earlier when we focused on making the happy-path work. Before continuing to iterate on the functionality, we should acknowledge that errors can happen when working with files. Beyond reading the directory, we can run into problems when we open individual files. Let's change our API (via our tests first, naturally) so that it can return an error.

```
func TestNewBlogPosts(t *testing.T) {
    fs := fstest.MapFS{
        "hello-world.md": {Data: []byte("hi")},
        "hello-world2.md": {Data: []byte("hola")},
    }

    posts, err := blogposts.NewPostsFromFS(fs)
```

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

    if len(posts) != len(fs) {
        t.Errorf("got %d posts, wanted %d posts", len(posts), len(fs))
    }
}
```

Run the test: it should complain about the wrong number of return values. Fixing the code is straightforward.

```
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
    dir, err := fs.ReadDir(fileSystem, ".")
    if err != nil {
        return nil, err
    }
    var posts []Post
    for range dir {
        posts = append(posts, Post{})
    }
    return posts, nil
}
```

This will make the test pass. The TDD practitioner in you might be annoyed we didn't see a failing test before writing the code to propagate the error from `fs.ReadDir`. To do this "properly", we'd need a new test where we inject a failing `fs.FS` test-double to make `fs.ReadDir` return an error.

```
type StubFailingFS struct {
}

func (s StubFailingFS) Open(name string) (fs.File, error) {
    return nil, errors.New("oh no, i always fail")
}
// later
_, err := blogposts.NewPostsFromFS(StubFailingFS{})
```

This should give you confidence in our approach. The interface we're using has one method, which makes creating test-doubles to test different scenarios trivial.

In some cases, testing error handling is the pragmatic thing to do but, in our case, we're not doing anything interesting with the error, we're just propagating it, so it's not worth the hassle of writing a new test.

Logically, our next iterations will be around expanding our `Post` type so that it has some useful data.

Write the test first

We'll start with the first line in the proposed blog post schema, the title field.

We need to change the contents of the test files so they match what was specified, and then we can make an assertion that it is parsed correctly.

```
func TestNewBlogPosts(t *testing.T) {
    fs := ftest.MapFS{
        "hello world.md": {Data: []byte("Title: Post 1")},
        "hello-world2.md": {Data: []byte("Title: Post 2")},
    }

    // rest of test code cut for brevity
    got := posts[0]
    want := blogposts.Post{Title: "Post 1"}

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

Try to run the test

```
./blogpost_test.go:58:26: unknown field 'Title' in struct literal of type blogposts.Post
```

Write the minimal amount of code for the test to run and check the failing test output

Add the new field to our Post type so that the test will run

```
type Post struct {
    Title string
}
```

Re-run the test, and you should get a clear, failing test

```
==== RUN  TestNewBlogPosts
==== RUN  TestNewBlogPosts/parses_the_post
blogpost_test.go:61: got {Title:}, want {Title:Post 1}
```

Write enough code to make it pass

We'll need to open each file and then extract the title

```
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
    dir, err := fs.ReadDir(fileSystem, ".")
    if err != nil {
        return nil, err
    }
    var posts []Post
    for _, f := range dir {
        post, err := getPost(fileSystem, f)
        if err != nil {
            return nil, err //todo: needs clarification, should we totally fail if one file fails? or just ignore it?
        }
        posts = append(posts, post)
    }
    return posts, nil
}

func getPost(fileSystem fs.FS, f fs.DirEntry) (Post, error) {
    postFile, err := fileSystem.Open(f.Name())
    if err != nil {
        return Post{}, err
    }
    defer postFile.Close()

    postData, err := io.ReadAll(postFile)
    if err != nil {
        return Post{}, err
    }

    post := Post{Title: string(postData)[7:]}
    return post, nil
}
```

Remember our focus at this point is not to write elegant code, it's just to get to a point where we have working software.

Even though this feels like a small increment forward it still required us to write a fair amount of code and make some assumptions in respect to error handling. This would be a point where you should talk to your colleagues and decide the best approach.

The iterative approach has given us fast feedback that our understanding of the requirements is incomplete.

fs.FS gives us a way of opening a file within it by name with its Open method. From there we read the data from the file and, for now, we do not need any sophisticated parsing, just cutting out the Title: text by slicing the string.

Refactor

Separating the 'opening file code' from the 'parsing file contents code' will make the code simpler to understand and work with.

```
func getPost(fileSystem fs.FS, f fs.DirEntry) (Post, error) {
    postFile, err := fileSystem.Open(f.Name())
    if err != nil {
        return Post{}, err
    }
    defer postFile.Close()
    return newPost(postFile)
}

func newPost(postFile fs.File) (Post, error) {
    postData, err := io.ReadAll(postFile)
    if err != nil {
        return Post{}, err
    }
    post := Post{Title: string(postData)[7:]}
    return post, nil
}
```

When you refactor out new functions or methods, take care and think about the arguments. You're designing here, and are free to think deeply about what is appropriate because you have passing tests. Think about coupling and cohesion. In this case you should ask yourself:

Does newPost have to be coupled to an fs.File ? Do we use all the methods and data from this type? What do we really need?

In our case we only use it as an argument to io.ReadAll which needs an io.Reader. So we should loosen the coupling in our function and ask for an io.Reader.

```
func newPost(postFile io.Reader) (Post, error) {
    postData, err := io.ReadAll(postFile)
    if err != nil {
        return Post{}, err
    }
    post := Post{Title: string(postData)[7:]}
    return post, nil
}
```

You can make a similar argument for our getPost function, which takes

an `fs.DirEntry` argument but simply calls `Name()` to get the file name. We don't need all that; let's decouple from that type and pass the file name through as a string. Here's the fully refactored code:

```
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
    dir, err := fs.ReadDir(fileSystem, ".")
    if err != nil {
        return nil, err
    }
    var posts []Post
    for _, f := range dir {
        post, err := getPost(fileSystem, f.Name())
        if err != nil {
            return nil, err //todo: needs clarification, should we totally fail if one file fails? or just ignore it?
        }
        posts = append(posts, post)
    }
    return posts, nil
}

func getPost(fileSystem fs.FS, fileName string) (Post, error) {
    postFile, err := fileSystem.Open(fileName)
    if err != nil {
        return Post{}, err
    }
    defer postFile.Close()
    return newPost(postFile)
}

func newPost(postFile io.Reader) (Post, error) {
    postData, err := io.ReadAll(postFile)
    if err != nil {
        return Post{}, err
    }

    post := Post{Title: string(postData)[7:]}
    return post, nil
}
```

From now on, most of our efforts can be neatly contained within `newPost`. The concerns of opening and iterating over files are done, and now we can focus on extracting the data for our `Post` type. Whilst not technically necessary, files are a nice way to logically group related things together, so I moved the `Post` type and `newPost` into a new `post.go` file.

Test helper

We should take care of our tests too. We're going to be making assertions on Posts a lot, so we should write some code to help with that

```
func assertPost(t *testing.T, got blogposts.Post, want blogposts.Post) {
    t.Helper()
    if !reflect.DeepEqual(got, want) {
        t.Errorf("got %+v, want %+v", got, want)
    }
}

assertPost(t, posts[0], blogposts.Post{Title: "Post 1"})
```

Write the test first

Let's extend our test further to extract the next line from the file, the description. Up until making it pass should now feel comfortable and familiar.

```
func TestNewBlogPosts(t *testing.T) {
    const (
        firstBody = `Title: Post 1
Description: Description 1`
        secondBody = `Title: Post 2
Description: Description 2`
    )

    fs := ftest.MapFS{
        "hello world.md": {Data: []byte(firstBody)},
        "hello-world2.md": {Data: []byte(secondBody)},
    }

    // rest of test code cut for brevity
    assertPost(t, posts[0], blogposts.Post{
        Title:      "Post 1",
        Description: "Description 1",
    })
}
```

Try to run the test

```
./blogpost_test.go:47:58: unknown field 'Description' in struct literal of type blogposts.Post
```

Write the minimal amount of code for the test to run and check the failing test output

Add the new field to Post.

```
type Post struct {
    Title string
    Description string
}
```

The tests should now compile, and fail.

```
==== RUN TestNewBlogPosts
blogpost_test.go:47: got {Title:Post 1
Description: Description 1 Description:}, want {Title:Post 1 Description:Description 1}
```

Write enough code to make it pass

The standard library has a handy library for helping you scan through data, line by line; [bufio.Scanner](#)

Scanner provides a convenient interface for reading data such as a file of newline-delimited lines of text.

```
func newPost(postFile io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postFile)

    scanner.Scan()
    titleLine := scanner.Text()

    scanner.Scan()
    descriptionLine := scanner.Text()

    return Post{Title: titleLine[7:], Description: descriptionLine[13:]}, nil
}
```

Handily, it also takes an `io.Reader` to read through (thank you again, loose-coupling), we don't need to change our function arguments.

Call `Scan` to read a line, and then extract the data using `Text`.

This function could never return an error. It would be tempting at this point to remove it from the return type, but we know we'll have to handle invalid file structures later so, we may as well leave it.

Refactor

We have repetition around scanning a line and then reading the text. We know we're going to do this operation at least one more time, it's

a simple refactor to DRY up so let's start with that.

```
func newPost(postFile io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postFile)

    readLine := func() string {
        scanner.Scan()
        return scanner.Text()
    }

    title := readLine()[7:]
    description := readLine()[13:]

    return Post{Title: title, Description: description}, nil
}
```

This has barely saved any lines of code, but that's rarely the point of refactoring. What I'm trying to do here is just separating the what from the how of reading lines to make the code a little more declarative to the reader.

Whilst the magic numbers of 7 and 13 get the job done, they're not awfully descriptive.

```
const (
    titleSeparator     = "Title: "
    descriptionSeparator = "Description: "
)

func newPost(postFile io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postFile)

    readLine := func() string {
        scanner.Scan()
        return scanner.Text()
    }

    title := readLine()[len(titleSeparator):]
    description := readLine()[len(descriptionSeparator):]

    return Post{Title: title, Description: description}, nil
}
```

Now that I'm staring at the code with my creative refactoring mind, I'd like to try making our `readLine` function take care of removing the tag. There's also a more readable way of trimming a prefix from a string with the function `strings.TrimPrefix`.

```
func newPost(postBody io.Reader) (Post, error) {
```

```
scanner := bufio.NewScanner(postBody)

readMetaLine := func(tagName string) string {
    scanner.Scan()
    return strings.TrimPrefix(scanner.Text(), tagName)
}

return Post{
    Title:    readMetaLine(titleSeparator),
    Description: readMetaLine(descriptionSeparator),
}, nil
}
```

You may or may not like this idea, but I do. The point is in the refactoring state we are free to play with the internal details, and you can keep running your tests to check things still behave correctly. We can always go back to previous states if we're not happy. The TDD approach gives us this license to frequently experiment with ideas, so we have more shots at writing great code.

The next requirement is extracting the post's tags. If you're following along, I'd recommend trying to implement it yourself before reading on. You should now have a good, iterative rhythm and feel confident to extract the next line and parse out the data.

For brevity, I will not go through the TDD steps, but here's the test with tags added.

```
func TestNewBlogPosts(t *testing.T) {
    const (
        firstBody = `Title: Post 1
Description: Description 1
Tags: tdd, go`
        secondBody = `Title: Post 2
Description: Description 2
Tags: rust, borrow-checker`
    )

    // rest of test code cut for brevity
    assertPost(t, posts[0], blogposts.Post{
        Title:      "Post 1",
        Description: "Description 1",
        Tags:      []string{"tdd", "go"},
    })
}
```

You're only cheating yourself if you just copy and paste what I write. To make sure we're all on the same page, here's my code which includes

extracting the tags.

```
const (
    titleSeparator      = "Title: "
    descriptionSeparator = "Description: "
    tagsSeparator       = "Tags: "
)

func newPost(postBody io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postBody)

    readMetaLine := func(tagName string) string {
        scanner.Scan()
        return strings.TrimPrefix(scanner.Text(), tagName)
    }

    return Post{
        Title:      readMetaLine(titleSeparator),
        Description: readMetaLine(descriptionSeparator),
        Tags:       strings.Split(readMetaLine(tagsSeparator), ", "),
    }, nil
}
```

Hopefully no surprises here. We were able to re-use `readMetaLine` to get the next line for the tags and then split them up using `strings.Split`.

The last iteration on our happy path is to extract the body.

Here's a reminder of the proposed file format.

Title: Hello, TDD world!

Description: First post on our wonderful blog

Tags: tdd, go

Hello world!

The body of posts starts after the `---`

We've read the first 3 lines already. We then need to read one more line, discard it and then the remainder of the file contains the post's body.

Write the test first

Change the test data to have the separator, and a body with a few newlines to check we grab all the content.

```
const (
    firstBody = `Title: Post 1
```

```
Description: Description 1
Tags: tdd, go
---
Hello
World`  
    secondBody = `Title: Post 2
Description: Description 2
Tags: rust, borrow-checker
---
B
L
M`  
)
```

Add to our assertion like the others

```
assertPost(t, posts[0], blogposts.Post{
    Title:      "Post 1",
    Description: "Description 1",
    Tags:       []string{"tdd", "go"},
    Body:       `Hello
World`,
})
```

Try to run the test

```
./blogpost_test.go:60:3: unknown field 'Body' in struct literal of type blogposts.Post
As we'd expect.
```

Write the minimal amount of code for the test to run and check the failing test output

Add Body to Post and the test should fail.

```
==== RUN TestNewBlogPosts
blogposts_test.go:38: got {Title:Post 1 Description:Description 1 Tags:[tdd go] Body:}, want {  
    World}
```

Write enough code to make it pass

1. Scan the next line to ignore the --- separator.
2. Keep scanning until there's nothing left to scan.

```
func newPost(postBody io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postBody)
```

```
readMetaLine := func(tagName string) string {
    scanner.Scan()
    return strings.TrimPrefix(scanner.Text(), tagName)
}

title := readMetaLine(titleSeparator)
description := readMetaLine(descriptionSeparator)
tags := strings.Split(readMetaLine(tagsSeparator), ", ")

scanner.Scan() // ignore a line

buf := bytes.Buffer{}
for scanner.Scan() {
    fmt.Fprintln(&buf, scanner.Text())
}
body := strings.TrimSuffix(buf.String(), "\n")

return Post{
    Title: title,
    Description: description,
    Tags: tags,
    Body: body,
}, nil
}
```

- `scanner.Scan()` returns a bool which indicates whether there's more data to scan, so we can use that with a for loop to keep reading through the data until the end.
- After every `Scan()` we write the data into the buffer using `fmt.Fprintln`. We use the version that adds a newline because the scanner removes the newlines from each line, but we need to maintain them.
- Because of the above, we need to trim the final newline, so we don't have a trailing one.

Refactor

Encapsulating the idea of getting the rest of the data into a function will help future readers quickly understand what is happening in `newPost`, without having to concern themselves with implementation specifics.

```
func newPost(postBody io.Reader) (Post, error) {
    scanner := bufio.NewScanner(postBody)

    readMetaLine := func(tagName string) string {
```

```
scanner.Scan()
    return strings.TrimPrefix(scanner.Text(), tagName)
}

return Post{
    Title:    readMetaLine(titleSeparator),
    Description: readMetaLine(descriptionSeparator),
    Tags:    strings.Split(readMetaLine(tagsSeparator), ", "),
    Body:    readBody(scanner),
}, nil
}

func readBody(scanner *bufio.Scanner) string {
    scanner.Scan() // ignore a line
    buf := bytes.Buffer{}
    for scanner.Scan() {
        fmt.Fprintln(&buf, scanner.Text())
    }
    return strings.TrimSuffix(buf.String(), "\n")
}
```

Iterating further

We've made our "steel thread" of functionality, taking the shortest route to get to our happy path, but clearly there's some distance to go before it is production ready.

We haven't handled:

- when the file's format is not correct
- the file is not a .md
- what if the order of the metadata fields is different? Should that be allowed? Should we be able to handle it?

Crucially though, we have working software, and we have defined our interface. The above are just further iterations, more tests to write and drive our behaviour. To support any of the above we shouldn't have to change our design, just implementation details.

Keeping focused on the goal means we made the important decisions, and validated them against the desired behaviour, rather than getting bogged down on matters that won't affect the overall design.

Wrapping up

fs.FS, and the other changes in Go 1.16 give us some elegant ways of reading data from file systems and testing them simply.

If you wish to try out the code "for real":

- Create a cmd folder within the project, add a main.go file
- Add the following code

```
import (
    blogposts "github.com/quii/fstest-spike"
    "log"
    "os"
)

func main() {
    posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
    if err != nil {
        log.Fatal(err)
    }
    log.Println(posts)
}
```

- Add some markdown files into a posts folder and run the program!

Notice the symmetry between the production code

```
posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
```

And the tests

```
posts, err := blogposts.NewPostsFromFS(fs)
```

This is when consumer-driven, top-down TDD feels correct.

A user of our package can look at our tests and quickly get up to speed with what it's supposed to do and how to use it. As maintainers, we can be confident our tests are useful because they're from a consumer's point of view. We're not testing implementation details or other incidental details, so we can be reasonably confident that our tests will help us, rather than hinder us when refactoring.

By relying on good software engineering practices like **dependency injection** our code is simple to test and re-use.

When you're creating packages, even if they're only internal to your project, prefer a top-down consumer driven approach. This will stop you over-imagining designs and making abstractions you may not even need and will help ensure the tests you write are useful.

The iterative approach kept every step small, and the continuous feedback helped us uncover unclear requirements possibly sooner than with other, more ad-hoc approaches.

Writing?

It's important to note that these new features only have operations for reading files. If your work needs to do writing, you'll need to look elsewhere. Remember to keep thinking about what the standard library offers currently, if you're writing data you should probably look into leveraging existing interfaces such as `io.Writer` to keep your code loosely-coupled and re-usable.

Further reading

- This was a light intro to `io/fs`. [Ben Congdon has done an excellent write-up](#) which was a lot of help for writing this chapter.
- [Discussion on the file system interfaces](#)

HTML Templates

You can find all the code here

We live in a world where everyone wants to build web applications with the latest flavour of the month frontend framework built upon gigabytes of transpiled JavaScript, working with a Byzantine build system; [but maybe that's not always necessary](#).

I'd say most Go developers value a simple, stable & fast toolchain but the frontend world frequently fails to deliver on this front.

Many websites do not need to be an [SPA](#). **HTML and CSS are fantastic ways of delivering content** and you can use Go to make a website to deliver HTML.

If you wish to still have some dynamic elements, you can still sprinkle in some client side JavaScript, or you may even want to try experimenting with [Hotwire](#) which allows you to deliver a dynamic experience with a server-side approach.

You can generate your HTML in Go with elaborate usage of `fmt.Fprintf`, but in this chapter you'll learn that Go's standard library has some tools to generate HTML in a simpler and more maintainable way. You'll also learn more effective ways of testing this kind of code that you may not have run in to before.

What we're going to build

In the [Reading Files](#) chapter we wrote some code that would take an `fs.FS` (a file-system), and return a slice of `Post` for each markdown file it encountered.

```
posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
```

Here is how we defined Post

```
type Post struct {
    Title, Description, Body string
    Tags                   []string
}
```

Here's an example of one of the markdown files that can be parsed.

Title: Welcome to my blog

Description: Introduction to my blog

Tags: cooking, family, live-laugh-love

First recipe!

Welcome to my **amazing recipe blog**. I am going to write about my family recipes, and make

If we continue our journey of writing blog software, we'd take this data and generate HTML from it for our web server to return in response to HTTP requests.

For our blog, we want to generate two kinds of page:

1. **View post.** Renders a specific post. The Body field in Post is a string containing markdown so that should be converted to HTML.
2. **Index.** Lists all of the posts, with hyperlinks to view the specific post.

We'll also want a consistent look and feel across our site, so for each page we'll have the usual HTML furniture like <html> and a <head> containing links to CSS stylesheets and whatever else we may want.

When you're building blog software you have a few options in terms of approach of how you build and send HTML to the user's browser.

We'll design our code so it accepts an io.Writer. This means the caller of our code has the flexibility to:

- Write them to an [os.File](#), so they can be statically served
- Write out the HTML directly to a [http.ResponseWriter](#)
- Or just write them to anything really! So long as it implements io.Writer the user can generate some HTML from a Post

Write the test first

As always, it's important to think about requirements before diving in too fast. How can we take this large-ish set of requirements and break it down in to a small, achievable step that we can focus on?

In my view, actually viewing content is higher priority than an index page. We could launch this product and share direct links to our wonderful content. An index page which can't link to the actual content isn't useful.

Still, rendering a post as described earlier still feels big. All the HTML furniture, converting the body markdown into HTML, listing tags, e.t.c.

At this stage I'm not overly concerned with the specific markup, and an easy first step would be just to check we can render the post's title as an `<h1>`. This feels like the smallest first step that can move us forward a bit.

```
package blogrenderer_test

import (
    "bytes"
    "github.com/quii/learn-go-with-tests/blogrenderer"
    "testing"
)

func TestRender(t *testing.T) {
    var (
        aPost = blogrenderer.Post{
            Title:      "hello world",
            Body:       "This is a post",
            Description: "This is a description",
            Tags:       []string{"go", "tdd"},
        }
    )

    t.Run("it converts a single post into HTML", func(t *testing.T) {
        buf := bytes.Buffer{}
        err := blogrenderer.Render(&buf, aPost)

        if err != nil {
            t.Fatal(err)
        }

        got := buf.String()
        want := `<h1>hello world</h1>`
        if got != want {
            t.Errorf("got '%s' want '%s'", got, want)
        }
    })
}
```

Our decision to accept an `io.Writer` also makes testing simple, in this

case we're writing to a `bytes.Buffer` which we can then later inspect the contents.

Try to run the test

If you've read the previous chapters of this book you should be well-practiced at this now. You won't be able to run the test because we don't have the package defined or the `Render` function. Try and follow the compiler messages yourself and get to a state where you can run the test and see that it fails with a clear message.

It's really important that you exercise your tests failing, you'll thank yourself when you accidentally make a test fail 6 months later that you put in the effort now to check it fails with a clear message.

Write the minimal amount of code for the test to run and check the failing test output

This is the minimal code to get the test running

```
package blogrenderer
```

```
// if you're continuing from the read files chapter, you shouldn't redefine this
type Post struct {
    Title, Description, Body string
    Tags                  []string
}

func Render(w io.Writer, p Post) error {
    return nil
}
```

The test should complain that an empty string doesn't equal what we want.

Write enough code to make it pass

```
func Render(w io.Writer, p Post) error {
    _, err := fmt.Fprintf(w, "<h1>%s</h1>", p.Title)
    return err
}
```

Remember, software development is primarily a learning activity. In order to discover and learn as we work, we need to work in a way that gives us frequent, high-quality feedback loops, and the easiest way to do that is work in small steps.

So we're not worrying about using any templating libraries right now. You can make HTML just with "normal" string templating just fine, and by skipping the template part we can validate a small bit of useful behaviour and we've done a small bit of design work for our package's API.

Refactor

Not much to refactor yet, so let's move to the next iteration

Write the test first

Now we have a very basic version working, we can now iterate on the test to expand on the functionality. In this case, rendering more information from the Post.

```
t.Run("it converts a single post into HTML", func(t *testing.T) {
    buf := bytes.Buffer{}
    err := blogrenderer.Render(&buf, aPost)

    if err != nil {
        t.Fatal(err)
    }

    got := buf.String()
    want := `<h1>hello world</h1>
<p>This is a description</p>
Tags: <ul><li>go</li><li>tdd</li></ul>`
```

```
    if got != want {
        t.Errorf("got '%s' want '%s'", got, want)
    }
})
```

Notice that writing this, feels awkward. Seeing all that markup in the test feels bad, and we haven't even put the body in, or the actual HTML we'd want with all of the `<head>` content and whatever page furniture we need.

Nonetheless, let's put up with the pain for now.

Try to run the test

It should fail, complaining it doesn't have the string we expect, as we're not rendering the description and tags.

Write enough code to make it pass

Try and do this yourself rather than copying the code. What you should find is that making this test pass is a bit annoying! When I tried, my first attempt got this error

```
==== RUN TestRender
==== RUN TestRender/it Converts_a_single_post_into_HTML
    renderer_test.go:32: got '<h1>hello world</h1><p>This is a description</p><ul><li>go</li><li>Tags: <ul><li>go</li><li></li></ul>'</p>This is a description</p>
    Tags: <ul><li>go</li><li></li></ul>'
```

New lines! Who cares? Well, our test does, because it's matching on an exact string value. Should it? I removed the newlines for now just to get the test passing.

```
func Render(w io.Writer, p Post) error {
    _, err := fmt.Fprintf(w, "<h1>%s</h1><p>%s</p>", p.Title, p.Description)
    if err != nil {
        return err
    }

    _, err = fmt.Fprint(w, "Tags: <ul>")
    if err != nil {
        return err
    }

    for _, tag := range p.Tags {
        _, err = fmt.Fprintf(w, "<li>%s</li>", tag)
        if err != nil {
            return err
        }
    }

    _, err = fmt.Fprint(w, "</ul>")
    if err != nil {
        return err
    }

    return nil
}
```

Yikes. Not the nicest code i've written, and we're still only at a very early implementation of our markup. We'll need so much more content and things on our page, we're quickly seeing that this approach is not appropriate.

Crucially though, we have a passing test; we have working software.

Refactor

With the safety-net of a passing test for working code, we can now think about changing our implementation approach at the refactoring stage.

Introducing templates

Go has two templating packages [text/template](#) and [html/template](#) and they share the same interface. What they both do is allow you to combine a template and some data to produce a string.

What's the difference with the HTML version?

Package template (html/template) implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package text/template and should be used instead of text/template whenever the output is HTML.

The templating language is very similar to [Mustache](#) and allows you to dynamically generate content in a very clean fashion with a nice separation of concerns. Compared to other templating languages you may have used, it is very constrained or "logic-less" as Mustache likes to say. This is an important, **and deliberate** design decision.

Whilst we're focusing on generating HTML here, if your project is doing complex string concatenations and incantations, you might want to reach for text/template to clean up your code.

Back to the code

Here is a template for our blog:

```
<h1>{{.Title}}</h1><p>{{.Description}}</p>Tags: <ul>{{range .Tags}}<li>{{.}}</li>{{end}}</ul>
```

Where do we define this string? Well, we have a few options, but to keep the steps small, let's just start with a plain old string

```
package blogrenderer
```

```
import (
    "html/template"
    "io"
)

const (
    postTemplate = `<h1>{{.Title}}</h1><p>{{.Description}}</p>Tags: <ul>{{range .Tags}}
```

```
)  
  
func Render(w io.Writer, p Post) error {  
    templ, err := template.New("blog").Parse(postTemplate)  
    if err != nil {  
        return err  
    }  
  
    if err := templ.Execute(w, p); err != nil {  
        return err  
    }  
  
    return nil  
}
```

We create a new template with a name, and then parse our template string. We can then use the Execute method on it, passing in our data, in this case the Post.

The template will substitute things like `{.Description}` with the content of `p.Description`. Templates also give you some programming primitives like range to loop over values, and if. You can find more details in the [text/template documentation](#).

This should be a pure refactor. We shouldn't need to change our tests and they should continue to pass. Importantly, our code is easier to read and has far less annoying error handling to contend with.

Frequently people complain about the verbosity of error handling in Go, but you might find you can find better ways to write your code so it's less error-prone in the first place, like here.

More refactoring

Using the html/template has definitely been an improvement, but having it as a string constant in our code isn't great:

- It's still quite difficult to read.
- It's not IDE/editor friendly. No syntax highlighting, ability to reformat, refactor, e.t.c.
- It looks like HTML, but you can't really work with it like you could a "normal" HTML file

What we'd like to do is have our templates live in separate files so we can better organise them, and work with them as if they're HTML files.

Create a folder called "templates" and inside it make a file called `blog.gohtml`, paste our template into the file.

Now change our code to embed the file systems using the [embedding functionality included in go 1.16](#).

```
package blogrendererer

import (
    "embed"
    "html/template"
    "io"
)

var (
    //go:embed "templates/*"
    postTemplates embed.FS
)

func Render(w io.Writer, p Post) error {
    templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
    if err != nil {
        return err
    }

    if err := templ.Execute(w, p); err != nil {
        return err
    }

    return nil
}
```

By embedding a "file system" into our code, we can load multiple templates and combine them freely. This will become useful when we want to share rendering logic across different templates, such as a header for the top of the HTML page and a footer.

Embed?

Embed was lightly touched on in [reading files](#). The [documentation from the standard library explains](#)

Package embed provides access to files embedded in the running Go program.

Go source files that import "embed" can use the //go:embed directive to initialize a variable of type string, []byte, or FS with the contents of files read from the package directory or subdirectories at compile time.

Why would we want to use this? Well the alternative is that we can

load our templates from a "normal" file system. However this means we'd have to make sure that the templates are in the correct file path wherever we want to use this software. In your job you may have various environments like development, staging and live. For this to work, you'd need to make sure your templates are copied to the correct place.

With embed, the files are included in your Go program when you build it. This means once you've built your program (which you should only do once), the files are always available to you.

What's handy is you can not only embed individual files, but also file systems; and that filesystem implements [io/fs](#) which means your code doesn't need to care what kind of file system it is working with.

If you wish to use different templates depending on configuration though, you may wish to stick to loading templates from disk in the more conventional way.

Next: Make the template "nice"

We don't really want our template to be defined as a one line string. We want to be able to space it out to make it easier to read and work with, something like this:

```
<h1>{{.Title}}</h1>  
  
<p>{{.Description}}</p>
```

Tags: `{{range .Tags}}{{.}}{{end}}`

But if we do this, our test fails. This is because our test is expecting a very specific string to be returned.

But really, we don't actually care about whitespace. Maintaining this test will become a nightmare if we have to keep painstakingly updating the assertion string every time we make minor changes to the markup. As the template grows, these kind of edits become harder to manage and the costs of work will spiral out of control.

Introducing Approval Tests

Go Approval Tests

ApprovalTests allows for easy testing of larger objects, strings and anything else that can be saved to a file (images, sounds, CSV, etc...)

The idea is similar to "golden" files, or snapshot testing. Rather than awkwardly maintaining strings within a test file, the approval tool can compare the output for you with an "approved" file you created. You then simply copy over the new version if you approve it. Re-run the test and you're back to green.

Add a dependency to "github.com/approvals/go-approval-tests" to your project and edit the test to the following

```
func TestRender(t *testing.T) {
    var (
        aPost = blogrenderer.Post{
            Title:      "hello world",
            Body:       "This is a post",
            Description: "This is a description",
            Tags:       []string{"go", "tdd"},
        }
    )

    t.Run("it converts a single post into HTML", func(t *testing.T) {
        buf := bytes.Buffer{}

        if err := blogrenderer.Render(&buf, aPost); err != nil {
            t.Fatal(err)
        }

        approvals.VerifyString(t, buf.String())
    })
}
```

The first time you run it, it will fail because we haven't approved anything yet

```
==== RUN TestRender
==== RUN TestRender/it Converts_a_single_post_into_HTML
    renderer_test.go:29: Failed Approval: received does not match approved.
```

It will have created two files, that look like the following

- renderer_test.TestRender.it_Converts_a_single_post_into_HTML.received.txt
- renderer_test.TestRender.it_Converts_a_single_post_into_HTML.approved.txt

The received file has the new, unapproved version of the output. Copy that into the empty approved file and re-run the test.

By copying the new version you have "approved" the change, and the test now passes.

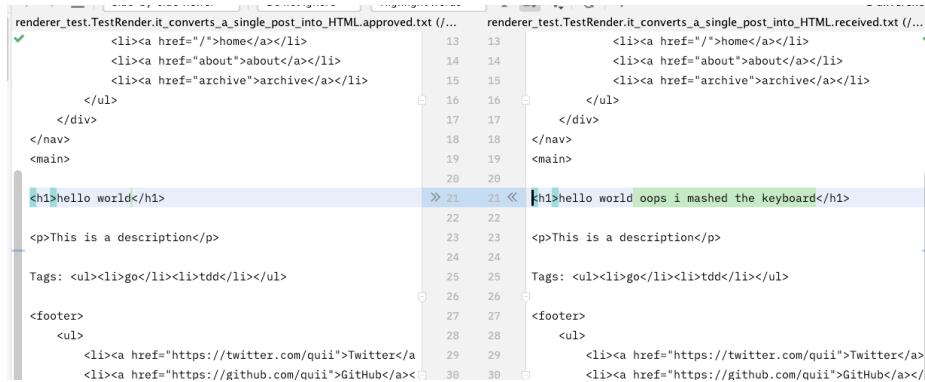
To see the workflow in action, edit the template to how we discussed to make it easier to read (but semantically, it's the same).

```
<h1>{{.Title}}</h1>  
<p>{{.Description}}</p>
```

Tags: `{{range .Tags}}{{.}}{{end}}`

Re-run the test. A new "received" file will be generated because the output of our code differs to the approved version. Give them a look, and if you're happy with the changes, simply copy over the new version and re-run the test. Be sure to commit the approved files to source control.

This approach makes managing changes to big ugly things like HTML far simpler. You can use a diff tool to view and manage the differences, and it keeps your test code cleaner.



```
renderer_test.TestRender.it Converts_a_single_post_into_HTML.approved.txt (/... renderer_test.TestRender.it Converts_a_single_post_into_HTML.received.txt (/...  
  <li><a href="/">home</a></li>  13 13  
  <li><a href="about">about</a></li>  14 14  
  <li><a href="archive">archive</a></li>  15 15  
  </ul>  16 16  
  </div>  17 17  
</nav>  18 18  
<main>  19 19  
  
<h1>hello world</h1>  21 21 <<  <h1>hello world oops i mashed the keyboard</h1>  
  
<p>This is a description</p>  
  
Tags: <ul><li>go</li><li>tdd</li></ul>  22 22  
  
<footer>  23 23  
  <ul>  24 24  
    <li><a href="https://twitter.com/quii">Twitter</a>  25 25  
    <li><a href="https://github.com/quii">GitHub</a>  26 26  
  </ul>  27 27  
</footer>  28 28  
  <ul>  29 29  
    <li><a href="https://twitter.com/quii">Twitter</a>  30 30  
    <li><a href="https://github.com/quii">GitHub</a>  30 30
```

This is actually a fairly minor usage of approval tests, which are an extremely useful tool in your testing arsenal. [Emily Bache](#) has an [interesting video where she uses approval tests to add an incredibly extensive set of tests to a complicated codebase that has zero tests](#). "Combinatorial Testing" is definitely something worth looking into.

Now that we have made this change, we still benefit from having our code well-tested, but the tests won't get in the way too much when we're tinkering with the markup.

Are we still doing TDD?

An interesting side-effect of this approach is it takes us away from TDD. Of course you could manually edit the approved files to the state you want, run your tests and then fix the templates so they output what you defined.

But that's just silly! TDD is a method for doing work, specifically designing; but that doesn't mean we have to dogmatically use it for

everything.

The important thing is, we've done the right thing and used TDD as a **design tool** to design our package's API. For templates changes our process can be:

- Make a small change to the template
- Run the approval test
- Eyeball the output to check it looks correct
- Make the approval
- Repeat

We still shouldn't give up the value of working in small achievable steps. Try to find ways to make the changes small and keep re-running the tests to get real feedback on what you're doing.

If we start doing things like changing the code around the templates, then of course that may warrant going back to our TDD method of work.

Expand the markup

Most websites have richer HTML than we have right now. For starters, a html element, along with a head, perhaps some nav too. Usually there's an idea of a footer too.

If our site is going to have different pages, we'd want to define these things in one place to keep our site looking consistent. Go templates support us defining sections which we can then import in to other templates.

Edit our existing template to import a top and bottom template

```
 {{template "top" .}}
<h1>{{.Title}}</h1>

<p>{{.Description}}</p>
```

Tags: {{range .Tags}}{{.}}{{end}}
{{template "bottom" .}}

Then create top.gohtml with the following

```
 {{define "top"}}
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My amazing blog!</title>
  <meta charset="UTF-8"/>
  <meta name="description" content="Wow, like and subscribe, it really helps the channel gu
```

```
</head>
<body>
<nav role="navigation">
  <div>
    <h1>Budding Gopher's blog</h1>
    <ul>
      <li><a href="/">home</a></li>
      <li><a href="about">about</a></li>
      <li><a href="archive">archive</a></li>
    </ul>
  </div>
</nav>
<main>
{{end}}
```

And bottom.gohtml

```
{{define "bottom"}}
</main>
<footer>
  <ul>
    <li><a href="https://twitter.com/quii">Twitter</a></li>
    <li><a href="https://github.com/quii">GitHub</a></li>
  </ul>
</footer>
</body>
</html>
{{end}}
```

(Obviously, feel free to put whatever markup you like!)

We now need to specify a specific template to run. In the blog renderer, change the Execute command to ExecuteTemplate

```
if err := templ.ExecuteTemplate(w, "blog.gohtml", p); err != nil {
  return err
}
```

Re-run your test. A new "received" file should be made and the test will fail. Check it over and if you're happy, approve it by copying it over the old version. Re-run the test again and it should pass.

An excuse to mess around with Benchmarking

Before pressing on, let's consider what our code does.

```
func Render(w io.Writer, p Post) error {
  templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
  if err != nil {
```

```
        return err
    }

    if err := templ.ExecuteTemplate(w, "blog.gohtml", p); err != nil {
        return err
    }

    return nil
}



- Parse the templates
- Use the template to render a post to an io.Writer

```

Whilst the performance impact of re-parsing the templates for each post in most cases will be fairly negligible, the effort to not do this is also pretty negligible and should tidy the code up a bit too.

To see the impact of not doing this parsing over and over, we can use the benchmarking tool to see how fast our function is.

```
func BenchmarkRender(b *testing.B) {
    var (
        aPost = blogrenderer.Post{
            Title:      "hello world",
            Body:       "This is a post",
            Description: "This is a description",
            Tags:       []string{"go", "tdd"},
        }
    )

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        blogrenderer.Render(io.Discard, aPost)
    }
}
```

On my computer, here are the results

```
BenchmarkRender-8 22124 53812 ns/op
```

To stop us having to re-parse the templates over and over, we'll create a type that'll hold the parsed template, and that'll have a method to do the rendering

```
type PostRenderer struct {
    templ *template.Template
}

func NewPostRenderer() (*PostRenderer, error) {
    templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
```

```
    if err != nil {
        return nil, err
    }

    return &PostRenderer{templ: templ}, nil
}

func (r *PostRenderer) Render(w io.Writer, p Post) error {
    if err := r.templ.ExecuteTemplate(w, "blog.gohtml", p); err != nil {
        return err
    }

    return nil
}
```

This does change the interface of our code, so we'll need to update our test

```
func TestRender(t *testing.T) {
    var (
        aPost = blogrenderer.Post{
            Title:      "hello world",
            Body:       "This is a post",
            Description: "This is a description",
            Tags:       []string{"go", "tdd"},
        }
    )

    postRenderer, err := blogrenderer.NewPostRenderer()

    if err != nil {
        t.Fatal(err)
    }

    t.Run("it converts a single post into HTML", func(t *testing.T) {
        buf := bytes.Buffer{}

        if err := postRenderer.Render(&buf, aPost); err != nil {
            t.Fatal(err)
        }

        approvals.VerifyString(t, buf.String())
    })
}
```

And our benchmark

```
func BenchmarkRender(b *testing.B) {
    var (
        aPost = blogrenderer.Post{
            Title:      "hello world",
            Body:       "This is a post",
            Description: "This is a description",
            Tags:       []string{"go", "tdd"},
        }
    )

    postRenderer, err := blogrenderer.NewPostRenderer()

    if err != nil {
        b.Fatal(err)
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        postRenderer.Render(io.Discard, aPost)
    }
}
```

The test should continue to pass. How about our benchmark?

BenchmarkRender-8 362124 3131 ns/op. The old NS per op were 53812 ns/op, so this is a decent improvement! As we add other methods to render, say an Index page, it should simplify the code as we don't need to duplicate the template parsing.

Back to the real work

In terms of rendering posts, the important part left is actually rendering the Body. If you recall, that should be markdown that the author has written, so it'll need converting to HTML.

We'll leave this as an exercise for you, the reader. You should be able to find a Go library to do this for you. Use the approval test to validate what you're doing.

On testing 3rd-party libraries

Note. Be careful not to worry too much about explicitly testing how a 3rd party library behaves in unit tests.

Writing tests against code you don't control is wasteful and adds maintenance overhead. Sometimes you may wish to use [dependency injection](#) to control a dependency and mock its behaviour for a test.

In this case though, I view converting the markdown into HTML as implementation detail of rendering, and our approval tests should give us enough confidence.

Render index

The next bit of functionality we're going to do is rendering an Index, listing the posts as a HTML ordered list.

We're expanding upon our API, so we'll put our TDD hat back on.

Write the test first

On the face of it an index page seems simple, but writing the test still prompts us to make some design choices

```
t.Run("it renders an index of posts", func(t *testing.T) {
    buf := bytes.Buffer{}
    posts := []blogrenderer.Post{{Title: "Hello World"}, {Title: "Hello World 2"}}

    if err := postRenderer.RenderIndex(&buf, posts); err != nil {
        t.Fatal(err)
    }

    got := buf.String()
    want := `<ol><li><a href="/post/hello-world">Hello World</a></li><li><a href="/post/he`
```



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

1. We're using the Post's title field as a part of the path of the URL, but we don't really want spaces in the URL so we're replacing them with hyphens.
2. We've added a RenderIndex method to our PostRenderer that again takes an io.Writer and a slice of Post.

If we had stuck with a test-after, approval tests approach here we would not be answering these questions in a controlled environment.

Tests give us space to think.

Try to run the test

```
./renderer_test.go:41:13: undefined: blogrenderer.RenderIndex
```

Write the minimal amount of code for the test to run and check the failing test output

```
func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    return nil
}
```

The above should get the following test failure

```
==== RUN TestRender
==== RUN TestRender/it_renderer_an_index_of_posts
    renderer_test.go:49: got "" want "<ol><li><a href=\"/post/hello-world\">Hello World</a></li>
--- FAIL: TestRender (0.00s)
```

Write enough code to make it pass

Even though this feels like it should be easy, it is a bit awkward. I did it in multiple steps

```
func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    indexTemplate := `<ol>{{range .}}<li><a href="/post/{{.Title}}">{{.Title}}</a></li>{{end}}
```

```
    templ, err := template.New("index").Parse(indexTemplate)
    if err != nil {
        return err
    }

    if err := templ.Execute(w, posts); err != nil {
        return err
    }

    return nil
}
```

I didn't want to bother with separate template files at first, I just wanted to get it working. I view the upfront template parsing and separation as refactoring I can do later.

This doesn't pass, but it's close.

```
==== RUN TestRender
==== RUN TestRender/it_renderer_an_index_of_posts
    renderer_test.go:49: got "<ol><li><a href=\"/post>Hello%20World\">Hello World</a></li>
--- FAIL: TestRender (0.00s)
    --- FAIL: TestRender/it_renderer_an_index_of_posts (0.00s)
```

You can see that the templating code is escaping the spaces in the href attributes. We need a way to do a string replace of spaces with

hyphens. We can't just loop through the []Post and replace them in-memory because we still want the spaces displayed to the user in the anchors.

We have a few options. The first one we'll explore is passing a function in to our template.

Passing functions into templates

```
func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    indexTemplate := `<ol>{{range .}}<li><a href="/post/{{sanitiseTitle.Title}}">{{.Title}}</a>
    templ, err := template.New("index").Funcs(template.FuncMap{
        "sanitiseTitle": func(title string) string {
            return strings.ToLower(strings.Replace(title, " ", "-", -1))
        },
    }).Parse(indexTemplate)
    if err != nil {
        return err
    }

    if err := templ.Execute(w, posts); err != nil {
        return err
    }

    return nil
}
```

Before you parse a template you can add a template.FuncMap into your template, which allows you to define functions that can be called within your template. In this case we've made a sanitiseTitle function which we then call inside our template with {{sanitiseTitle.Title}}.

This is a powerful feature, being able to send functions in to your template will allow you to do some very cool things, but, should you? Going back to the principles of Mustache and logic-less templates, why did they advocate for logic-less? **What is wrong with logic in templates?**

As we've shown, in order to test our templates, we've had to introduce a whole different kind of testing.

Imagine you introduce a function into a template which has a few different permutations of behaviour and edge cases, **how will you test it?** With this current design, your only means of testing this logic is by rendering HTML and comparing strings. This is not an easy or sane way of testing logic, and definitely not what you'd want for important business logic.

Even though the approval tests technique has reduced the cost of maintaining these tests, they're still more expensive to maintain than most unit tests you'll write. They're still sensitive to any minor markup changes you might make, it's just we've made it easier to manage. We should still strive to architect our code so we don't have to write many tests around our templates, and try and separate concerns so any logic that doesn't need to live inside our rendering code is properly separated.

What Mustache-influenced templating engines give you is a useful constraint, don't try to circumvent it too often; **don't go against the grain**. Instead, embrace the idea of [view models](#), where you construct specific types that contain the data you need to render, in a way that's convenient for the templating language.

This way, whatever important business logic you use to generate that bag of data can be unit tested separately, away from the messy world of HTML and templating.

Separating concerns

So what could we do instead?

Add a method to Post and then call that in the template We can call methods in our templating code on the types we send, so we could add a SanitisedTitle method to Post. This would simplify the template and we could easily unit test this logic separately if we wish. This is probably the easiest solution, although not necessarily the simplest.

A downside to this approach is that this is still view logic. It's not interesting to the rest of the system but it now becomes a part of the API for a core domain object. This kind of approach over time can lead to you creating [God Objects](#).

Create a dedicated view model type, such as PostViewModel with exactly the data we need Rather than our rendering code being coupled to the domain object, Post, it instead takes a view model.

```
type PostViewModel struct {
    Title, SanitisedTitle, Description, Body string
    Tags                                         []string
}
```

Callers of our code would have to map from []Post to []PostView, generating the SanitizedTitle. A way to keep this clean would be to have

a func NewPostView(p Post) PostView which would encapsulate the mapping.

This would keep our rendering code logic-less and is probably the strictest separation of concerns we could do, but the trade-off is a slightly more convoluted process to get our posts rendered.

Both options are fine, in this case I am tempted to go with the first. As you evolve the system you should be wary of adding more and more ad-hoc methods just to grease the wheels of rendering; dedicated view models become more useful when the transformation between the domain object and view becomes more involved.

So we can add our method to Post

```
func (p Post) SanitisedTitle() string {
    return strings.ToLower(strings.Replace(p.Title, " ", "-", -1))
}
```

And then we can go back to a simpler world in our rendering code

```
func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    indexTemplate := `<ol>{{range .}}<li><a href="/post/{{.SanitisedTitle}}">{{.Title}}</a>
    templ, err := template.New("index").Parse(indexTemplate)
    if err != nil {
        return err
    }

    if err := templ.Execute(w, posts); err != nil {
        return err
    }

    return nil
}
```

Refactor

Finally the test should be passing. We can now move our template into a file (templates/index.gohtml) and load it once, when we construct our renderer.

```
package blogrenderer

import (
    "embed"
    "html/template"
    "io"
)
```

```
var (
    //go:embed "templates/*"
    postTemplates embed.FS
)

type PostRenderer struct {
    templ *template.Template
}

func NewPostRenderer() (*PostRenderer, error) {
    templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
    if err != nil {
        return nil, err
    }

    return &PostRenderer{templ: templ}, nil
}

func (r *PostRenderer) Render(w io.Writer, p Post) error {
    return r.templ.ExecuteTemplate(w, "blog.gohtml", p)
}

func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    return r.templ.ExecuteTemplate(w, "index.gohtml", posts)
}
```

By parsing more than one template into templ we now have to call ExecuteTemplate and specify which template we wish to render as appropriate, but hopefully you'll agree the code we've arrived at looks great.

There is a slight risk if someone renames one of the template files, it would introduce a bug, but our fast to run unit tests would catch this quickly.

Now we're happy with our package's API design and got some basic behaviour driven out with TDD, let's change our test to use approvals.

```
t.Run("it renders an index of posts", func(t *testing.T) {
    buf := bytes.Buffer{}
    posts := []blogrenderer.Post{{Title: "Hello World"}, {Title: "Hello World 2"}}

    if err := postRenderer.RenderIndex(&buf, posts); err != nil {
        t.Fatal(err)
    }

    approvals.VerifyString(t, buf.String())
```

```
})
```

Remember to run the test to see it fail, and then approve the change.

Finally we can add our page furniture to our index page:

```
{ {{template "top" .}}
<ol>{{range .}}<li><a href="/post/{{.SanitisedTitle}}">{{.Title}}</a></li>{{end}}</ol>
{{template "bottom" .}}
```

Re-run the test, approve the change and we're done with the index!

Rendering the markdown body

I encouraged you to try it yourself, here's the approach I ended up taking.

```
package blogrenderer

import (
    "embed"
    "github.com/gomarkdown/markdown"
    "github.com/gomarkdown/markdown/parser"
    "html/template"
    "io"
)

var (
    //go:embed "templates/*"
    postTemplates embed.FS
)

type PostRenderer struct {
    templ   *template.Template
    mdParser *parser.Parser
}

func NewPostRenderer() (*PostRenderer, error) {
    templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
    if err != nil {
        return nil, err
    }

    extensions := parser.CommonExtensions | parser.AutoHeadingIDs
    parser := parser.NewWithExtensions(extensions)

    return &PostRenderer{templ: templ, mdParser: parser}, nil
}
```

```
func (r *PostRenderer) Render(w io.Writer, p Post) error {
    return r.templ.ExecuteTemplate(w, "blog.gohtml", newPostVM(p, r))
}

func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
    return r.templ.ExecuteTemplate(w, "index.gohtml", posts)
}

type postViewModel struct {
    Post
    HTMLBody template.HTML
}

func newPostVM(p Post, r *PostRenderer) postViewModel {
    vm := postViewModel{Post: p}
    vm.HTMLBody = template.HTML(markdown.ToHTML([]byte(p.Body), r.mdParser, nil))
    return vm
}
```

I used the excellent [gomarkdown](#) library which worked exactly how I'd hope.

If you tried to do this yourself you may have found that your body render had the HTML escaped. This is a security feature of Go's html/template package to stop malicious 3rd-party HTML being outputted.

To circumvent this, in the type you send to the render, you'll need to wrap your trusted HTML in [template.HTML](#)

HTML encapsulates a known safe HTML document fragment. It should not be used for HTML from a third-party, or HTML with unclosed tags or comments. The outputs of a sound HTML sanitiser and a template escaped by this package are fine for use with HTML.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

So I created an **unexported** view model (postViewModel), because I still viewed this as internal implementation detail to rendering. I have no need to test this separately and I don't want it polluting my API.

I construct one when rendering so I can parse the Body into HTMLBody and then I use that field in the template to render the HTML.

Wrapping up

If you combine your learnings of the [reading files](#) chapter and this one, you can comfortably make a well-tested, simple, static site generator and spin up a blog of your own. Find some CSS tutorials and you can make it look nice too.

This approach extends beyond blogs. Taking data from any source, be it a database, an API or a file-system and converting it into HTML and returning it from a server is a simple technique spanning many decades. People like to bemoan the complexity of modern web development but are you sure you're not just inflicting the complexity on yourself?

Go is wonderful for web development, especially when you think clearly about what your real requirements are for the website you're making. Generating HTML on the server is often a better, simpler and more performant approach than creating a "web application" with technologies like React.

What we've learned

- How to create and render HTML templates.
- How to compose templates together and [DRY](#) up related markup and help us keep a consistent look and feel.
- How to pass functions into templates, and why you should think twice about it.
- How to write "Approval Tests", which help us test the big ugly output of things like template renderers.

On logic-less templates

As always, this is all about **separation of concerns**. It's important we consider what the responsibilities are of the various parts of our system. Too often people leak important business logic into templates, mixing up concerns and making systems difficult to understand, maintain and test.

Not just for HTML

Remember that go has `text/template` to generate other kinds of data from a template. If you find yourself needing to transform data into some kind of structured output, the techniques laid out in this chapter can be useful.

References and further material

- [John Calhoun's 'Learn Web Development with Go'](#) has a number of excellent articles on templating.
- [Hotwire](#) - You can use these techniques to create Hotwire web applications. It has been built by Basecamp who are primarily a Ruby on Rails shop, but because it is server-side, we can use it with Go.

Generics

[You can find all the code for this chapter here](#)

This chapter will give you an introduction to generics, dispel reservations you may have about them, and give you an idea how to simplify some of your code in the future. After reading this you'll know how to write:

- A function that takes generic arguments
- A generic data-structure

Our own test helpers (`AssertEqual`, `AssertNotEqual`)

To explore generics we'll write some test helpers.

Assert on integers

Let's start with something basic and iterate toward our goal

```
import "testing"

func TestAssertFunctions(t *testing.T) {
    t.Run("asserting on integers", func(t *testing.T) {
        AssertEqual(t, 1, 1)
        AssertNotEqual(t, 1, 2)
    })
}

func AssertEqual(t *testing.T, got, want int) {
    t.Helper()
    if got != want {
        t.Errorf("got %d, want %d", got, want)
    }
}

func AssertNotEqual(t *testing.T, got, want int) {
```

```
t.Helper()
if got == want {
    t.Errorf("didn't want %d", got)
}
}
```

Assert on strings

Being able to assert on the equality of integers is great but what if we want to assert on string ?

```
t.Run("asserting on strings", func(t *testing.T) {
    AssertEqual(t, "hello", "hello")
    AssertNotEqual(t, "hello", "Grace")
})
```

You'll get an error

```
# github.com/quii/learn-go-with-tests/generics [github.com/quii/learn-go-with-tests/generics.test]
./generics_test.go:12:18: cannot use "hello" (untyped string constant) as int value in argument to generics.Foo.Bar(int)
./generics_test.go:13:21: cannot use "hello" (untyped string constant) as int value in argument to generics.Foo.Bar(int)
./generics_test.go:13:30: cannot use "Grace" (untyped string constant) as int value in argument to generics.Foo.Bar(int)
```

If you take your time to read the error, you'll see the compiler is complaining that we're trying to pass a string to a function that expects an integer.

Recap on type-safety If you've read the previous chapters of this book, or have experience with statically typed languages, this should not surprise you. The Go compiler expects you to write your functions, structs e.t.c. by describing what types you wish to work with.

You can't pass a string to a function that expects an integer.

Whilst this can feel like ceremony, it can be extremely helpful. By describing these constraints,

- Make function implementation simpler. By describing to the compiler what types you work with, you **constrain the number of possible valid implementations**. You can't "add" a Person and a BankAccount. You can't capitalise an integer. In software, constraints are often extremely helpful.
- Are prevented from accidentally passing data to a function you didn't mean to.

Go offers you a way to be more abstract with your types with [interfaces](#), so that you can design functions that do not take concrete types but instead, types that offer the behaviour you need. This gives you some flexibility whilst maintaining type-safety.

A function that takes a string or an integer? (or indeed, other things)

Another option Go has to make your functions more flexible is by declaring the type of your argument as `interface{}` which means "anything".

Try changing the signatures to use this type instead.

```
func AssertEqual(got, want interface{})
```

```
func AssertNotEqual(got, want interface{})
```

The tests should now compile and pass. If you try making them fail you'll see the output is a bit ropey because we're using the integer `%d` format string to print our messages, so change them to the general `%+v` format for a better output of any kind of value.

The problem with `interface{}`

Our `AssertX` functions are quite naive but conceptually aren't too different to how other [popular libraries offer this functionality](#)

```
func (is *I) Equal(a, b interface{})
```

So what's the problem?

By using `interface{}` the compiler can't help us when writing our code, because we're not telling it anything useful about the types of things passed to the function. Try comparing two different types.

```
AssertEqual(1, "1")
```

In this case, we get away with it; the test compiles, and it fails as we'd hope, although the error message `got 1, want 1` is unclear; but do we want to be able to compare strings with integers? What about comparing a Person with an Airport?

Writing functions that take `interface{}` can be extremely challenging and bug-prone because we've lost our constraints, and we have no information at compile time as to what kinds of data we're dealing with.

This means **the compiler can't help us** and we're instead more likely to have **runtime errors** which could affect our users, cause outages, or worse.

Often developers have to use reflection to implement these ahem generic functions, which can get complicated to read and write, and can hurt the performance of your program.

Our own test helpers with generics

Ideally, we don't want to have to make specific AssertX functions for every type we ever deal with. We'd like to be able to have one AssertEqual function that works with any type but does not let you compare [apples and oranges](#).

Generics offer us a way to make abstractions (like interfaces) by letting us **describe our constraints**. They allow us to write functions that have a similar level of flexibility that interface{} offers but retain type-safety and provide a better developer experience for callers.

```
func TestAssertFunctions(t *testing.T) {
    t.Run("asserting on integers", func(t *testing.T) {
        AssertEqual(t, 1, 1)
        AssertNotEqual(t, 1, 2)
    })

    t.Run("asserting on strings", func(t *testing.T) {
        AssertEqual(t, "hello", "hello")
        AssertNotEqual(t, "hello", "Grace")
    })

    // AssertEqual(t, 1, "1") // uncomment to see the error
}

func AssertEqual[T comparable](t *testing.T, got, want T) {
    t.Helper()
    if got != want {
        t.Errorf("got %v, want %v", got, want)
    }
}

func AssertNotEqual[T comparable](t *testing.T, got, want T) {
    t.Helper()
    if got == want {
        t.Errorf("didn't want %v", got)
    }
}
```

To write generic functions in Go, you need to provide "type parameters" which is just a fancy way of saying "describe your generic type and give it a label".

In our case the type of our type parameter is comparable and we've given it the label of T. This label then lets us describe the types for the arguments to our function (got, want T).

We're using comparable because we want to describe to the compiler that we wish to use the == and != operators on things of type T in our function, we want to compare! If you try changing the type to any,

```
func AssertNotEqual[T any](got, want T)
```

You'll get the following error:

```
prog.go2:15:5: cannot compare got != want (operator != not defined for T)
```

Which makes a lot of sense, because you can't use those operators on every (or any) type.

Is a generic function with **T any** the same as **interface{}** ?

Consider two functions

```
func GenericFoo[T any](x, y T)
```

```
func InterfaceyFoo(x, y interface{})
```

What's the point of generics here? Doesn't any describe... anything?

In terms of constraints, any does mean "anything" and so does interface{}. In fact, any was added in 1.18 and is just an alias for interface{}.

The difference with the generic version is you're still describing a specific type and what that means is we've still constrained this function to only work with one type.

What this means is you can call InterfaceyFoo with any combination of types (e.g InterfaceyFoo(apple, orange)). However GenericFoo still offers some constraints because we've said that it only works with one type, T.

Valid:

- GenericFoo(apple1, apple2)
- GenericFoo(orange1, orange2)
- GenericFoo(1, 2)
- GenericFoo("one", "two")

Not valid (fails compilation):

- GenericFoo(apple1, orange1)
- GenericFoo("1", 1)

If your function returns the generic type, the caller can also use the type as it was, rather than having to make a type assertion because when a function returns interface{} the compiler cannot make any guarantees about the type.

Next: Generic data types

We're going to create a `stack` data type. Stacks should be fairly straightforward to understand from a requirements point of view. They're a collection of items where you can Push items to the "top" and to get items back again you Pop items from the top (LIFO - last in, first out).

For the sake of brevity I've omitted the TDD process that arrived me at the following code for a stack of ints, and a stack of strings.

```
type StackOfInts struct {
    values []int
}

func (s *StackOfInts) Push(value int) {
    s.values = append(s.values, value)
}

func (s *StackOfInts) IsEmpty() bool {
    return len(s.values) == 0
}

func (s *StackOfInts) Pop() (int, bool) {
    if s.IsEmpty() {
        return 0, false
    }

    index := len(s.values) - 1
    el := s.values[index]
    s.values = s.values[:index]
    return el, true
}

type StackOfStrings struct {
    values []string
}

func (s *StackOfStrings) Push(value string) {
    s.values = append(s.values, value)
}

func (s *StackOfStrings) IsEmpty() bool {
    return len(s.values) == 0
}

func (s *StackOfStrings) Pop() (string, bool) {
```

```
if s.IsEmpty() {
    return "", false
}

index := len(s.values) - 1
el := s.values[index]
s.values = s.values[:index]
return el, true
}
```

I've created a couple of other assertion functions to help out

```
func AssertTrue(t *testing.T, got bool) {
    t.Helper()
    if !got {
        t.Errorf("got %v, want true", got)
    }
}
```

```
func AssertFalse(t *testing.T, got bool) {
    t.Helper()
    if got {
        t.Errorf("got %v, want false", got)
    }
}
```

And here's the tests

```
func TestStack(t *testing.T) {
    t.Run("integer stack", func(t *testing.T) {
        myStackOfInts := new(StackOfInts)

        // check stack is empty
        AssertTrue(t, myStackOfInts.IsEmpty())

        // add a thing, then check it's not empty
        myStackOfInts.Push(123)
        AssertFalse(t, myStackOfInts.IsEmpty())

        // add another thing, pop it back again
        myStackOfInts.Push(456)
        value, _ := myStackOfInts.Pop()
        AssertEqual(t, value, 456)
        value, _ = myStackOfInts.Pop()
        AssertEqual(t, value, 123)
        AssertTrue(t, myStackOfInts.IsEmpty())
    })
}
```

```
t.Run("string stack", func(t *testing.T) {
    myStackOfStrings := new(StackOfStrings)

    // check stack is empty
    AssertTrue(t, myStackOfStrings.IsEmpty())

    // add a thing, then check it's not empty
    myStackOfStrings.Push("123")
    AssertFalse(t, myStackOfStrings.IsEmpty())

    // add another thing, pop it back again
    myStackOfStrings.Push("456")
    value, _ := myStackOfStrings.Pop()
    AssertEqual(t, value, "456")
    value, _ = myStackOfStrings.Pop()
    AssertEqual(t, value, "123")
    AssertTrue(t, myStackOfStrings.IsEmpty())
)
}
```

Problems

- The code for both StackOfStrings and StackOfInts is almost identical. Whilst duplication isn't always the end of the world, it's more code to read, write and maintain.
- As we're duplicating the logic across two types, we've had to duplicate the tests too.

We really want to capture the idea of a stack in one type, and have one set of tests for them. We should be wearing our refactoring hat right now which means we should not be changing the tests because we want to maintain the same behaviour.

Without generics, this is what we could do

```
type StackOfInts = Stack
type StackOfStrings = Stack

type Stack struct {
    values []interface{}
}

func (s *Stack) Push(value interface{}) {
    s.values = append(s.values, value)
}

func (s *Stack) IsEmpty() bool {
```

```
    return len(s.values) == 0
}

func (s *Stack) Pop() (interface{}, bool) {
    if s.IsEmpty() {
        var zero interface{}
        return zero, false
    }

    index := len(s.values) - 1
    el := s.values[index]
    s.values = s.values[:index]
    return el, true
}
```

- We're aliasing our previous implementations of StackOfInts and StackOfStrings to a new unified type Stack
- We've removed the type safety from the Stack by making it so values is a slice of interface{}

To try this code, you'll have to remove the type constraints from our assert functions:

```
func AssertEqual(t *testing.T, got, want interface{})
```

If you do this, our tests still pass. Who needs generics?

The problem with throwing out type safety

The first problem is the same as we saw with our AssertEquals - we've lost type safety. I can now Push apples onto a stack of oranges.

Even if we have the discipline not to do this, the code is still unpleasant to work with because when methods **return interface{} they are horrible to work with.**

Add the following test,

```
t.Run("interface stack DX is horrid", func(t *testing.T) {
    myStackOfInts := new(StackOfInts)

    myStackOfInts.Push(1)
    myStackOfInts.Push(2)
    firstNum, _ := myStackOfInts.Pop()
    secondNum, _ := myStackOfInts.Pop()
    AssertEqual(t, firstNum+secondNum, 3)
})
```

You get a compiler error, showing the weakness of losing type-safety:

invalid operation: operator + not defined on firstNum (variable of type interface{})

When Pop returns interface{} it means the compiler has no information about what the data is and therefore severely limits what we can do. It can't know that it should be an integer, so it does not let us use the + operator.

To get around this, the caller has to do a [type assertion](#) for each value.

```
t.Run("interface stack dx is horrid", func(t *testing.T) {
    myStackOfInts := new(StackOfInts)

    myStackOfInts.Push(1)
    myStackOfInts.Push(2)
    firstNum, _ := myStackOfInts.Pop()
    secondNum, _ := myStackOfInts.Pop()

    // get our ints from out interface{}
    reallyFirstNum, ok := firstNum.(int)
    AssertTrue(t, ok) // need to check we definitely got an int out of the interface{}

    reallySecondNum, ok := secondNum.(int)
    AssertTrue(t, ok) // and again!

    AssertEqual(t, reallyFirstNum+reallySecondNum, 3)
})
```

The unpleasantness radiating from this test would be repeated for every potential user of our Stack implementation, yuck.

Generic data structures to the rescue

Just like you can define generic arguments to functions, you can define generic data structures.

Here's our new Stack implementation, featuring a generic data type.

```
type Stack[T any] struct {
    values []T
}

func (s *Stack[T]) Push(value T) {
    s.values = append(s.values, value)
}

func (s *Stack[T]) IsEmpty() bool {
    return len(s.values) == 0
}
```

```
func (s *Stack[T]) Pop() (T, bool) {
    if s.IsEmpty() {
        var zero T
        return zero, false
    }

    index := len(s.values) - 1
    el := s.values[index]
    s.values = s.values[:index]
    return el, true
}
```

Here's the tests, showing them working how we'd like them to work, with full type-safety.

```
func TestStack(t *testing.T) {
    t.Run("integer stack", func(t *testing.T) {
        myStackOfInts := new(Stack[int])

        // check stack is empty
        AssertTrue(t, myStackOfInts.IsEmpty())

        // add a thing, then check it's not empty
        myStackOfInts.Push(123)
        AssertFalse(t, myStackOfInts.IsEmpty())

        // add another thing, pop it back again
        myStackOfInts.Push(456)
        value, _ := myStackOfInts.Pop()
        AssertEqual(t, value, 456)
        value, _ = myStackOfInts.Pop()
        AssertEqual(t, value, 123)
        AssertTrue(t, myStackOfInts.IsEmpty())

        // can get the numbers we put in as numbers, not untyped interface{}
        myStackOfInts.Push(1)
        myStackOfInts.Push(2)
        firstNum, _ := myStackOfInts.Pop()
        secondNum, _ := myStackOfInts.Pop()
        AssertEqual(t, firstNum+secondNum, 3)
    })
}
```

You'll notice the syntax for defining generic data structures is consistent with defining generic arguments to functions.

```
type Stack[T any] struct {
```

```
    values []T
}
```

It's almost the same as before, it's just that what we're saying is the **type of the stack constrains what type of values you can work with**.

Once you create a Stack[Orange] or a Stack[Apple] the methods defined on our stack will only let you pass in and will only return the particular type of the stack you're working with:

```
func (s *Stack[T]) Pop() (T, bool)
```

You can imagine the types of implementation being somehow generated for you, depending on what type of stack you create:

```
func (s *Stack[Orange]) Pop() (Orange, bool)
```

```
func (s *Stack[Apple]) Pop() (Apple, bool)
```

Now that we have done this refactoring, we can safely remove the string stack test because we don't need to prove the same logic over and over.

Using a generic data type we have:

- Reduced duplication of important logic.
- Made Pop return T so that if we create a Stack[int] we in practice get back int from Pop; we can now use + without the need for type assertion gymnastics.
- Prevented misuse at compile time. You cannot Push oranges to an apple stack.

Wrapping up

This chapter should have given you a taste of generics syntax, and some ideas as to why generics might be helpful. We've written our own Assert functions which we can safely re-use to experiment with other ideas around generics, and we've implemented a simple data structure to store any type of data we wish, in a type-safe manner.

Generics are simpler than using interface{} in most cases

If you're inexperienced with statically-typed languages, the point of generics may not be immediately obvious, but I hope the examples in this chapter have illustrated where the Go language isn't as expressive as we'd like. In particular using interface{} makes your code:

- Less safe (mix apples and oranges), requires more error handling
- Less expressive, interface{} tells you nothing about the data

-
- More likely to rely on [reflection](#), type-assertions etc which makes your code more difficult to work with and more error prone as it pushes checks from compile-time to runtime

Using statically typed languages is an act of describing constraints. If you do it well, you create code that is not only safe and simple to use but also simpler to write because the possible solution space is smaller.

Generics gives us a new way to express constraints in our code, which as demonstrated will allow us to consolidate and simplify code that was not possible until Go 1.18.

Will generics turn Go into Java?

- No.

There's a lot of [FUD \(fear, uncertainty and doubt\)](#) in the Go community about generics leading to nightmare abstractions and baffling code bases. This is usually caveatted with "they must be used carefully".

Whilst this is true, it's not especially useful advice because this is true of any language feature.

Not many people complain about our ability to define interfaces which, like generics is a way of describing constraints within our code. When you describe an interface you are making a design choice that could be poor, generics are not unique in their ability to make confusing, annoying to use code.

You're already using generics

When you consider that if you've used arrays, slices or maps; you've already been a consumer of generic code.

```
var myApples []Apples
// You cant do this!
append(myApples, Orange{})
```

Abstraction is not a dirty word

It's easy to dunk on [AbstractSingletonProxyFactoryBean](#) but let's not pretend a code base with no abstraction at all isn't also bad. It's your job to gather related concepts when appropriate, so your system is easier to understand and change; rather than being a collection of disparate functions and types with a lack of clarity.

Make it work, make it right, make it fast

People run in to problems with generics when they're abstracting too quickly without enough information to make good design decisions.

The TDD cycle of red, green, refactor means that you have more guidance as to what code you actually need to deliver your behaviour, **rather than imagining abstractions up front**; but you still need to be careful.

There's no hard and fast rules here but resist making things generic until you can see that you have a useful generalisation. When we created the various Stack implementations we importantly started with concrete behaviour like StackOfStrings and StackOfInts backed by tests. From our real code we could start to see real patterns, and backed by our tests, we could explore refactoring toward a more general-purpose solution.

People often advise you to only generalise when you see the same code three times, which seems like a good starting rule of thumb.

A common path I've taken in other programming languages has been:

- One TDD cycle to drive some behaviour
- Another TDD cycle to exercise some other related scenarios

Hmm, these things look similar - but a little duplication is better than coupling to a bad abstraction

- Sleep on it
- Another TDD cycle

OK, I'd like to try to see if I can generalise this thing. Thank goodness I am so smart and good-looking because I use TDD, so I can refactor whenever I wish, and the process has helped me understand what behaviour I actually need before designing too much.

- This abstraction feels nice! The tests are still passing, and the code is simpler
- I can now delete a number of tests, I've captured the essence of the behaviour and removed unnecessary detail

Revisiting arrays and slices with generics

The code for this chapter is a continuation from Arrays and Slices, found [here](#)

Take a look at both SumAll and SumAllTails that we wrote in [arrays and slices](#). If you don't have your version please copy the code from

the [arrays and slices](#) chapter along with the tests.

```
// Sum calculates the total from a slice of numbers.
func Sum(numbers []int) int {
    var sum int
    for _, number := range numbers {
        sum += number
    }
    return sum
}

// SumAllTails calculates the sums of all but the first number given a collection of slices.
func SumAllTails(numbersToSum ...[]int) []int {
    var sums []int
    for _, numbers := range numbersToSum {
        if len(numbers) == 0 {
            sums = append(sums, 0)
        } else {
            tail := numbers[1:]
            sums = append(sums, Sum(tail))
        }
    }

    return sums
}
```

Do you see a recurring pattern?

- Create some kind of "initial" result value.
- Iterate over the collection, applying some kind of operation (or function) to the result and the next item in the slice, setting a new value for the result
- Return the result.

This idea is commonly talked about in functional programming circles, often times called '[reduce](#)' or [fold](#).

In functional programming, fold (also termed reduce, accumulate, aggregate, compress, or inject) refers to a family of higher-order functions that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value. Typically, a fold is presented with a combining function, a top node of a data structure, and possibly some default values to be used under certain conditions. The fold then proceeds to combine elements of the data structure's hierarchy, using the function in a systematic way.

Go has always had higher-order functions, and as of version 1.18 it also has [generics](#), so it is now possible to define some of these functions discussed in our wider field. There's no point burying your head in the sand, this is a very common abstraction outside the Go ecosystem and it'll be beneficial to understand it.

Now, I know some of you are probably cringing at this.

Go is supposed to be simple

Don't conflate easiness, with simplicity. Doing loops and copy-pasting code is easy, but it's not necessarily simple. For more on simple vs easy, watch [Rich Hickey's masterpiece of a talk - Simple Made Easy](#).

Don't conflate unfamiliarity, with complexity. Fold/reduce may initially sound scary and computer-sciencey but all it really is, is an abstraction over a very common operation. Taking a collection, and combining it into one item. When you step back, you'll realise you probably do this a lot.

A generic refactor

A mistake people often make with shiny new language features is they start by using them without having a concrete use-case. They rely on conjecture and guesswork to guide their efforts.

Thankfully we've written our "useful" functions and have tests around them, so now we are free to experiment with ideas in the refactoring stage of TDD and know that whatever we're trying, has a verification of its value via our unit tests.

Using generics as a tool for simplifying code via the refactoring step is far more likely to guide you to useful improvements, rather than premature abstractions.

We are safe to try things out, re-run our tests, if we like the change we can commit. If not, just revert the change. This freedom to experiment is one of the truly huge values of TDD.

You should be familiar with the generics syntax [from the previous chapter](#), try and write your own Reduce function and use it inside Sum and SumAllTails.

Hints

If you think about the arguments to your function first, it'll give you a very small set of valid solutions

- The array you want to reduce

-
- Some kind of combining function, or accumulator

"Reduce" is an incredibly well documented pattern, there's no need to re-invent the wheel. [Read the wiki, in particular the lists section](#), it should prompt you for another argument you'll need.

In practice, it is convenient and natural to have an initial value

My first-pass of Reduce

```
func Reduce[A any](collection []A, accumulator func(A, A) A, initialValue A) A {
    var result = initialValue
    for _, x := range collection {
        result = accumulator(result, x)
    }
    return result
}
```

Reduce captures the essence of the pattern, it's a function that takes a collection, an accumulating function, an initial value, and returns a single value. There's no messy distractions around concrete types.

If you understand generics syntax, you should have no problem understanding what this function does. By using the recognised term Reduce, programmers from other languages understand the intent too.

The usage

```
// Sum calculates the total from a slice of numbers.
func Sum(numbers []int) int {
    add := func(acc, x int) int { return acc + x }
    return Reduce(numbers, add, 0)
}

// SumAllTails calculates the sums of all but the first number given a collection of slices.
func SumAllTails(numbers ...[]int) []int {
    sumTail := func(acc, x []int) []int {
        if len(x) == 0 {
            return append(acc, 0)
        } else {
            tail := x[1:]
            return append(acc, Sum(tail))
        }
    }
}
```

```
    return Reduce(numbers, sumTail, []int{})  
}
```

Sum and SumAllTails now describe the behaviour of their computations as the functions declared on their first lines respectively. The act of running the computation on the collection is abstracted away in Reduce.

Further applications of reduce

Using tests we can play around with our reduce function to see how re-usable it is. I have copied over our generic assertion functions from the previous chapter.

```
func TestReduce(t *testing.T) {  
    t.Run("multiplication of all elements", func(t *testing.T) {  
        multiply := func(x, y int) int {  
            return x * y  
        }  
  
        AssertEqual(t, Reduce([]int{1, 2, 3}, multiply, 1), 6)  
    })  
  
    t.Run("concatenate strings", func(t *testing.T) {  
        concatenate := func(x, y string) string {  
            return x + y  
        }  
  
        AssertEqual(t, Reduce([]string{"a", "b", "c"}, concatenate, ""), "abc")  
    })  
}
```

The zero value

In the multiplication example, we show the reason for having a default value as an argument to Reduce. If we relied on Go's default value of 0 for int, we'd multiply our initial value by 0, and then the following ones, so you'd only ever get 0. By setting it to 1, the first element in the slice will stay the same, and the rest will multiply by the next elements.

If you wish to sound clever with your nerd friends, you'd call this [The Identity Element](#).

In mathematics, an identity element, or neutral element, of a binary operation operating on a set is an element of the

set which leaves unchanged every element of the set when the operation is applied.

In addition, the identity element is 0.

$$1 + 0 = 1$$

With multiplication, it is 1.

$$1 * 1 = 1$$

What if we wish to reduce into a different type from A?

Suppose we had a list of transactions Transaction and we wanted a function that would take them, plus a name to figure out their bank balance.

Let's follow the TDD process.

Write the test first

```
func TestBadBank(t *testing.T) {
    transactions := []Transaction{
        {
            From: "Chris",
            To:   "Riya",
            Sum:  100,
        },
        {
            From: "Adil",
            To:   "Chris",
            Sum:  25,
        },
    }

    AssertEqual(t, BalanceFor(transactions, "Riya"), 100)
    AssertEqual(t, BalanceFor(transactions, "Chris"), -75)
    AssertEqual(t, BalanceFor(transactions, "Adil"), -25)
}
```

Try to run the test

```
# github.com/quii/learn-go-with-tests/arrays/v8 [github.com/quii/learn-go-with-tests/arrays/v8.t
./bad_bank_test.go:6:20: undefined: Transaction
./bad_bank_test.go:18:14: undefined: BalanceFor
```

Write the minimal amount of code for the test to run and check the failing test output

We don't have our types or functions yet, add them to make the test run.

```
type Transaction struct {
    From string
    To   string
    Sum  float64
}

func BalanceFor(transactions []Transaction, name string) float64 {
    return 0.0
}
```

When you run the test you should see the following:

```
==== RUN TestBadBank
    bad_bank_test.go:19: got 0, want 100
    bad_bank_test.go:20: got 0, want -75
    bad_bank_test.go:21: got 0, want -25
--- FAIL: TestBadBank (0.00s)
```

Write enough code to make it pass

Let's write the code as if we didn't have a Reduce function first.

```
func BalanceFor(transactions []Transaction, name string) float64 {
    var balance float64
    for _, t := range transactions {
        if t.From == name {
            balance -= t.Sum
        }
        if t.To == name {
            balance += t.Sum
        }
    }
    return balance
}
```

Refactor

At this point, have some source control discipline and commit your work. We have working software, ready to challenge Monzo, Barclays, et al.

Now our work is committed, we are free to play around with it, and try some different ideas out in the refactoring phase. To be fair, the code we have isn't exactly bad, but for the sake of this exercise, I want to demonstrate the same code using Reduce.

```
func BalanceFor(transactions []Transaction, name string) float64 {
    adjustBalance := func(currentBalance float64, t Transaction) float64 {
        if t.From == name {
            return currentBalance - t.Sum
        }
        if t.To == name {
            return currentBalance + t.Sum
        }
        return currentBalance
    }
    return Reduce(transactions, adjustBalance, 0.0)
}
```

But this won't compile.

```
./bad_bank.go:19:35: type func(acc float64, t Transaction) float64 of adjustBalance does not match
```

The reason is we're trying to reduce to a different type than the type of the collection. This sounds scary, but actually just requires us to adjust the type signature of Reduce to make it work. We won't have to change the function body, and we won't have to change any of our existing callers.

```
func Reduce[A, B any](collection []A, accumulator func(B, A) B, initialValue B) B {
    var result = initialValue
    for _, x := range collection {
        result = accumulator(result, x)
    }
    return result
}
```

We've added a second type constraint which has allowed us to loosen the constraints on Reduce. This allows people to Reduce from a collection of A into a B. In our case from Transaction to float64.

This makes Reduce more general-purpose and reusable, and still type-safe. If you try and run the tests again they should compile, and pass.

Extending the bank

For fun, I wanted to improve the ergonomics of the bank code. I've omitted the TDD process for brevity.

```
func TestBadBank(t *testing.T) {
    var (
        riya = Account{Name: "Riya", Balance: 100}
        chris = Account{Name: "Chris", Balance: 75}
        adil = Account{Name: "Adil", Balance: 200}

        transactions = []Transaction{
            NewTransaction(chris, riya, 100),
            NewTransaction(adil, chris, 25),
        }
    )

    newBalanceFor := func(account Account) float64 {
        return NewBalanceFor(account, transactions).Balance
    }

    AssertEqual(t, newBalanceFor(riya), 200)
    AssertEqual(t, newBalanceFor(chris), 0)
    AssertEqual(t, newBalanceFor(adil), 175)
}
```

And here's the updated code

```
package main

type Transaction struct {
    From string
    To   string
    Sum  float64
}

func NewTransaction(from, to Account, sum float64) Transaction {
    return Transaction{From: from.Name, To: to.Name, Sum: sum}
}

type Account struct {
    Name string
    Balance float64
}

func NewBalanceFor(account Account, transactions []Transaction) Account {
    return Reduce(
        transactions,
        applyTransaction,
        account,
    )
}
```

```
func applyTransaction(a Account, transaction Transaction) Account {
    if transaction.From == a.Name {
        a.Balance -= transaction.Sum
    }
    if transaction.To == a.Name {
        a.Balance += transaction.Sum
    }
    return a
}
```

I feel this really shows the power of using concepts like Reduce. The NewBalanceFor feels more declarative, describing what happens, rather than how. Often when we're reading code, we're darting through lots of files, and we're trying to understand what is happening, rather than how, and this style of code facilitates this well.

If I wish to dig in to the detail I can, and I can see the business logic of applyTransaction without worrying about loops and mutating state; Reduce takes care of that separately.

Fold/reduce are pretty universal

The possibilities are endless™ with Reduce (or Fold). It's a common pattern for a reason, it's not just for arithmetic or string concatenation. Try a few other applications.

- Why not mix some color.RGBA into a single colour?
- Total up the number of votes in a poll, or items in a shopping basket.
- More or less anything involving processing a list.

Find

Now that Go has generics, combining them with higher-order functions, we can reduce a lot of boilerplate code within our projects, to help our systems be easier to understand and manage.

No longer do you need to write specific Find functions for each type of collection you want to search, instead re-use or write a Find function. If you understood the Reduce function above, writing a Find function will be trivial.

Here's a test

```
func TestFind(t *testing.T) {
    t.Run("find first even number", func(t *testing.T) {
```

```
numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

firstEvenNumber, found := Find(numbers, func(x int) bool {
    return x%2 == 0
})
AssertTrue(t, found)
AssertEqual(t, firstEvenNumber, 2)
})
```

And here's the implementation

```
func Find[A any](items []A, predicate func(A) bool) (value A, found bool) {
    for _, v := range items {
        if predicate(v) {
            return v, true
        }
    }
    return
}
```

Again, because it takes a generic type, we can re-use it in many ways

```
type Person struct {
    Name string
}

t.Run("Find the best programmer", func(t *testing.T) {
    people := []Person{
        Person{Name: "Kent Beck"},
        Person{Name: "Martin Fowler"},
        Person{Name: "Chris James"},
    }

    king, found := Find(people, func(p Person) bool {
        return strings.Contains(p.Name, "Chris")
    })

    AssertTrue(t, found)
    AssertEqual(t, king, Person{Name: "Chris James"})
})
```

As you can see, this code is flawless.

Wrapping up

When done tastefully, higher-order functions like these will make your code simpler to read and maintain, but remember the rule of thumb:

Use the TDD process to drive out real, specific behaviour that you actually need, in the refactoring stage you then might discover some useful abstractions to help tidy the code up.

Practice combining TDD with good source control habits. Commit your work when your test is passing, before trying to refactor. This way if you make a mess, you can easily get yourself back to your working state.

Names matter

Make an effort to do some research outside of Go, so you don't re-invent patterns that already exist with an already established name.

Writing a function takes a collection of A and converts them to B? Don't call it Convert, that's [Map](#). Using the "proper" name for these items will reduce the cognitive burden for others and make it more search engine friendly to learn more.

This doesn't feel idiomatic?

Try to have an open-mind.

Whilst the idioms of Go won't, and shouldn't radically change due to generics being released, the idioms will change - due to the language changing! This should not be a controversial point.

Saying

This is not idiomatic

Without any more detail, is not an actionable, or useful thing to say. Especially when discussing new language features.

Discuss with your colleagues patterns and style of code based on their merits rather than dogma. So long as you have well-designed tests, you'll always be able to refactor and shift things as you understand what works well for you, and your team.

Resources

Fold is a real fundamental in computer science. Here's some interesting resources if you wish to dig more into it

- [Wikipedia: Fold](#)
- [A tutorial on the universality and expressiveness of fold](#)

Introduction to acceptance testing

At \$WORK, we've been running into the need to have "graceful shutdown" for our services. Graceful shutdown makes sure your system finishes its work properly before it is terminated. A real-world analogy would be someone trying to wrap up a phone call properly before moving on to the next meeting, rather than just hanging up mid-sentence.

This chapter will give an intro to graceful shutdown in the context of an HTTP server, and how to write "acceptance tests" to give yourself confidence in the behaviour of your code.

After reading this you'll know how to share packages with excellent tests, reduce maintenance efforts, and increase confidence in the quality of your work.

Just enough info about Kubernetes

We run our software on [Kubernetes](#) (K8s). K8s will terminate "pods" (in practice, our software) for various reasons, and a common one is when we push new code that we want to deploy.

We are setting ourselves high standards regarding [DORA metrics](#), so we work in a way where we deploy small, incremental improvements and features to production multiple times per day.

When k8s wishes to terminate a pod, it initiates a "[termination lifecycle](#)", and a part of that is sending a SIGTERM signal to our software. This is k8s telling our code:

You need to shut yourself down, finish whatever work you're doing because after a certain "grace period", I will send SIGKILL, and it's lights out for you.

On SIGKILL any work your program might've been doing will be immediately stopped.

If you do not have grace

Depending on the nature of your software, if you ignore SIGTERM, you can run into problems.

Our specific problem was with in-flight HTTP requests. When an automated test was exercising our API, if k8s decided to stop the pod, the server would die, the test would not get a response from the server, and the test will fail.

This would trigger an alert in our incidents channel which requires a dev to stop what they're doing and address the problem. These

intermittent failures are an annoying distraction for our team.

These problems are not unique to our tests. If a user sends a request to your system and the process gets terminated mid-flight, they'll likely be greeted with a 5xx HTTP error, not the kind of user experience you want to deliver.

When you have grace

What we want to do is listen for SIGTERM, and rather than instantly killing the server, we want to:

- Stop listening to any more requests
- Allow any in-flight requests to finish
- Then terminate the process

How to have grace

Thankfully, Go already has a mechanism for gracefully shutting down a server with [net/http/Server.Shutdown](#).

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down. If the provided context expires before the shutdown is complete, Shutdown returns the context's error, otherwise it returns any error returned from closing the Server's underlying Listener(s).

To handle SIGTERM we can use [os/signal.Notify](#), which will send any incoming signals to a channel we provide.

By using these two features from the standard library, you can listen for SIGTERM and shutdown gracefully.

Graceful shutdown package

To that end, I wrote <https://pkg.go.dev/github.com/quii/go-graceful-shutdown>. It provides a decorator function for a *http.Server to call its Shutdown method when a SIGTERM signal is detected

```
func main() {
    httpServer := &http.Server{Addr: ":8080", Handler: http.HandlerFunc(acceptancetests.SlowHTTPHandler)}
    server := gracefulshutdown.NewServer(httpServer)

    if err := server.ListenAndServe(); err != nil {
        log.Fatal(err)
    }
}
```

```
// this will typically happen if our responses aren't written before the ctx deadline, not much
log.Fatalf("uh oh, didnt shutdown gracefully, some responses may have been lost %v", err)
}

// hopefully, you'll always see this instead
log.Println("shutdown gracefully! all responses were sent")
}
```

The specifics around the code are not too important for this read, but it is worth having a quick look over the code before carrying on.

Tests and feedback loops

When we wrote the gracefulshutdown package, we had unit tests to prove it behaves correctly which gave us the confidence to aggressively refactor. However, we still didn't feel "confident" that it **really** worked.

We added a cmd package and made a real program to use the package we were writing. We'd manually fire it up, fire off an HTTP request to it, and then send a SIGTERM to see what would happen.

The engineer in you should be feeling uncomfortable with manual testing. It's boring, it doesn't scale, it's inaccurate, and it's wasteful. If you're writing a package you intend to share, but also want to keep it simple and cheap to change, manual testing is not going to cut it.

Acceptance tests

If you've read the rest of this book, you will have mostly written "unit tests". Unit tests are a fantastic tool for enabling fearless refactoring, driving good modular design, preventing regressions, and facilitating fast feedback.

By their nature, they only test small parts of your system. Usually, unit tests alone are not enough for an effective testing strategy. Remember, we want our systems to **always be shippable**. We can't rely on manual testing, so we need another kind of testing: **acceptance tests**.

What are they?

Acceptance tests are a kind of "black-box test". They are sometimes referred to as "functional tests". They should exercise the system as a user of the system would.

The term “black-box” refers to the idea that the test code has no access to the internals of the system, it can only use its public interface and make assertions on the behaviours it observes. This means they can only test the system as a whole.

This is an advantageous trait because it means the tests exercise the system the same as a user would, it can’t use any special workarounds that could make a test pass, but not actually prove what you need to prove. This is similar to the principle of preferring your unit test files to live inside a separate test package, for example, package mypkg_test rather than package mypkg.

Benefits of acceptance tests

- When they pass, you know your entire system behaves how you want it to.
- They are more accurate, quicker, and require less effort than manual testing.
- When written well, they act as accurate, verified documentation of your system. It doesn’t fall into the trap of documentation that diverges from the real behaviour of the system.
- No mocking! It’s all real.

Potential drawbacks vs unit tests

- They are expensive to write.
- They take longer to run.
- They are dependent on the design of the system.
- When they fail, they typically don’t give you a root cause, and can be difficult to debug.
- They don’t give you feedback on the internal quality of your system. You could write total garbage and still make an acceptance test pass.
- Not all scenarios are practical to exercise due to the black-box nature.

For this reason, it is foolish to only rely on acceptance tests. They do not have many of the qualities unit tests have, and a system with a large number of acceptance tests will tend to suffer in terms of maintenance costs and poor lead time.

Lead time? Lead time refers to how long it takes from a commit being merged into your main branch to it being deployed in production. This number can vary from weeks and even months for some teams to a matter of minutes. Again, at \$WORK, we value DORA’s findings and want to keep our lead time to under 10 minutes.

A balanced testing approach is required for a reliable system with excellent lead time, and this is usually described in terms of the [Test Pyramid](#).

How to write basic acceptance tests

How does this relate to the original problem? We've just written a package here, and it is entirely unit-testable.

As I mentioned, the unit tests weren't quite giving us the confidence we needed. We want to be really sure the package works when integrated with a real, running program. We should be able to automate the manual checks we were making.

Let's take a look at the test program:

```
func main() {
    httpServer := &http.Server{Addr: ":8080", Handler: http.HandlerFunc(acceptancetests.SlowH
    server := gracefulshutdown.NewServer(httpServer)

    if err := server.ListenAndServe(); err != nil {
        // this will typically happen if our responses aren't written before the ctx deadline, not much
        log.Fatalf("uh oh, didnt shutdown gracefully, some responses may have been lost %v", err)
    }

    // hopefully, you'll always see this instead
    log.Println("shutdown gracefully! all responses were sent")
}
```

You may have guessed that SlowHandler has a `time.Sleep` to delay responding, so I had time to SIGTERM and see what happens. The rest is fairly boilerplate:

- Make a net/http/Server;
- Wrap it in the library (see: [Decorator pattern](#));
- Use the wrapped version to ListenAndServe.

High-level steps for the acceptance test

- Build the program
- Run it (and wait for it listen on 8080)
- Send an HTTP request to the server
- Before the server has a chance to send an HTTP response, send SIGTERM
- See if we still get a response

Building and running the program

```
package acceptancetests
```

```
import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "os/exec"
    "path/filepath"
    "syscall"
    "time"
)

const (
    baseBinName = "temp-testbinary"
)

func LaunchTestProgram(port string) (cleanup func(), sendInterrupt func() error, err error) {
    binName, err := buildBinary()
    if err != nil {
        return nil, nil, err
    }

    sendInterrupt, kill, err := runServer(binName, port)

    cleanup = func() {
        if kill != nil {
            kill()
        }
        os.Remove(binName)
    }

    if err != nil {
        cleanup() // even though it's not listening correctly, the program could still be running
        return nil, nil, err
    }

    return cleanup, sendInterrupt, nil
}

func buildBinary() (string, error) {
    binName := randomString(10) + "-" + baseBinName

    build := exec.Command("go", "build", "-o", binName)
```

```
if err := build.Run(); err != nil {
    return "", fmt.Errorf("cannot build tool %s: %s", binName, err)
}
return binName, nil
}

func runServer(binName string, port string) (sendInterrupt func() error, kill func(), err error) {
    dir, err := os.Getwd()
    if err != nil {
        return nil, nil, err
    }

    cmdPath := filepath.Join(dir, binName)

    cmd := exec.Command(cmdPath)

    if err := cmd.Start(); err != nil {
        return nil, nil, fmt.Errorf("cannot run temp converter: %s", err)
    }

    kill = func() {
        _ = cmd.Process.Kill()
    }

    sendInterrupt = func() error {
        return cmd.Process.Signal(syscall.SIGTERM)
    }

    err = waitForServerListening(port)

    return
}

func waitForServerListening(port string) error {
    for i := 0; i < 30; i++ {
        conn, _ := net.Dial("tcp", net.JoinHostPort("localhost", port))
        if conn != nil {
            conn.Close()
            return nil
        }
        time.Sleep(100 * time.Millisecond)
    }
    return fmt.Errorf("nothing seems to be listening on localhost:%s", port)
}
```

```
func randomString(n int) string {
    var letters = []rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789")
    s := make([]rune, n)
    for i := range s {
        s[i] = letters[rand.Intn(len(letters))]
    }
    return string(s)
}
```

LaunchTestProgram is responsible for:

- building the program
- launching the program
- waiting for it to listen on port 8080
- providing a cleanup function to kill the program and delete it to ensure that when our tests finish, we're left in a clean state
- providing an interrupt function to send the program a SIGTERM to let us test the behaviour

Admittedly, this is not the nicest code in the world, but just focus on the exported function LaunchTestProgram, the un-exported functions it calls are uninteresting boilerplate.

As discussed, acceptance testing tends to be trickier to set up. This code does make the testing code substantially simpler to read, and often with acceptance tests once you've written the ceremonious code, it's done, and you can forget about it.

The acceptance test(s)

We wanted to have two acceptance tests for two programs, one with graceful shutdown and one without, so we, and the readers can see the difference in behaviour. With LaunchTestProgram to build and run the programs, it's quite simple to write acceptance tests for both, and we benefit from re-use with some helper functions.

Here is the test for the server with a graceful shutdown, [you can find the test without on GitHub](#)

```
package main

import (
    "testing"
    "time"

    "github.com/quii/go-graceful-shutdown/acceptancetests"
    "github.com/quii/go-graceful-shutdown/assert"
```

```
)  
  
const (  
    port = "8080"  
    url = "<http://localhost:" > +port  
)  
  
func TestGracefulShutdown(t *testing.T) {  
    cleanup, sendInterrupt, err := acceptancetests.LaunchTestProgram(port)  
    if err != nil {  
        t.Fatal(err)  
    }  
    t.Cleanup(cleanup)  
  
    // just check the server works before we shut things down  
    assert.CanGet(t, url)  
  
    // fire off a request, and before it has a chance to respond send SIGTERM.  
    time.AfterFunc(50*time.Millisecond, func() {  
        assert.NoError(t, sendInterrupt())  
    })  
    // Without graceful shutdown, this would fail  
    assert.CanGet(t, url)  
  
    // after interrupt, the server should be shutdown, and no more requests will work  
    assert.CantGet(t, url)  
}  
}
```

With the setup encapsulated away, the tests are comprehensive, describe the behaviour, and are relatively easy to follow.

assert.CanGet/CantGet are helper functions I made to DRY up this common assertion for this suite.

```
func CanGet(t testing.TB, url string) {  
    errChan := make(chan error)  
  
    go func() {  
        res, err := http.Get(url)  
        if err != nil {  
            errChan <- err  
            return  
        }  
        res.Body.Close()  
        errChan <- nil  
    }()  
}
```

```
select {
    case err := <-errChan:
        NoError(t, err)
    case <-time.After(3 * time.Second):
        t.Errorf("timed out waiting for request to %q", url)
    }
}
```

This will fire off a GET to URL on a goroutine, and if it responds without error before 3 seconds, then it will not fail. `CantGet` is omitted for brevity, [but you can view it on GitHub here](#).

It's important to note again, Go has all the tools you need to write acceptance tests out of the box. You don't need a special framework to build acceptance tests.

Small investment with a big pay-off

With these tests, readers can look at the example programs and be confident that the example actually works, so they can be confident in the package's claims.

Importantly, as the author, we get **fast feedback** and **massive confidence** that the package works in a real-world setting.

```
go test -count=1 ./...
ok    github.com/quii/go-graceful-shutdown 0.196s
?    github.com/quii/go-graceful-shutdown/acceptancetests [no test files]
ok    github.com/quii/go-graceful-shutdown/acceptancetests/withgracefulshutdown 4.785s
ok    github.com/quii/go-graceful-shutdown/acceptancetests/withoutgracefulshutdown 2.914s
?    github.com/quii/go-graceful-shutdown/assert [no test files]
```

Wrapping up

In this blog post, we introduced acceptance tests into your testing tool belt. They are invaluable when you start to build real systems and are an important complement to your unit tests.

The nature of how to write acceptance tests depends on the system you're building, but the principles stay the same. Treat your system like a "black box". If you're making a website, your tests should act like a user, so you'll want to use a headless web browser like [Selenium](#), to click on links, fill in forms, etc. For a RESTful API, you'll send HTTP requests using a client.

Taking it further for more complicated systems

Non-trivial systems don't tend to be single-process applications like the one we've discussed. Typically, you'll depend on other systems such as a database. For these scenarios, you'll need to automate a local environment to test with. Tools like [docker-compose](#) are useful for spinning up containers of the environment you need to run your system locally.

The next chapter

In this post the acceptance test was written retrospectively. However, in [Growing Object-Oriented Software](#) the authors show that we can use acceptance tests in a test-driven approach to act as a "north-star" to guide our efforts.

As systems get more complex, the costs of writing and maintaining acceptance tests can quickly spiral out of control. There are countless stories of development teams being hamstrung by expensive acceptance test suites.

The next chapter will introduce using acceptance test to guide our design along with principles and techniques for managing the costs of acceptance tests.

Improving the quality of open-source

If you're writing packages you intend to share, I'd encourage you to create simple example programs demonstrating what your package does and invest time in having simple-to-follow acceptance tests to give yourself, and potential users of your work, confidence.

Like [Testable Examples](#), seeing this little extra effort in developer experience goes a long way toward building trust in your work, and will reduce your own maintenance costs.

Recruitment plug for \$WORK

If you fancy working in an environment with other engineers solving interesting problems, live near or around London or Porto, and enjoy the contents of this chapter and book - please [reach out to me on Twitter](#), and maybe we can work together soon!

Learn Go with Tests - Scaling Acceptance Tests (and light intro to gRPC)

This chapter is a follow-up to [Intro to acceptance tests](#). You can find [the finished code for this chapter on GitHub](#).

Acceptance tests are essential, and they directly impact your ability to confidently evolve your system over time, with a reasonable cost of change.

They're also a fantastic tool to help you work with legacy code. When faced with a poor codebase without any tests, please resist the temptation to start refactoring. Instead, write some acceptance tests to give you a safety net to freely change the system's internals without affecting its functional external behaviour. ATs need not be concerned with internal quality, so they're a great fit in these situations.

After reading this, you'll appreciate that acceptance tests are useful for verification and can also be used in the development process by helping us change our system more deliberately and methodically, reducing wasted effort.

Prerequisite material

The inspiration for this chapter is borne of many years of frustration with acceptance tests. Two videos I would recommend you watch are:

- Dave Farley - [How to write acceptance tests](#)
- Nat Pryce - [E2E functional tests that can run in milliseconds](#)

"Growing Object Oriented Software" (GOOS) is such an important book for many software engineers, including myself. The approach it prescribes is the one I coach engineers I work with to follow.

- [GOOS](#) - Nat Pryce & Steve Freeman

Finally, [Riya Dattani](#) and I spoke about this topic in the context of BDD in our talk, [Acceptance tests, BDD and Go](#).

Recap

We're talking about "black-box" tests that verify your system behaves as expected from the outside, from a "**business perspective**". The tests do not have access to the innards of the system it tests; they're only concerned with **what** your system does rather than **how**.

Anatomy of bad acceptance tests

Over many years, I've worked for several companies and teams. Each of them recognised the need for acceptance tests; some way to test a system from a user's point of view and to verify it works how it's intended, but almost without exception, the cost of these tests became a real problem for the team.

- Slow to run
- Brittle
- Flaky
- Expensive to maintain, and seem to make changing the software harder than it ought to be
- Can only run in a particular environment, causing slow and poor feedback loops

Let's say you intend to write an acceptance test around a website you're building. You decide to use a headless web browser (like [Selenium](#)) to simulate a user clicking buttons on your website to verify it does what it needs to do.

Over time, your website's markup has to change as new features are discovered, and engineers bike-shed over whether something should be an `<article>` or a `<section>` for the billionth time.

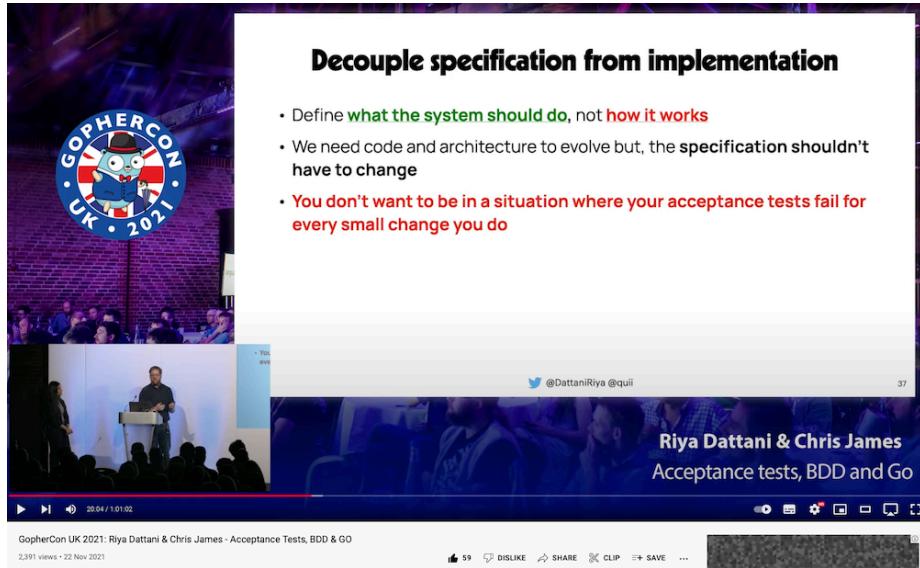
Even though your team are only making minor changes to the system, barely noticeable to the actual user, you find yourself wasting lots of time updating your ATs.

Tight-coupling

Think about what prompts acceptance tests to change:

- An external behaviour change. If you want to change what the system does, changing the acceptance test suite seems reasonable, if not desirable.
- An implementation detail change / refactoring. Ideally, this shouldn't prompt a change, or if it does, a minor one.

Too often, though, the latter is the reason acceptance tests have to change. To the point where engineers even become reluctant to change their system because of the perceived effort of updating tests!



These problems stem from not applying well-established and practised engineering habits written by the authors mentioned above. **You can't write acceptance tests like unit tests**; they require more thought and different practices.

Anatomy of good acceptance tests

If we want acceptance tests that only change when we change behaviour and not implementation detail, it stands to reason that we need to separate those concerns.

On types of complexity

As software engineers, we have to deal with two kinds of complexity.

- **Accidental complexity** is the complexity we have to deal with because we're working with computers, stuff like networks, disks, APIs, etc.
- **Essential complexity** is sometimes referred to as "domain logic". It's the particular rules and truths within your domain.
 - For example, "if an account owner withdraws more money than is available, they are overdrawn". This statement says nothing about computers; this statement was true before computers were even used in banks!

Essential complexity should be expressible to a non-technical person, and it's valuable to have modelled it in our "domain" code, and in our

acceptance tests.

Separation of concerns

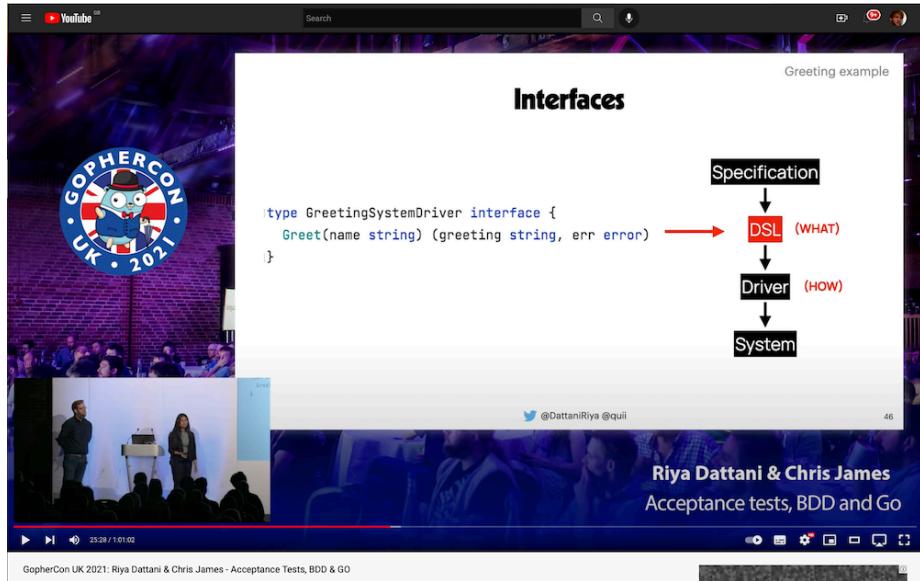
What Dave Farley proposed in the video earlier, and what Riya and I also discussed, is we should have the idea of **specifications**. Specifications describe the behaviour of the system we want without being coupled with accidental complexity or implementation detail.

This idea should feel reasonable to you. In production code, we frequently strive to separate concerns and decouple units of work. Would you not hesitate to introduce an interface to allow your HTTP handler to decouple it from non-HTTP concerns? Let's take this same line of thinking for our acceptance tests.

Dave Farley describes a specific structure.



At GopherconUK, Riya and I put this in Go terms.



Testing on steroids

Decoupling how the specification is executed allows us to reuse it in different scenarios. We can:

Make our drivers configurable This means you can run your ATs locally, in your staging and (ideally) production environments.

- Too many teams engineer their systems such that acceptance tests are impossible to run locally. This introduces an intolerably slow feedback loop. Wouldn't you rather be confident your ATs will pass before integrating your code? If the tests start breaking, is it acceptable that you'd be unable to reproduce the failure locally and instead, have to commit changes and cross your fingers that it'll pass 20 minutes later in a different environment?
- Remember, just because your tests pass in staging doesn't mean your system will work. Dev/Prod parity is, at best, a white lie. [I test in prod](#).
- There are always differences between the environments that can affect the behaviour of your system. A CDN could have some cache headers incorrectly set; a downstream service you depend on may behave differently; a configuration value may be incorrect. But wouldn't it be nice if you could run your specifications in prod to catch these problems quickly?

Plug in different drivers to test other parts of your system

This flexibility allows us to test behaviours at different abstraction and architectural layers, which allows us to have more focused tests beyond black-box tests.

- For instance, you may have a web page with an API behind it. Why not use the same specification to test both? You can use a headless web browser for the web page, and HTTP calls for the API.
- Taking this idea further, ideally, we want the **code to model essential complexity** (as "domain" code) so we should also be able to use our specifications for unit tests. This will give swift feedback that the essential complexity in our system is modelled and behaves correctly.

Acceptance tests changing for the right reasons

With this approach, the only reason for your specifications to change is if the behaviour of the system changes, which is reasonable.

- If your HTTP API has to change, you have one obvious place to update it, the driver.
- If your markup changes, again, update the specific driver.

As your system grows, you'll find yourself reusing drivers for multiple tests, which again means if implementation detail changes, you only have to update one, usually obvious place.

When done right, this approach gives us flexibility in our implementation detail and stability in our specifications. Importantly, it provides a simple and obvious structure for managing change, which becomes essential as a system and its team grows.

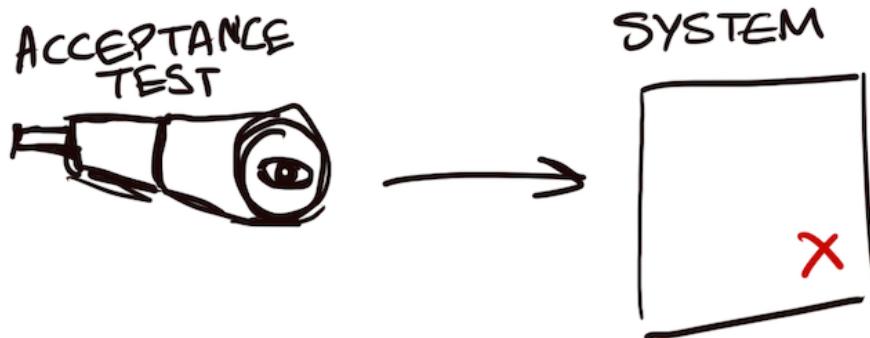
Acceptance tests as a method for software development

In our talk, Riya and I discussed acceptance tests and their relation to BDD. We talked about how starting your work by trying to understand the problem you're trying to solve and expressing it as a specification helps focus your intent and is a great way to start your work.

I was first introduced to this way of working in GOOS. A while ago, I summarised the ideas on my blog. Here is an extract from my post [Why TDD](#)

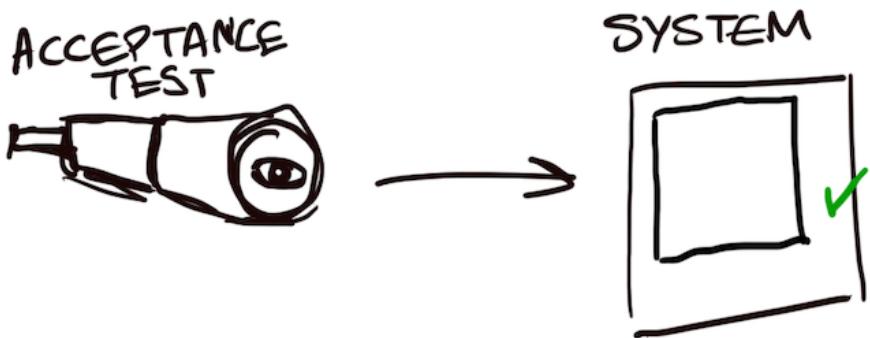
TDD is focused on letting you design for the behaviour you precisely need, iteratively. When starting a new area, you must identify a key, necessary behaviour and aggressively cut scope.

Follow a "top-down" approach, starting with an acceptance test (AT) that exercises the behaviour from the outside. This will act as a north-star for your efforts. All you should be focused on is making that test pass. This test will likely be failing for a while whilst you develop enough code to make it pass.



Once your AT is set up, you can break into the TDD process to drive out enough units to make the AT pass. The trick is to not worry too much about design at this point; get enough code to make the AT pass because you're still learning and exploring the problem.

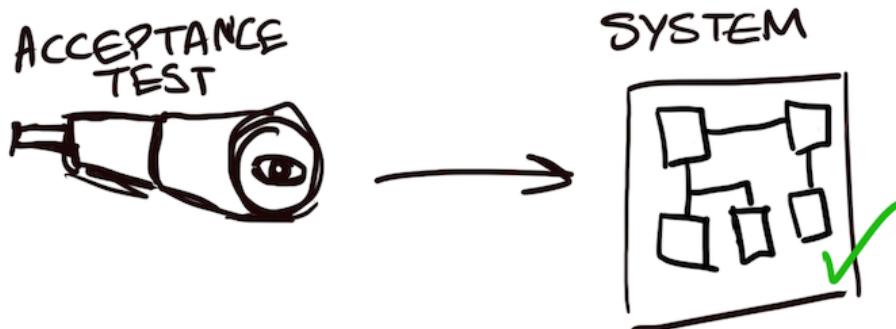
Taking this first step is often more extensive than you think, setting up web servers, routing, configuration, etc., which is why keeping the scope of the work small is essential. We want to make that first positive step on our blank canvas and have it backed by a passing AT so we can continue to iterate quickly and safely.



As you develop, listen to your tests, and they should give you signals to help you push your design in a better direction but, again, anchored to the behaviour rather than our imagination.

Typically, your first "unit" that does the hard work to make the AT pass will grow too big to be comfortable, even for this small amount

of behaviour. This is when you can start thinking about how to break the problem down and introduce new collaborators.



This is where test doubles (e.g. fakes, mocks) are handy because most of the complexity that lives internally within software doesn't usually reside in implementation detail but "between" the units and how they interact.

The perils of bottom-up This is a "top-down" approach rather than a "bottom-up". Bottom-up has its uses, but it carries an element of risk. By building "services" and code without it being integrated into your application quickly and without verifying a high-level test, **you risk wasting lots of effort on unvalidated ideas.**

This is a crucial property of the acceptance-test-driven approach, using tests to get real validation of our code.

Too many times, I've encountered engineers who have made a chunk of code, in isolation, bottom-up, they think is going to solve a job, but it:

- Doesn't work how we want to
- Does stuff we don't need
- Doesn't integrate easily
- Requires a ton of re-writing anyway

This is waste.

Enough talk, time to code

Unlike other chapters, you'll need [Docker](#) installed because we'll be running our applications in containers. It's assumed at this point in the book you're comfortable writing Go code, importing from different packages, etc.

Create a new project with go mod init github.com/quii/go-specs-greet (you can put whatever you like here but if you change the path you will need to change all internal imports to match)

Make a folder specifications to hold our specifications, and add a file greet.go

package specifications

```
import (
    "testing"

    "github.com/alecthomas/assert/v2"
)

type Greeter interface {
    Greet() (string, error)
}

func GreetSpecification(t testing.TB, greeter Greeter) {
    got, err := greeter.Greet()
    assert.NoError(t, err)
    assert.Equal(t, got, "Hello, world")
}
```

My IDE (Goland) takes care of the fuss of adding dependencies for me, but if you need to do it manually, you'd do

go get github.com/alecthomas/assert/v2

Given Farley's acceptance test design (Specification->DSL->Driver->System), we now have a decoupled specification from implementation. It doesn't know or care about how we Greet; it's just concerned with the essential complexity of our domain. Admittedly this complexity isn't much right now, but we'll expand upon the spec to add more functionality as we further iterate. It's always important to start small!

You could view the interface as our first step of a DSL; as the project grows, you may find the need to abstract differently, but for now, this is fine.

At this point, this level of ceremony to decouple our specification from implementation might make some people accuse us of "overly abstracting". **I promise you that acceptance tests that are too coupled to implementation become a real burden on engineering teams.** I am confident that most acceptance tests out in the wild are expensive to maintain due to this inappropriate coupling; rather than the reverse of being overly abstract.

We can use this specification to verify any "system" that can Greet.

First system: HTTP API

We require to provide a "greeter service" over HTTP. So we'll need to create:

1. A **driver**. In this case, one works with an HTTP system by using an **HTTP client**. This code will know how to work with our API. Drivers translate DSLs into system-specific calls; in our case, the driver will implement the interface specifications define.
2. An **HTTP server** with a greet API
3. A **test**, which is responsible for managing the life-cycle of spinning up the server and then plugging the driver into the specification to run it as a test

Write the test first

The initial process for creating a black-box test that compiles and runs your program, executes the test and then cleans everything up can be quite labour intensive. That's why it's preferable to do it at the start of your project with minimal functionality. I typically start all my projects with a "hello world" server implementation, with all of my tests set up and ready for me to build the actual functionality quickly.

The mental model of "specifications", "drivers", and "acceptance tests" can take a little time to get used to, so follow carefully. It can be helpful to "work backwards" by trying to call the specification first.

Create some structure to house the program we intend to ship.

```
mkdir -p cmd/httpserver
```

Inside the new folder, create a new file greeter_server_test.go, and add the following.

```
package main_test

import (
    "testing"

    "github.com/quii/go-specs-greet/specifications"
)

func TestGreeterServer(t *testing.T) {
    specifications.GreetSpecification(t, nil)
}
```

We wish to run our specification in a Go test. We already have access to a `*testing.T`, so that's the first argument, but what about the second?

`specifications.Greeter` is an interface, which we will implement with a Driver by changing the new `TestGreeterServer` code to the following:

```
import (
    go_specs_greet "github.com/quii/go-specs-greet"
)

func TestGreeterServer(t *testing.T) {
    driver := go_specs_greet.Driver{BaseURL: "http://localhost:8080"}
    specifications.GreetSpecification(t, driver)
}
```

It would be favourable for our Driver to be configurable to run it against different environments, including locally, so we have added a `BaseUrl` field.

Try to run the test

```
./greeter_server_test.go:46:12: undefined: go_specs_greet.Driver
```

We're still practising TDD here! It's a big first step we have to make; we need to make a few files and write maybe more code than we're typically used to, but when you're first starting, this is often the case. It's so important we try to remember the red step's rules.

Commit as many sins as necessary to get the test passing

Write the minimal amount of code for the test to run and check the failing test output

Hold your nose; remember, we can refactor when the test has passed. Here's the code for the driver in `driver.go` which we will place in the project root:

```
package go_specs_greet

import (
    "io"
    "net/http"
)

type Driver struct {
    BaseURL string
}
```

```
func (d Driver) Greet() (string, error) {
    res, err := http.Get(d.BaseURL + "/greet")
    if err != nil {
        return "", err
    }
    defer res.Body.Close()
    greeting, err := io.ReadAll(res.Body)
    if err != nil {
        return "", err
    }
    return string(greeting), nil
}
```

Notes:

- You could argue that I should be writing tests to drive out the various if err != nil, but in my experience, so long as you're not doing anything with the err, tests that say "you return the error you get" are relatively low value.
- **You shouldn't use the default HTTP client.** Later we'll pass in an HTTP client to configure it with timeouts etc., but for now, we're just trying to get ourselves to a passing test.
- In our greeter_server_test.go we called the Driver function from go_specs_greet package which we have now created, don't forget to add github.com/quii/go-specs-greet to its imports. Try and rerun the tests; they should now compile but not pass.

Get "http://localhost:8080/greet": dial tcp [::1]:8080: connect: connection refused

We have a Driver, but we have not started our application yet, so it cannot do an HTTP request. We need our acceptance test to coordinate building, running and finally killing our system for the test to run.

Running our application

It's common for teams to build Docker images of their systems to deploy, so for our test we'll do the same

To help us use Docker in our tests, we will use [Testcontainers](#). Testcontainers gives us a programmatic way to build Docker images and manage container life-cycles.

go get github.com/testcontainers/testcontainers-go

Now you can edit cmd/httpserver/greeter_server_test.go to read as follows:

```
package main_test

import (
    "context"
    "testing"

    "github.com/alecthomas/assert/v2"
    go_specs_greet "github.com/quii/go-specs-greet"
    "github.com/quii/go-specs-greet/specifications"
    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

func TestGreeterServer(t *testing.T) {
    ctx := context.Background()

    req := testcontainers.ContainerRequest{
        FromDockerfile: testcontainers.FromDockerfile{
            Context:   "../../",
            Dockerfile: "./cmd/httpserver/Dockerfile",
            // set to false if you want less spam, but this is helpful if you're having troubles
            PrintBuildLog: true,
        },
        ExposedPorts: []string{"8080:8080"},
        WaitingFor:  wait.ForHTTP("/").WithPort("8080"),
    }
    container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
        ContainerRequest: req,
        Started:         true,
    })
    assert.NoError(t, err)
    t.Cleanup(func() {
        assert.NoError(t, container.Terminate(ctx))
    })

    driver := go_specs_greet.Driver{BaseURL: "http://localhost:8080"}
    specifications.GreetSpecification(t, driver)
}

Try and run the test.

==== RUN TestGreeterHandler
2022/09/10 18:49:44 Starting container id: 03e8588a1be4 image: docker.io/testcontainers/ryuk
2022/09/10 18:49:45 Waiting for container id 03e8588a1be4 image: docker.io/testcontainers/ryuk
2022/09/10 18:49:45 Container is ready id: 03e8588a1be4 image: docker.io/testcontainers/ryuk
greeter_server_test.go:32: Did not expect an error but got:
    Error response from daemon: Cannot locate specified Dockerfile: ./cmd/httpserver/Dockerfile
```

```
--- FAIL: TestGreeterHandler (0.59s)
```

We need to create a Dockerfile for our program. Inside our httpserver folder, create a Dockerfile and add the following.

```
FROM golang:1.18-alpine  
WORKDIR /app  
COPY go.mod ./  
RUN go mod download  
COPY ..  
RUN go build -o svr cmd/httpserver/*.go
```

```
EXPOSE 8080  
CMD [ "./svr" ]
```

Don't worry too much about the details here; it can be refined and optimised, but for this example, it'll suffice. The advantage of our approach here is we can later improve our Dockerfile and have a test to prove it works as we intend it to. This is a real strength of having black-box tests!

Try and rerun the test; it should complain about not being able to build the image. Of course, that's because we haven't written a program to build yet!

For the test to fully execute, we'll need to create a program that listens on 8080, but **that's all**. Stick to the TDD discipline, don't write the production code that would make the test pass until we've verified the test fails as we'd expect.

Create a main.go inside our httpserver folder with the following

```
package main  
  
import (  
    "log"  
    "net/http"  
)  
  
func main() {  
    handler := http.HandlerFunc(func(writer http.ResponseWriter, request *http.Request) {  
        })  
    if err := http.ListenAndServe(":8080", handler); err != nil {  
        log.Fatal(err)  
    }  
}
```

```
    }
}
```

Try to run the test again, and it should fail with the following.

```
greet.go:16: Expected values to be equal:  
    +Hello, World  
    \ No newline at end of file  
--- FAIL: TestGreeterHandler (2.09s)
```

Write enough code to make it pass

Update the handler to behave how our specification wants it to

```
import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    handler := http.HandlerFunc(func(w http.ResponseWriter, _ *http.Request) {
        fmt.Fprint(w, "Hello, world")
    })
    if err := http.ListenAndServe(":8080", handler); err != nil {
        log.Fatal(err)
    }
}
```

Refactor

Whilst this technically isn't a refactor, we shouldn't rely on the default HTTP client, so let's change our Driver, so we can supply one, which our test will give.

```
import (
    "io"
    "net/http"
)

type Driver struct {
    baseURL string
    Client  *http.Client
}

func (d Driver) Greet() (string, error) {
    res, err := d.Client.Get(d.baseURL + "/greet")
```

```
if err != nil {
    return "", err
}
defer res.Body.Close()
greeting, err := io.ReadAll(res.Body)
if err != nil {
    return "", err
}
return string(greeting), nil
}
```

In our test in cmd/httpserver/greeter_server_test.go, update the creation of the driver to pass in a client.

```
client := http.Client{
    Timeout: 1 * time.Second,
}
```

```
driver := go_specs_greet.Driver{BaseURL: "http://localhost:8080", Client: &client}
specifications.GreetSpecification(t, driver)
```

It's good practice to keep main.go as simple as possible; it should only be concerned with piecing together the building blocks you make into an application.

Create a file in the project root called handler.go and move our code into there.

```
package go_specs_greet

import (
    "fmt"
    "net/http"
)

func Handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, world")
}
```

Update main.go to import and use the handler instead.

```
package main

import (
    "net/http"
    go_specs_greet "github.com/quii/go-specs-greet"
)
```

```
func main() {
    handler := http.HandlerFunc(go_specs_greet.Handler)
    http.ListenAndServe(":8080", handler)
}
```

Reflect

The first step felt like an effort. We've made several go files to create and test an HTTP handler that returns a hard-coded string. This "iteration 0" ceremony and setup will serve us well for further iterations.

Changing functionality should be simple and controlled by driving it through the specification and dealing with whatever changes it forces us to make. Now the Dockerfile and testcontainers are set up for our acceptance test; we shouldn't have to change these files unless the way we construct our application changes.

We'll see this with our following requirement, greet a particular person.

Write the test first

Edit our specification

package specifications

```
import (
    "testing"
    "github.com/alecthomas/assert/v2"
)

type Greeter interface {
    Greet(name string) (string, error)
}

func GreetSpecification(t testing.TB, greeter Greeter) {
    got, err := greeter.Greet("Mike")
    assert.NoError(t, err)
    assert.Equal(t, got, "Hello, Mike")
}
```

To allow us to greet specific people, we need to change the interface to our system to accept a name parameter.

Try to run the test

```
./greeter_server_test.go:48:39: cannot use driver (variable of type go_specs_greet.Driver) as type  
go_specs_greet.Driver does not implement specifications.Greeter (wrong type for Greet method  
have Greet() (string, error)  
want Greet(name string) (string, error)
```

The change in the specification has meant our driver needs to be updated.

Write the minimal amount of code for the test to run and check the failing test output

Update the driver so that it specifies a name query value in the request to ask for a particular name to be greeted.

```
import "io"  
  
func (d Driver) Greet(name string) (string, error) {  
    res, err := d.Client.Get(d.BaseURL + "/greet?name=" + name)  
    if err != nil {  
        return "", err  
    }  
    defer res.Body.Close()  
    greeting, err := io.ReadAll(res.Body)  
    if err != nil {  
        return "", err  
    }  
    return string(greeting), nil  
}
```

The test should now run, and fail.

```
greet.go:16: Expected values to be equal:  
    -Hello, world  
    \ No newline at end of file  
    +Hello, Mike  
    \ No newline at end of file  
--- FAIL: TestGreeterHandler (1.92s)
```

Write enough code to make it pass

Extract the name from the request and greet.

```
import (  
    "fmt"  
    "net/http"  
)
```

```
func Handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %s", r.URL.Query().Get("name"))
}
```

The test should now pass.

Refactor

In [HTTP Handlers Revisited](#), we discussed how important it is for HTTP handlers should only be responsible for handling HTTP concerns; any “domain logic” should live outside of the handler. This allows us to develop domain logic in isolation from HTTP, making it simpler to test and understand.

Let’s pull apart these concerns.

Update our handler in ./handler.go as follows:

```
func Handler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")
    fmt.Fprint(w, Greet(name))
}
```

Create new file ./greet.go:

```
package go_specs_greet

import "fmt"

func Greet(name string) string {
    return fmt.Sprintf("Hello, %s", name)
}
```

A slight diversion in to the “adapter” design pattern

Now that we’ve separated our domain logic of greeting people into a separate function, we are now free to write unit tests for our greet function. This is undoubtedly a lot simpler than testing it through a specification that goes through a driver that hits a web server, to get a string!

Wouldn’t it be nice if we could reuse our specification here too? After all, the specification’s point is decoupled from implementation details. If the specification captures our **essential complexity** and our “domain” code is supposed to model it, we should be able to use them together.

Let's give it a go by creating ./greet_test.go as follows:

```
package go_specs_greet_test

import (
    "testing"

    go_specs_greet "github.com/quii/go-specs-greet"
    "github.com/quii/go-specs-greet/specifications"
)

func TestGreet(t *testing.T) {
    specifications.GreetSpecification(t, go_specs_greet.Greet)
}
```

This would be nice, but it doesn't work

```
./greet_test.go:11:39: cannot use go_specs_greet.Greet (value of type func(name string) string)
    func(name string) string does not implement specifications.Greeter (missing Greet method)
```

Our specification wants something that has a method `Greet()` not a function.

The compilation error is frustrating; we have a thing that we "know" is a Greeter, but it's not quite in the right **shape** for the compiler to let us use it. This is what the **adapter** pattern caters for.

In software engineering, the **adapter pattern** is a [software design pattern](#) (also known as [wrapper](#), an alternative naming shared with the [decorator pattern](#)) that allows the [interface](#) of an existing [class](#) to be used as another interface.^[1] It is often used to make existing classes work with others without modifying their [source code](#).

A lot of fancy words for something relatively simple, which is often the case with design patterns, which is why people tend to roll their eyes at them. The value of design patterns is not specific implementations but a language to describe specific solutions to common problems engineers face. If you have a team that has a shared vocabulary, it reduces the friction in communication.

Add this code in ./specifications/adapters.go

```
type GreetAdapter func(name string) string

func (g GreetAdapter) Greet(name string) (string, error) {
    return g(name), nil
}
```

We can now use our adapter in our test to plug our `Greet` function into the specification.

```
package go_specs_greet_test

import (
    "testing"

    gospecsgreet "github.com/quii/go-specs-greet"
    "github.com/quii/go-specs-greet/specifications"
)

func TestGreet(t *testing.T) {
    specifications.GreetSpecification(
        t,
        specifications.GreetAdapter(gospecsgreet.Greet),
    )
}
```

The adapter pattern is handy when you have a type that exhibits the behaviour that an interface wants, but isn't in the right shape.

Reflect

The behaviour change felt simple, right? OK, maybe it was simply due to the nature of the problem, but this method of work gives you discipline and a simple, repeatable way of changing your system from top to bottom:

- Analyse your problem and identify a slight improvement to your system that pushes you in the right direction
- Capture the new essential complexity in a specification
- Follow the compilation errors until the AT runs
- Update your implementation to make the system behave according to the specification
- Refactor

After the pain of the first iteration, we didn't have to edit our acceptance test code because we have the separation of specifications, drivers and implementation. Changing our specification required us to update our driver and finally our implementation, but the boilerplate code around how to spin up the system as a container was unaffected.

Even with the overhead of building a docker image for our application and spinning up the container, the feedback loop for testing our **entire** application is very tight:

```
quii@Chriss-MacBook-Pro go-specs-greet % go test ./...
ok    github.com/quii/go-specs-greet 0.181s
ok    github.com/quii/go-specs-greet/cmd/httpserver 2.221s
```

```
?    github.com/quii/go-specs-greet/specifications [no test files]
```

Now, imagine your CTO has now decided that gRPC is the future. She wants you to expose this same functionality over a gRPC server whilst maintaining the existing HTTP server.

This is an example of **accidental complexity**. Remember, accidental complexity is the complexity we have to deal with because we're working with computers, stuff like networks, disks, APIs, etc. **The essential complexity has not changed**, so we shouldn't have to change our specifications.

Many repository structures and design patterns are mainly dealing with separating types of complexity. For instance, "ports and adapters" ask that you separate your domain code from anything to do with accidental complexity; that code lives in an "adapters" folder.

Making the change easy

Sometimes, it makes sense to do some refactoring before making a change.

First make the change easy, then make the easy change

~Kent Beck

For that reason, let's move our http code - driver.go and handler.go - into a package called httpserver within an adapters folder and change their package names to httpserver.

You'll now need to import the root package into handler.go to refer to the Greet method...

```
package httpserver
```

```
import (
    "fmt"
    "net/http"

    go_specs_greet "github.com/quii/go-specs-greet/domain/interactions"
)

func Handler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")
    fmt.Fprint(w, go_specs_greet.Greet(name))
}
```

import your httpserver adapter into main.go:

```
package main

import (
    "net/http"

    "github.com/quii/go-specs-greet/adapters/httpserver"
)

func main() {
    handler := http.HandlerFunc(httpserver.Handler)
    http.ListenAndServe(":8080", handler)
}
```

and update the import and reference to Driver in greeter_server_test.go:

```
driver := httpserver.Driver{BaseURL: "http://localhost:8080", Client: &client}
```

Finally, it's helpful to gather our domain level code in to its own folder too. Don't be lazy and have a domain folder in your projects with hundreds of unrelated types and functions. Make an effort to think about your domain and group ideas that belong together, together. This will make your project easier to understand and will improve the quality of your imports.

Rather than seeing

```
domain.Greet
```

Which is just a bit weird, instead favour

```
interactions.Greet
```

Create a domain folder to house all your domain code, and within it, an interactions folder. Depending on your tooling, you may have to update some imports and code.

Our project tree should now look like this:

```
quii@Chriss-MacBook-Pro go-specs-greet % tree
```

```
.
├── Dockerfile
├── Makefile
├── README.md
└── adapters
    └── httpserver
        ├── driver.go
        └── handler.go
└── cmd
    └── httpserver
        ├── greeter_server_test.go
        └── main.go
```

```
└── domain
    └── interactions
        ├── greet.go
        └── greet_test.go
    └── go.mod
    └── go.sum
    └── specifications
        └── adapters.go
        └── greet.go
```

Our domain code, **essential complexity**, lives at the root of our go module, and code that will allow us to use them in “the real world” are organised into **adapters**. The cmd folder is where we can compose these logical groupings into practical applications, which have black-box tests to verify it all works. Nice!

Finally, we can do a tiny bit of tidying up our acceptance test. If you consider the high-level steps of our acceptance test:

- Build a docker image
- Wait for it to be listening on some port
- Create a driver that understands how to translate the DSL into system specific calls
- Plug in the driver into the specification

... you'll realise we have the same requirements for an acceptance test for the gRPC server!

The adapters folder seems a good place as any, so inside a file called docker.go, encapsulate the first two steps in a function that we'll reuse next.

package adapters

```
import (
    "context"
    "fmt"
    "testing"
    "time"

    "github.com/alechthomas/assert/v2"
    "github.com/docker/go-connections/nat"
    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

func StartDockerServer(
    t testing.TB,
    port string,
```

```
    dockerFilePath string,
) {
    ctx := context.Background()
    t.Helper()
    req := testcontainers.ContainerRequest{
        FromDockerfile: testcontainers.FromDockerfile{
            Context:      "././.",
            Dockerfile:   dockerFilePath,
            PrintBuildLog: true,
        },
        ExposedPorts: []string{fmt.Sprintf("%s:%s", port, port)},
        WaitingFor:  wait.ForListeningPort(nat.Port(port)).WithStartupTimeout(5 * time.Second),
    }
    container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
        ContainerRequest: req,
        Started:         true,
    })
    assert.NoError(t, err)
    t.Cleanup(func() {
        assert.NoError(t, container.Terminate(ctx))
    })
}
```

This gives us an opportunity to clean up our acceptance test a little

```
func TestGreeterServer(t *testing.T) {
    var (
        port      = "8080"
        dockerFilePath = "./cmd/httpserver/Dockerfile"
        baseURL   = fmt.Sprintf("http://localhost:%s", port)
        driver     = httpserver.Driver{BaseURL: baseURL, Client: &http.Client{
            Timeout: 1 * time.Second,
        }}
    )
    adapters.StartDockerServer(t, port, dockerFilePath)
    specifications.GreetSpecification(t, driver)
}
```

This should make writing the next test simpler.

Write the test first

This new functionality can be accomplished by creating a new adapter to interact with our domain code. For that reason we:

- Shouldn't have to change the specification;

-
- Should be able to reuse the specification;
 - Should be able to reuse the domain code.

Create a new folder grpcserver inside cmd to house our new program and the corresponding acceptance test. Inside cmd/grpc_server/greeter_server_test.go, add an acceptance test, which looks very similar to our HTTP server test, not by coincidence but by design.

```
package main_test

import (
    "fmt"
    "testing"

    "github.com/quii/go-specs-greet/adapters"
    "github.com/quii/go-specs-greet/adapters/grpcserver"
    "github.com/quii/go-specs-greet/specifications"
)

func TestGreeterServer(t *testing.T) {
    var (
        port      = "50051"
        dockerFilePath = "./cmd/grpcserver/Dockerfile"
        driver     = grpcserver.Driver{Addr: fmt.Sprintf("localhost:%s", port)}
    )

    adapters.StartDockerServer(t, port, dockerFilePath)
    specifications.GreetSpecification(t, &driver)
}
```

The only differences are:

- We use a different docker file, because we're building a different program
- This means we'll need a new Driver, that'll use gRPC to interact with our new program

Try to run the test

```
./greeter_server_test.go:26:12: undefined: grpcserver
```

We haven't created a Driver yet, so it won't compile.

Write the minimal amount of code for the test to run and check the failing test output

Create a grpcserver folder inside adapters and inside it create driver.go

```
package grpcserver

type Driver struct {
    Addr string
}

func (d Driver) Greet(name string) (string, error) {
    return "", nil
}
```

If you run again, it should now compile but not pass because we haven't created a Dockerfile and corresponding program to run.

Create a new Dockerfile inside cmd/grpcserver.

```
FROM golang:1.18-alpine
```

```
WORKDIR /app
```

```
COPY go.mod ./
```

```
RUN go mod download
```

```
COPY ..
```

```
RUN go build -o svr cmd/grpcserver/*.go
```

```
EXPOSE 8080
CMD [ "./svr" ]
```

And a main.go

```
package main
```

```
import "fmt"
```

```
func main() {
    fmt.Println("implement me")
}
```

You should find now that the test fails because our server is not listening on the port. Now is the time to start building our client and server with gRPC.

Write enough code to make it pass

gRPC

If you're unfamiliar with gRPC, I'd start by looking at the [gRPC website](#). Still, for this chapter, it's just another kind of adapter into our system, a way for other systems to call (remote **p**rocedure **c**all) our excellent domain code.

The twist is you define a "service definition" using Protocol Buffers. You then generate server and client code from the definition. This not only works for Go but for most mainstream languages too. This means you can share a definition with other teams in your company who may not even write Go and can still do service-to-service communication smoothly.

If you haven't used gRPC before, you'll need to install a **Protocol buffer compiler** and some **Go plugins**. The [gRPC website](#) has clear instructions on [how to do this](#).

Inside the same folder as our new driver, add a greet.proto file with the following

```
syntax = "proto3";  
  
option go_package = "github.com/quii/adapters/grpcserver";  
  
package grpcserver;  
  
service Greeter {  
    rpc Greet (GreetRequest) returns (GreetReply) {}  
}  
  
message GreetRequest {  
    string name = 1;  
}  
  
message GreetReply {  
    string message = 1;  
}
```

To understand this definition, you don't need to be an expert in Protocol Buffers. We define a service with a Greet method and then describe the incoming and outgoing message types.

Inside adapters/grpcserver run the following to generate the client and server code

```
protoc --go_out=. --go_opt=paths=source_relative \  
--go-grpc_out=. --go-grpc_opt=paths=source_relative \
```

```
greet.proto
```

If it worked, we would have some code generated for us to use. Let's start by using the generated client code inside our Driver.

```
package grpcserver

import (
    "context"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
)

type Driver struct {
    Addr string
}

func (d Driver) Greet(name string) (string, error) {
    //todo: we shouldn't redial every time we call greet, refactor out when we're green
    conn, err := grpc.Dial(d.Addr, grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        return "", err
    }
    defer conn.Close()

    client := NewGreeterClient(conn)
    greeting, err := client.Greet(context.Background(), &GreetRequest{
        Name: name,
    })
    if err != nil {
        return "", err
    }

    return greeting.Message, nil
}
```

Now that we have a client, we need to update our main.go to create a server. Remember, at this point; we're just trying to get our test to pass and not worrying about code quality.

```
package main
```

```
import (
    "context"
    "log"
    "net"
```

```
"github.com/quii/go-specs-greet/adapters/grpcserver"
"google.golang.org/grpc"
)

func main() {
    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatal(err)
    }
    s := grpc.NewServer()
    grpcserver.RegisterGreeterServer(s, &GreetServer{})

    if err := s.Serve(lis); err != nil {
        log.Fatal(err)
    }
}

type GreetServer struct {
    grpcserver.UnimplementedGreeterServer
}

func (g GreetServer) Greet(ctx context.Context, request *grpcserver.GreetRequest) (*grpcserver.GreetReply, error) {
    return &grpcserver.GreetReply{Message: "fixme"}, nil
}
```

To create our gRPC server, we have to implement the interface it generated for us

```
// GreeterServer is the server API for Greeter service.
// All implementations must embed UnimplementedGreeterServer
// for forward compatibility
type GreeterServer interface {
    Greet(context.Context, *GreetRequest) (*GreetReply, error)
    mustEmbedUnimplementedGreeterServer()
}
```

Our main function:

- Listens on a port
- Creates a GreetServer that implements the interface, and then registers it with grpcServer.RegisterGreeterServer, along with a grpc.Server.
- Uses the server with the listener

It wouldn't be a massive extra effort to call our domain code inside greetServer.Greet rather than hard-coding fix-me in the message, but I'd like to run our acceptance test first to see if everything is working on a transport level and verify the failing test output.

```
greet.go:16: Expected values to be equal:  
-fixme  
\ No newline at end of file  
+Hello, Mike  
\ No newline at end of file
```

Nice! We can see our driver is able to connect to our gRPC server in the test.

Now, call our domain code inside our GreetServer

```
type GreetServer struct {  
    grpcserver.UnimplementedGreeterServer  
}  
  
func (g GreetServer) Greet(ctx context.Context, request *grpcserver.GreetRequest) (*grpcserver.  
    return &grpcserver.GreetReply{Message: interactions.Greet(request.Name)}, nil  
}
```

Finally, it passes! We have an acceptance test that proves our gRPC greet server behaves how we'd like.

Refactor

We committed several sins to get the test passing, but now they're passing, we have the safety net to refactor.

Simplify main

As before, we don't want main to have too much code inside it. We can move our new GreetServer into adapters/grpcserver as that's where it should live. In terms of cohesion, if we change the service definition, we want the "blast-radius" of change to be confined to that area of our code.

Don't redial in our driver every time

We only have one test, but if we expand our specification (we will), it doesn't make sense for the Driver to redial for every RPC call.

```
package grpcserver  
  
import (  
    "context"  
    "sync"  
  
    "google.golang.org/grpc"
```

```
        "google.golang.org/grpc/credentials/insecure"
    )

type Driver struct {
    Addr string

    connectionOnce sync.Once
    conn          *grpc.ClientConn
    client        GreeterClient
}

func (d *Driver) Greet(name string) (string, error) {
    client, err := d.getClient()
    if err != nil {
        return "", err
    }

    greeting, err := client.Greet(context.Background(), &GreetRequest{
        Name: name,
    })
    if err != nil {
        return "", err
    }

    return greeting.Message, nil
}

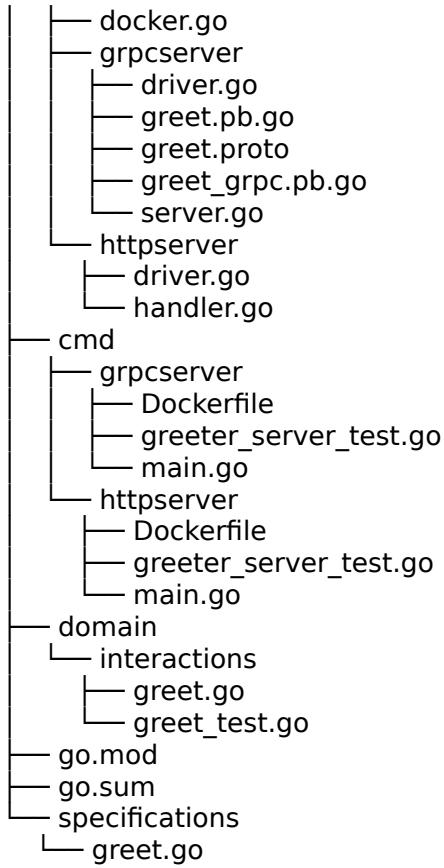
func (d *Driver) getClient() (GreeterClient, error) {
    var err error
    d.connectionOnce.Do(func() {
        d.conn, err = grpc.Dial(d.Addr, grpc.WithTransportCredentials(insecure.NewCredentials()))
        d.client = NewGreeterClient(d.conn)
    })
    return d.client, err
}
```

Here we're showing how we can use `sync.Once` to ensure our Driver only attempts to create a connection to our server once.

Let's take a look at the current state of our project structure before moving on.

```
quii@Chriss-MacBook-Pro go-specs-greet % tree
```

```
.
├── Makefile
├── README.md
└── adapters
```



- adapters have cohesive units of functionality grouped together
- cmd holds our applications and corresponding acceptance tests
- Our code is totally decoupled from any accidental complexity

Consolidating Dockerfile

You've probably noticed the two Dockerfiles are almost identical beyond the path to the binary we wish to build.

Dockerfiles can accept arguments to let us reuse them in different contexts, which sounds perfect. We can delete our 2 Dockerfiles and instead have one at the root of the project with the following

```
FROM golang:1.18-alpine
```

```
WORKDIR /app
```

```
ARG bin_to_build
```

```
COPY go.mod ./  
RUN go mod download  
COPY ..  
RUN go build -o svr cmd/${bin_to_build}/main.go  
CMD [ "./svr" ]
```

We'll have to update our StartDockerServer function to pass in the argument when we build the images

```
func StartDockerServer(  
    t testing.TB,  
    port string,  
    binToBuild string,  
) {  
    ctx := context.Background()  
    t.Helper()  
    req := testcontainers.ContainerRequest{  
        FromDockerfile: testcontainers.FromDockerfile{  
            Context: " ../../.",  
            Dockerfile: "Dockerfile",  
            BuildArgs: map[string]*string{  
                "bin_to_build": &binToBuild,  
            },  
            PrintBuildLog: true,  
        },  
        ExposedPorts: []string{fmt.Sprintf("%s:%s", port, port)},  
        WaitingFor: wait.ForListeningPort(nat.Port(port)).WithStartupTimeout(5 * time.Second),  
    }  
    container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{  
        ContainerRequest: req,  
        Started: true,  
    })  
    assert.NoError(t, err)  
    t.Cleanup(func() {  
        assert.NoError(t, container.Terminate(ctx))  
    })  
}
```

And finally, update our tests to pass in the image to build (do this for the other test and change grpcserver to httpserver).

```
func TestGreeterServer(t *testing.T) {  
    var (  
        port = "50051"  
    )
```

```
    driver = grpcserver.Driver{Addr: fmt.Sprintf("localhost:%s", port)}
)
adapters.StartDockerServer(t, port, "grpcserver")
specifications.GreetSpecification(t, &driver)
}
```

Separating different kinds of tests

Acceptance tests are great in that they test the whole system works from a pure user-facing, behavioural POV, but they do have their downsides compared to unit tests:

- Slower
- Quality of feedback is often not as focused as a unit test
- Doesn't help you with internal quality, or design

The [Test Pyramid](#) guides us on the kind of mix we want for our test suite, you should read Fowler's post for more detail, but the very simplistic summary for this post is "lots of unit tests and a few acceptance tests".

For that reason, as a project grows you often may be in situations where the acceptance tests can take a few minutes to run. To offer a friendly developer experience for people checking out your project, you can enable developers to run the different kinds of tests separately.

It's preferable that running `go test ./...` should be runnable with no further set up from an engineer, beyond say a few key dependencies such as the Go compiler (obviously) and perhaps Docker.

Go provides a mechanism for engineers to run only "short" tests with the [short flag](#)

```
go test -short ./...
```

We can add to our acceptance tests to see if the user wants to run our acceptance tests by inspecting the value of the flag

```
if testing.Short() {
    t.Skip()
}
```

I made a Makefile to show this usage

```
build:
    golangci-lint run
    go test ./...
```

```
unit-tests:  
  go test -short ./...
```

When should I write acceptance tests?

The best practice is to favour having lots of fast running unit tests and a few acceptance tests, but how do you decide when you should write an acceptance test, vs unit tests?

It's difficult to give a concrete rule, but the questions I typically ask myself are:

- Is this an edge case? I'd prefer to unit test those
- Is this something that the non-computer people talk about a lot? I would prefer to have a lot of confidence the key thing "really" works, so I'd add an acceptance test
- Am I describing a user journey, rather than a specific function? Acceptance test
- Would unit tests give me enough confidence? Sometimes you're taking an existing journey that already has an acceptance test, but you're adding other functionality to deal with different scenarios due to different inputs. In this case, adding another acceptance test adds a cost but brings little value, so I'd prefer some unit tests.

Iterating on our work

With all this effort, you'd hope extending our system will now be simple. Making a system that is simple to work on, is not necessarily easy, but it's worth the time, and is substantially easier to do when you start a project.

Let's extend our API to include a "curse" functionality.

Write the test first

This is brand-new behaviour, so we should start with an acceptance test. In our specification file, add the following

```
type MeanGreeter interface {  
    Curse(name string) (string, error)  
}  
  
func CurseSpecification(t *testing.T, meany MeanGreeter) {  
    got, err := meany.Curse("Chris")  
    assert.NoError(t, err)
```

```
    assert.Equal(t, got, "Go to hell, Chris!")
}
```

Pick one of our acceptance tests and try to use the specification

```
func TestGreeterServer(t *testing.T) {
    if testing.Short() {
        t.Skip()
    }
    var (
        port  = "50051"
        driver = grpcserver.Driver{Addr: fmt.Sprintf("localhost:%s", port)}
    )

    t.Cleanup(driver.Close)
    adapters.StartDockerServer(t, port, "grpcserver")
    specifications.GreetSpecification(t, &driver)
    specifications.CurseSpecification(t, &driver)
}
```

Try to run the test

```
# github.com/quii/go-specs-greet/cmd/grpcserver_test [github.com/quii/go-specs-greet/cmd/grpcserver_test.go:27:39: cannot use &driver (value of type *grpcserver.Driver) as type specifications.MeanGreeter
*grpcserver.Driver does not implement specifications.MeanGreeter (missing Curse method)]
```

Our Driver doesn't support Curse yet.

Write the minimal amount of code for the test to run and check the failing test output

Remember we're just trying to get the test to run, so add the method to Driver

```
func (d *Driver) Curse(name string) (string, error) {
    return "", nil
}
```

If you try again, the test should compile, run, and fail

```
greet.go:26: Expected values to be equal:
+Go to hell, Chris!
\ No newline at end of file
```

Write enough code to make it pass

We'll need to update our protocol buffer specification have a Curse method on it, and then regenerate our code.

```
service Greeter {
    rpc Greet (GreetRequest) returns (GreetReply) {}
    rpc Curse (GreetRequest) returns (GreetReply) {}
}
```

You could argue that reusing the types GreetRequest and GreetReply is inappropriate coupling, but we can deal with that in the refactoring stage. As I keep stressing, we're just trying to get the test passing, so we verify the software works, then we can make it nice.

Re-generate our code with (inside adapters/grpcserver).

```
protoc --go_out=. --go_opt=paths=source_relative \
--go-grpc_out=. --go-grpc_opt=paths=source_relative \
greet.proto
```

Update driver

Now the client code has been updated, we can now call Curse in our Driver

```
func (d *Driver) Curse(name string) (string, error) {
    client, err := d.getClient()
    if err != nil {
        return "", err
    }

    greeting, err := client.Curse(context.Background(), &GreetRequest{
        Name: name,
    })
    if err != nil {
        return "", err
    }

    return greeting.Message, nil
}
```

Update server

Finally, we need to add the Curse method to our Server

```
package grpcserver
```

```
import (
    "context"
    "fmt"

    "github.com/quii/go-specs-greet/domain/interactions"
```

```
)  
  
type GreetServer struct {  
    UnimplementedGreeterServer  
}  
  
func (g GreetServer) Curse(ctx context.Context, request *GreetRequest) (*GreetReply, error) {  
    return &GreetReply{Message: fmt.Sprintf("Go to hell, %s!", request.Name)}, nil  
}  
  
func (g GreetServer) Greet(ctx context.Context, request *GreetRequest) (*GreetReply, error) {  
    return &GreetReply{Message: interactions.Greet(request.Name)}, nil  
}
```

The tests should now pass.

Refactor

Try doing this yourself.

- Extract the Curse “domain logic”, away from the grpc server, as we did for Greet. Use the specification as a unit test against your domain logic
- Have different types in the protobuf to ensure the message types for Greet and Curse are decoupled.

Implementing Curse for the HTTP server

Again, an exercise for you, the reader. We have our domain-level specification and our domain-level logic neatly separated. If you’ve followed this chapter, this should be very straightforward.

- Add the specification to the existing acceptance test for the HTTP server
- Update your Driver
- Add the new endpoint to the server, and reuse the domain code to implement the functionality. You may wish to use http.NewServeMux to handle the routeing to the separate endpoints.

Remember to work in small steps, commit and run your tests frequently. If you get really stuck [you can find my implementation on GitHub](#).

Enhance both systems by updating the domain logic with a unit test

As mentioned, not every change to a system should be driven via an acceptance test. Permutations of business rules and edge cases should be simple to drive via a unit test if you have separated concerns well.

Add a unit test to our Greet function to default the name to World if it is empty. You should see how simple this is, and then the business rules are reflected in both applications for "free".

Wrapping up

Building systems with a reasonable cost of change requires you to have ATs engineered to help you, not become a maintenance burden. They can be used as a means of guiding, or as a GOOS says, "growing" your software methodically.

Hopefully, with this example, you can see our application's predictable, structured workflow for driving change and how you could use it for your work.

You can imagine talking to a stakeholder who wants to extend the system you work on in some way. Capture it in a domain-centric, implementation-agnostic way in a specification, and use it as a north star towards your efforts. Riya and I describe leveraging BDD techniques like "Example Mapping" [in our GopherconUK talk](#) to help you understand the essential complexity more deeply and allow you to write more detailed and meaningful specifications.

Separating essential and accidental complexity concerns will make your work less ad-hoc and more structured and deliberate; this ensures the resiliency of your acceptance tests and helps them become less of a maintenance burden.

Dave Farley gives an excellent tip:

Imagine the least technical person that you can think of, who understands the problem-domain, reading your Acceptance Tests. The tests should make sense to that person.

Specifications should then double up as documentation. They should specify clearly how a system should behave. This idea is the principle around tools like [Cucumber](#), which offers you a DSL for capturing behaviours as code, and then you convert that DSL into system calls, just like we did here.

What has been covered

- Writing abstract specifications allows you to express the essential complexity of the problem you're solving and remove accidental complexity. This will enable you to reuse the specifications in different contexts.
- How to use [Testcontainers](#) to manage the life-cycle of your system for ATs. This allows you to thoroughly test the image you intend to ship on your computer, giving you fast feedback and confidence.
- A brief intro into containerising your application with Docker
- gRPC
- Rather than chasing canned folder structures, you can use your development approach to naturally drive out the structure of your application, based on your own needs

Further material

- In this example, our "DSL" is not much of a DSL; we just used interfaces to decouple our specification from the real world and allow us to express domain logic cleanly. As your system grows, this level of abstraction might become clumsy and unclear. [Read into the "Screenplay Pattern"](#) if you want to find more ideas as to how to structure your specifications.
- For emphasis, [Growing Object-Oriented Software, Guided by Tests](#), is a classic. It demonstrates applying this "London style", "top-down" approach to writing software. Anyone who has enjoyed Learn Go with Tests should get much value from reading GOOS.
- [In the example code repository](#), there's more code and ideas I haven't written about here, such as multi-stage docker build, you may wish to check this out.
 - In particular, for fun, I made a **third program**, a website with some HTML forms to Greet and Curse. The Driver leverages the excellent-looking <https://github.com/go-rod/rod> module, which allows it to work with the website with a browser, just like a user would. Looking at the git history, you can see how I started not using any templating tools "just to make it work" Then, once I passed my acceptance test, I had the freedom to do so without fear of breaking things. -->

Working without mocks, stubs and spies

This chapter delves into the world of test doubles and explores how they influence the testing and development process. We'll uncover the limitations of traditional mocks, stubs, and spies and introduce a more efficient and adaptable approach using fakes and contracts.

tl;dr

- Mocks, spies and stubs encourage you to encode assumptions of the behaviour of your dependencies ad-hocly in each test.
- These assumptions are usually not validated beyond manual checking, so they threaten your test suite's usefulness.
- Fakes and contracts give us a more sustainable method for creating test doubles with validated assumptions and better reuse than the alternatives.

This is a longer chapter than normal, so as a palette cleanser, you should explore an [example repo first](#). In particular, check out the [planner test](#).

In [Mocking](#), we learned how mocks, stubs and spies are useful tools for controlling and inspecting the behaviour of units of code in conjunction with [Dependency Injection](#).

As a project grows, though, these kinds of test doubles can become a maintenance burden, and we should instead look to other design ideas to keep our system easy to reason and test.

Fakes and **contracts** allow developers to test their systems with more realistic scenarios, improve local development experience with faster and more accurate feedback loops, and manage the complexity of evolving dependencies.

A primer on test doubles

It's easy to roll your eyes when people like me are pedantic about the nomenclature of test doubles, but the distinctive kinds of test doubles help us talk about this topic and the trade-offs we're making with clarity.

Test doubles is the collective noun for the different ways you can construct dependencies that you can control for a **subject under test (SUT)**, the thing you're testing. Test doubles are often a better alternative than using the real dependency as it can avoid issues like

-
- Needing the internet to use an API
 - Avoid latency and other performance issues
 - Unable to exercise non-happy path cases
 - Decoupling your build from another team's.
 - You wouldn't want to prevent deployments if an engineer in another team accidentally shipped a bug

In Go, you'll typically model a dependency with an interface, then implement your version to control the behaviour in a test. **Here are the kinds of test doubles covered in this post.**

Given this interface of a hypothetical recipe API:

```
type RecipeBook interface {
    GetRecipes() ([]Recipe, error)
    AddRecipes(...Recipe) error
}
```

We can construct test doubles in various ways, depending on how we're trying to test something that uses a RecipeBook.

Stubs return the same canned data every time they are called

```
type StubRecipeStore struct {
    recipes []Recipe
    err     error
}

func (s *StubRecipeStore) GetRecipes() ([]Recipe, error) {
    return s.recipes, s.err
}

// AddRecipes omitted for brevity
```

```
// in test, we can set up the stub to always return specific recipes, or an error
stubStore := &StubRecipeStore{
    recipes: someRecipes,
}
```

Spies are like stubs but also record how they were called so the test can assert that the SUT calls the dependencies in specific ways.

```
type SpyRecipeStore struct {
    AddCalls [][][]Recipe
    err     error
}

func (s *SpyRecipeStore) AddRecipes(r ...Recipe) error {
    s.AddCalls = append(s.AddCalls, r)
    return s.err
}
```

```
}

// GetRecipes omitted for brevity
// in test
spyStore := &SpyRecipeStore{}
sut := NewThing(spyStore)
sut.DoStuff()

// now we can check the store had the right recipes added by inspectiong spyStore.AddCalls
Mocks are like a superset of the above, but they only respond with specific data to specific invocations. If the SUT calls the dependencies with the wrong arguments, it'll typically panic.

// set up the mock with expected calls
mockStore := &MockRecipeStore{}
mockStore.WhenCalledWith(someRecipes).Return(someError)

// when the sut uses the dependency, if it doesn't call it with someRecipes, usually mocks will p
Fakes are like a genuine version of the dependency but implemented in a way more suited to fast running, reliable tests and local development. Often, your system will have some abstraction around persistence, which will be implemented with a database, but in your tests, you could use an in-memory fake instead.

type FakeRecipeStore struct {
    recipes []Recipe
}

func (f *FakeRecipeStore) GetRecipes() ([]Recipe, error) {
    return f.recipes, nil
}

func (f *FakeRecipeStore) AddRecipes(r ...Recipe) error {
    f.recipes = append(f.recipes, r...)
    return nil
}
```

Fakes are useful because:

- Their statefulness is useful for tests involving multiple subjects and invocations, such as an integration test. Managing state with the other kinds of test doubles is generally discouraged.
- If they have a sensible API, offer a more natural way of asserting state. Rather than spying on specific calls to a dependency, you can query its final state to see if the real effect you want happened.

-
- You can use them to run your application locally without spinning up or depending on real dependencies. This will usually improve developer experience (DX) because the fakes will be faster and more reliable than their real counterparts.

Spies, Mocks and Stubs can typically be autogenerated from an interface using a tool or using reflection. However, as Fakes encode the behaviour of the dependency you're trying to make a double for, you'll have to write at least most of the implementation yourself

The problem with stubs and mocks

In [Anti-patterns](#), there are details on how using test doubles must be done carefully. Creating a messy test suite is easy if you don't use them tastefully. As a project grows though, other problems can creep in.

When you encode behaviour into test doubles, you are adding your assumptions as to how the real dependency works into the test. If there is a discrepancy between the behaviour of the double and the real dependency, or if one happens over time (e.g. the real dependency changes, which has to be expected), **you may have passing tests but failing software.**

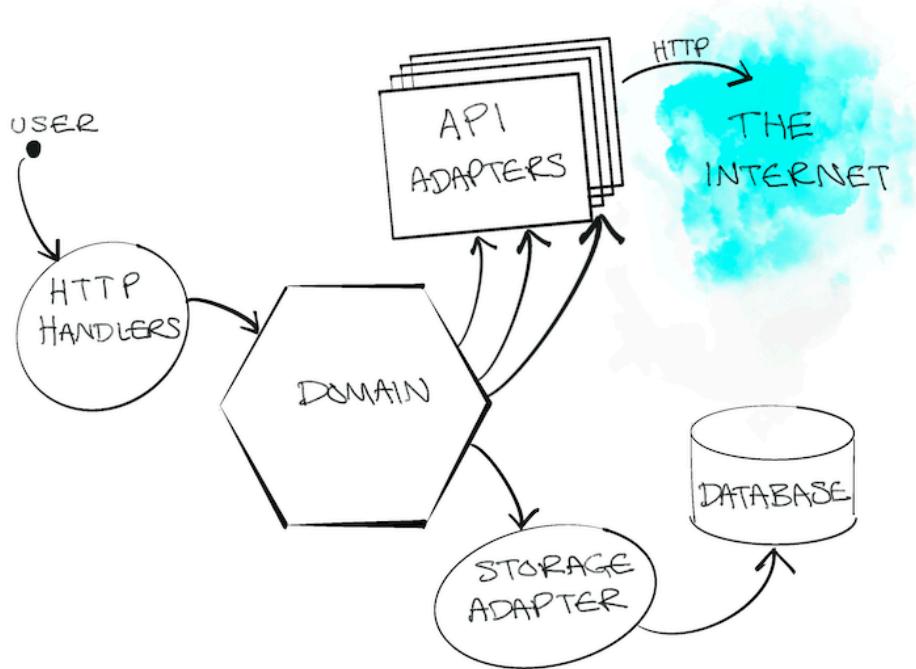
Stubs, spies and mocks, in particular, represent other challenges, mainly as a project grows. To illustrate this, I will describe a project I worked on.

Example case study

Some details are changed compared to what really happened, and it has been simplified greatly for brevity. **Any resemblance to actual persons, living or dead, is purely coincidental.**

I worked on a system that had to call **six** different APIs, written and maintained by other teams across the globe. They were REST-ish, and the job of our system was to create and manage resources in them all. When we called all the APIs correctly for each system, magic (business value) would happen.

Our application was structured in a hexagonal / ports & adapters architecture. Our domain code was decoupled from the mess of the outside world we had to deal with. Our "adapters" were, in effect, Go clients that encapsulated calling the various APIs.



Troubles Naturally, we took a test-driven approach to building the system. We leveraged stubs to simulate the downstream API responses and had a handful of acceptance tests to reassure ourselves everything should work.

The APIs we had to call for the most part, though, were:

- poorly documented
- run by teams who had lots of other conflicting priorities and pressures, so it wasn't easy to get time with them
- often lacking test coverage, so would break in fun and unexpected ways, regress, etc
- were still being built and evolved

This led to **a lot of flaky tests** and a lot of headaches. A significant amount of our time was spent pinging lots of busy people on Slack trying to get answers as to:

- Why has the API started doing x?
- Why is the API doing something different when we do y?

Software development is rarely as straightforward as you'd hope; it's a learning exercise. We had to continuously learn how the external APIs worked. As we learned and adapted, we had to update and add

to our test suite, in particular, **changing our stubs to match the actual behaviour of the APIs.**

The trouble is, this took up much of our time and led to more mistakes. When your knowledge of a dependency changes, you must find the **right** test to update to change the stub's behaviour, and there's a real risk of neglecting to update it in other stubs representing the same dependency.

Test strategy On top of this, as the system was growing and requirements were changing, we realised that our test strategy was unsuitable. We had a handful of acceptance tests that would give us confidence the system as a whole worked and then a large number of unit tests for the various packages we wrote.

We needed something in between; we often wanted to change the behaviour of various system parts together **but not have to spin up the entire system for an acceptance test.** Unit tests alone did not give us confidence that the various components worked as a whole; they couldn't tell (and verify) the story of what we were trying to achieve. **We wanted integration tests.**

Integration tests Integration tests prove that two or more "units" work correctly when combined (or integrated!). These units can be the code you write or the code you write integrated with someone else's code, such as a database.

As a project grows, you want to write more integration tests to prove large parts of your system "hang together" - or integrates!

You may be tempted to write more black-box acceptance tests, but they quickly become costly regarding your build time and maintenance costs. It can be too expensive to spin up an entire system when you only want to check a subset of the system (but not just a single unit) behaves how it should. Writing expensive black-box tests for every bit of functionality you do is not sustainable for larger systems.

Enter: Fakes The problem was the way our units were tested was reliant on stubs, which are, for the most part, stateless. We wanted to write tests covering multiple, stateful API calls, where we may create a resource at the start and then edit it later.

The following is a cut-down version of a test we want to do.

The SUT is a "service layer" dealing with "use case" requests. We want to prove if a customer is created, when their details change, we successfully update the resources we made in the respective APIs.

Here are the requirements given to the team as a user story.

Given a user is registered with API 1, 2 and 3

When the customer's social security number is changed

Then, the change is propagated into APIs 1, 2 and 3

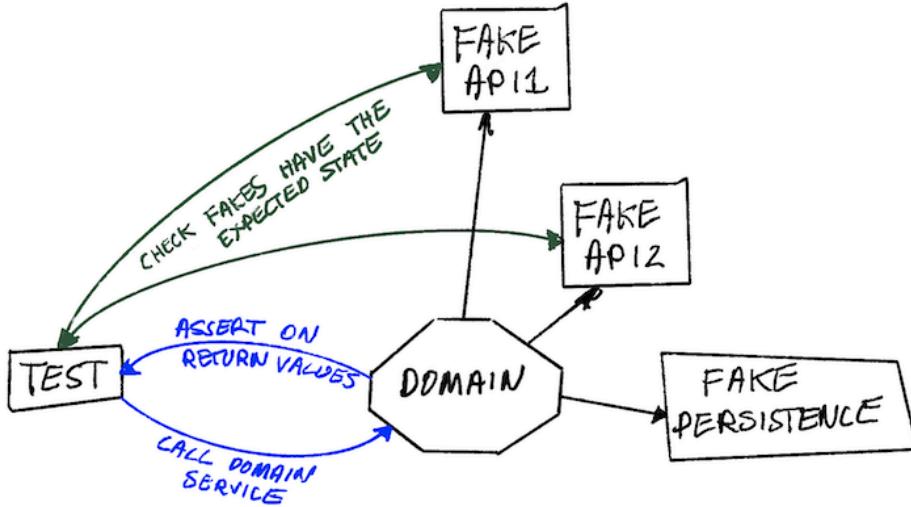
sequenceDiagram

```
User->>SUT: Create customer
SUT->>API1: Create resource for customer
API1->>SUT: Response with generated ID
SUT->>API2: Create resource for customer
API2->>SUT: Response with generated ID
SUT->>Storage: Persist identifiers for customer
User->>SUT: Change customer's social security number
SUT->>Storage: Get customer
Storage->>SUT: Details, including IDs generated by the APIs
SUT->>API1: Update resource
SUT->>API2: Update resource
```

Tests that cut across multiple units are usually incompatible with stubs **because they're not suited to maintaining state**. We could write a black-box acceptance test, but the costs of these tests would quickly spiral out of control.

In addition, it is complicated to test edge cases with a black-box test because you cannot control the dependencies. For instance, we wanted to prove that a rollback mechanism would be fired if one API call failed.

We needed to use **fakes**. By modelling our dependencies as stateful APIs with in-memory fakes, we were able to write integration tests with a much broader scope, **to allow us to test real use cases worked**, again without having to spin up the whole system, and instead have almost the same speed as unit tests.



Using fakes, **we can make assertions based on the final states of the respective systems rather than relying on complicated spying**. We'd ask each fake what records it held for the customer and assert they were updated. This feels more natural; if we manually checked our system, we would query those APIs to check their state, not inspect our request logs to see if we sent particular JSON payloads.

```
// take our lego-bricks and assemble the system for the test
fakeAPI1 := fakes.NewAPI1()
fakeAPI2 := fakes.NewAPI2() // etc..
customerService := customer.NewService(fakeAPI1, fakeAPI2, etc...)

// create new customer
newCustomerRequest := NewCustomerReq{
    // ...
}
createdCustomer, err := customerService.New(newCustomerRequest)
assert.NoErr(t, err)

// we can verify all the details are as expected in the various fakes in a natural way, as if they're
fakeAPI1Customer := fakeAPI1.Get(createdCustomer.FakeAPI1Details.ID)
assert.Equal(t, fakeAPI1Customer.SocialSecurityNumber, newCustomerRequest.SocialSecurityN

// repeat for the other apis we care about

// update customer
updatedCustomerRequest := NewUpdateReq{SocialSecurityNumber: "123", InternalID: created
assert.NoErr(t, customerService.Update(updatedCustomerRequest))
```

```
// again we can check the various fakes to see if the state ends up how we want it
updatedFakeAPICustomer := fakeAPI1.Get(createdCustomer.FakeAPI1Details.ID)
assert.Equal(t, updatedFakeAPICustomer.SocialSecurityNumber, updatedCustomerRequest.Soc
```

This is simpler to write and easier to read than checking various function call arguments made via spies.

This approach lets us have tests that cut across broad parts of our system, letting us write more **meaningful** tests about the use cases we'd be discussing at stand-up whilst still executing exceptionally quickly.

Fakes bring more of the benefits of encapsulation In the example above, the tests were not concerned with how the dependencies behaved beyond verifying their end state. We created the fake versions of the dependencies and injected them into the part of the system we're testing.

With mocks/stubs, we'd have to set up each dependency to handle certain scenarios, return certain data, etc. This brings behaviour and implementation detail into your tests, weakening the benefits of encapsulation.

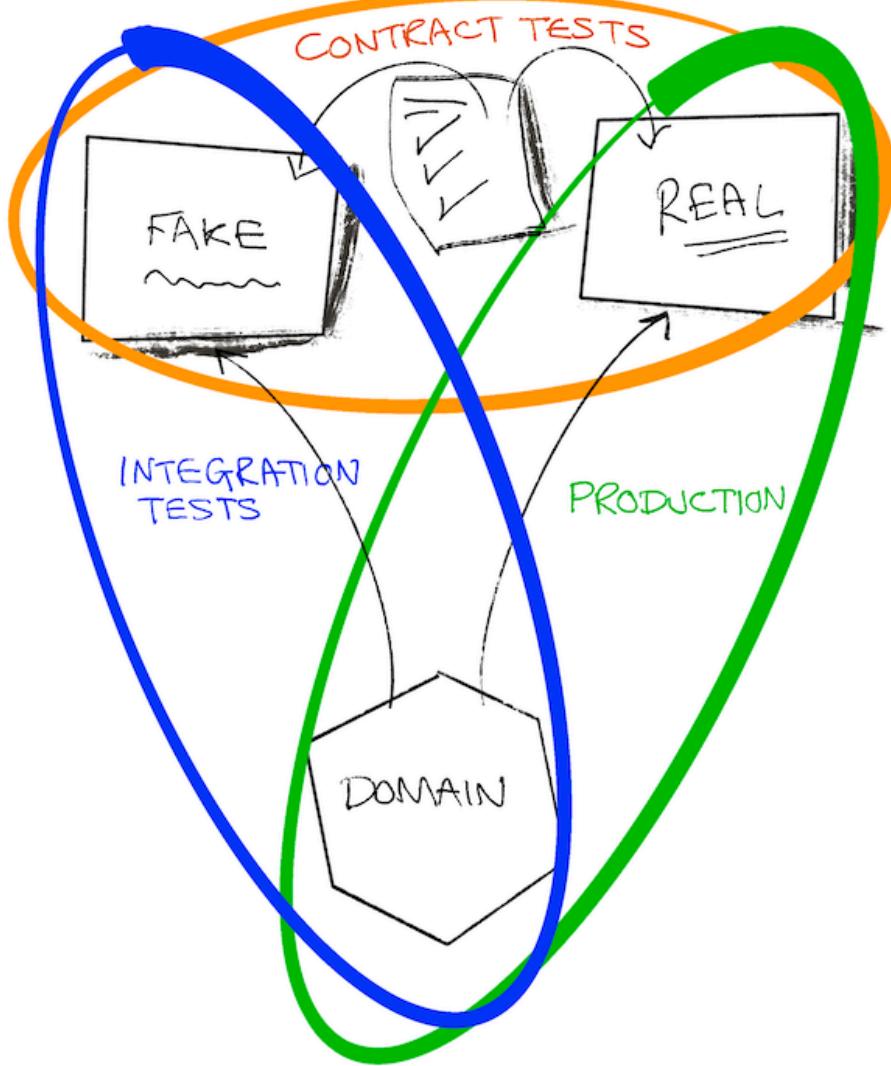
We model dependencies behind interfaces so that, as clients, we don't have to care how it works, but with a "mockist" approach, we do have to care **in every test**.

The maintenance costs of fakes Fakes are costlier than other test doubles, at least in terms of code written; they must carry state and simulate the behaviour of whatever they're faking. Any discrepancies in behaviour between your fake and the real thing **carry a risk** that your tests aren't in line with reality. This leads to the scenario where you have passing tests but broken software.

Whenever you integrate with another system, be it another team's API or a database, you'll make assumptions based on its behaviour. These could be captured from API docs, in-person conversations, emails, Slack threads, etc.

Wouldn't it be helpful if we could **codify our assumptions** to run them against both our fake and the actual system to see if our knowledge is correct in a repeatable and documented way?

Contracts are the means to this end. They helped us manage the assumptions we made on the other team's systems and make them explicit. Way more explicit and useful than email exchanges or endless Slack threads!



By having a contract, we can assume that we can use a fake and an actual dependency interchangeably. This is not only useful for constructing tests but also for local development.

Here is an example of a contract for one of the APIs the system depends on

```
type API1Customer struct {
    Name string
    ID   string
}
```

```
type API1 interface {
    CreateCustomer(ctx context.Context, name string) (API1Customer, error)
    GetCustomer(ctx context.Context, id string) (API1Customer, error)
    UpdateCustomer(ctx context.Context, id string, name string) error
}

type API1Contract struct {
    NewAPI1 func() API1
}

func (c API1Contract) Test(t *testing.T) {
    t.Run("can create, get and update a customer", func(t *testing.T) {
        var (
            ctx = context.Background()
            sut = c.NewAPI1()
            name = "Bob"
        )

        customer, err := sut.CreateCustomer(ctx, name)
        expect.NoError(t, err)

        got, err := sut.GetCustomer(ctx, customer.ID)
        expect.NoError(t, err)
        expect.Equal(t, customer, got)

        newName := "Robert"
        expect.NoError(t, sut.UpdateCustomer(ctx, customer.ID, newName))

        got, err = sut.GetCustomer(ctx, customer.ID)
        expect.NoError(t, err)
        expect.Equal(t, newName, got.Name)
    })

    // example of strange behaviours we didn't expect
    t.Run("the system will not allow you to add 'Dave' as a customer", func(t *testing.T) {
        var (
            ctx = context.Background()
            sut = c.NewAPI1()
            name = "Dave"
        )

        _, err := sut.CreateCustomer(ctx, name)
        expect.Error(t, ErrDavelsForbidden)
    })
}
```

As discussed in [Scaling Acceptance Tests](#), by testing against an interface rather than a concrete type, the test becomes:

- Decoupled from implementation detail
- Can be re-used in different contexts.

Which are the requirements for a contract. It allows us to verify and develop our fake and test it against the actual implementation.

To create our in-memory fake, we can use the contract in a test.

```
func TestInMemoryAPI1(t *testing.T) {
    API1Contract{NewAPI1: func() API1 {
        return inmemory.NewAPI1()
    }}.Test(t)
}
```

And here is the fake's code

```
func NewAPI1() *API1 {
    return &API1{customers: make(map[string]planner.API1Customer)}
}

type API1 struct {
    i      int
    customers map[string]planner.API1Customer
}

func (a *API1) CreateCustomer(ctx context.Context, name string) (planner.API1Customer, error) {
    if name == "Dave" {
        return planner.API1Customer{}, ErrDavelsForbidden
    }

    newCustomer := planner.API1Customer{
        Name: name,
        ID:  strconv.Itoa(a.i),
    }
    a.customers[newCustomer.ID] = newCustomer
    a.i++
    return newCustomer, nil
}

func (a *API1) GetCustomer(ctx context.Context, id string) (planner.API1Customer, error) {
    return a.customers[id], nil
}

func (a *API1) UpdateCustomer(ctx context.Context, id string, name string) error {
    customer := a.customers[id]
    customer.Name = name
    return nil
}
```

```
a.customers[id] = customer
return nil
}
```

Evolving software

Most software is not built and "finished" forever, in one release.

It's an incremental learning exercise, adapting to customer demands and other external changes. In the example, the APIs we were calling were also evolving and changing; plus, as we developed our software, we learned more about what system we really needed to make. Assumptions we made in our contracts turned out to be wrong or became wrong.

Thankfully, once the setup for the contracts was made, we had a simple way to deal with change. Once we learned something new, as a result of a bug being fixed or a colleague informing us that the API was changing, we'd:

1. Write a test to exercise the new scenario. A part of this will involve changing the contract to **drive** you to simulate the behaviour in the fake
2. Running the test should fail, but before anything else, run the contract against the real dependency to ensure the change to the contract is valid.
3. Update the fake so it conforms to the contract.
4. Make the test pass.
5. Refactor.
6. Run all the tests and ship.

Running the full test suite before checking in may result in other tests failing due to the fake having a different behaviour. This is a **good thing!** You can now fix all the other areas of the system depending on the changed system; confident they will also handle this scenario in production. Without this approach, you'd have to remember to find all the relevant tests and update the stubs. Error-prone, labourious and boring.

Superior developer experience

Having the suite of fakes with corresponding contracts felt like a superpower. We could finally tame the complexity of the APIs we had to deal with.

Writing tests for various scenarios became much simpler. We no longer had to assemble a series of stubs and spies for every test;

we could take our set of units or modules (the fakes, our own "services") and assemble them very easily to exercise the various weird and wonderful scenarios we needed.

Every test with a stub, spy or mock has to care about how the external system behaves, due to the ad-hoc setup. On the other hand, fakes can be treated like any other well-encapsulated unit of code, where the details are hidden away from you, and you can just use them.

We could run a very realistic version of the system locally, and as it was all in memory, it would start and run extremely quickly. This meant our test times were extremely fast, which felt very impressive, given how comprehensive the suite was.

If our acceptance tests failed in our staging environment, our first step was to run our contracts against the APIs we depended on. We often identified issues **before the other systems' developers did**.

Off the happy path with decorators

For error scenarios, stubs are more convenient because you have direct access to how it behaves in the test, whereas fakes tend to be fairly black-box. This is a deliberate design choice, as we want the users of them (e.g. tests) not to be concerned with how they work; they should trust they do the right thing due to the backing of the contract.

How do we make the fakes fail, to exercise non-happy path concerns?

There are plenty of scenarios where, as a developer, you need to modify the behaviour of some code without changing its source. The **decorator pattern** is often a way to take a unit of code and add things like logging, telemetry, retries and more. We can use it to wrap our fakes to override behaviours when necessary.

Returning to the API1 example, we can create a type that implements the needed interface and wraps around the fake.

```
type API1Decorator struct {
    delegate     API1
    CreateCustomerFunc func(ctx context.Context, name string) (API1Customer, error)
    GetCustomerFunc  func(ctx context.Context, id string) (API1Customer, error)
    UpdateCustomerFunc func(ctx context.Context, id string, name string) error
}

// assert API1Decorator implements API1
var _ API1 = &API1Decorator{}


func NewAPI1Decorator(delegate API1) *API1Decorator {
```

```
    return &API1Decorator{delegate: delegate}
}

func (a *API1Decorator) CreateCustomer(ctx context.Context, name string) (API1Customer, error) {
    if a.CreateCustomerFunc != nil {
        return a.CreateCustomerFunc(ctx, name)
    }
    return a.delegate.CreateCustomer(ctx, name)
}

func (a *API1Decorator) GetCustomer(ctx context.Context, id string) (API1Customer, error) {
    if a.GetCustomerFunc != nil {
        return a.GetCustomerFunc(ctx, id)
    }
    return a.delegate.GetCustomer(ctx, id)
}

func (a *API1Decorator) UpdateCustomer(ctx context.Context, id string, name string) error {
    if a.UpdateCustomerFunc != nil {
        return a.UpdateCustomerFunc(ctx, id, name)
    }
    return a.delegate.UpdateCustomer(ctx, id, name)
}
```

In our tests, we can then use the XXXFunc field to modify the behaviour of the test-double, just like you would with stubs, spies or mocks.

```
failingAPI1 = NewAPI1Decorator(inmemory.NewAPI1())
failingAPI1.UpdateCustomerFunc = func(ctx context.Context, id string, name string) error {
    return errors.New("failed to update customer")
}
```

However, this is awkward and requires you to exercise some judgement. With this approach, you are losing the guarantees from your contract as you are introducing ad-hoc behaviour to your fake in tests.

It would be best to examine your context, you may conclude it would be simpler to test specific unhappy paths at the unit test level using a stub.

Isn't this extra code waste?

It is wishful thinking to believe we should only ever write code that serves customers and expect a system we can build on efficiently. People have a very warped opinion of what waste is (see my post: [The ghost of Henry Ford is ruining your development team](#)).

Automated tests do not directly benefit customers, but we write them to make ourselves more efficient with our work (you don't write tests to chase coverage scores, right?).

Engineers must easily simulate scenarios (in a repeatable fashion, not ad-hocly) to debug, test, and fix issues. **In-memory fakes and good modular design allow us to isolate the relevant actors for a scenario to write fast, appropriate tests extremely cheaply.** This flexibility enables developers to iterate on a system far more manageably than a tangled mess, tested via expensive to-write and run black-box tests or, worse, manual testing on a shared environment.

This is an example of [simple vs. easy](#). Of course, fakes and contracts will result in more code being written than stubs and spies in the short term, but the result is a more straightforward and cheaper-to-maintain system in the longer run. Updating spies, stubs and mocks piecemeal is labour-intensive and error-prone, as you won't have corresponding contracts to check your test doubles behave correctly.

This approach represents a slightly increased upfront cost but with far lower costs once the contracts and fakes are set up. Fakes are more reusable and reliable than ad-hoc test doubles like stubs.

It feels very liberating and gives you **confidence** when using an existing, battle-tested fake rather than setting up a stub when writing a new test.

How does this fit into TDD?

I wouldn't recommend starting with a contract; that's bottom-up design, which, in general, I find I need to be more clever for, and there's a danger I'll overthink hypothetical requirements.

This technique is compatible with the "acceptance test driven approach" as discussed in earlier chapters, [The Why of TDD](#) and in [GOOS](#)

- Write a failing [acceptance test](#).
- Drive out enough code to make it pass, which usually will result in some "service layer" that'll depend on an API, a database, or whatever. Usually, you will have business logic code decoupled from external concerns (such as persistence, calling a database, etc.) via an interface.
- Implement the interface with an in-memory fake at first to make all the tests pass locally and validate the initial design.
- To push to production, you can't use in-memory! Encode the assumptions you made against the fake into a contract.

-
- Use the contract to create the actual dependency, such as a MySQL version of a store.
 - Ship.

Where's the chapter on testing databases?

This has been a common request that I have put off for over five years. The reason is this chapter will always be my answer.

Don't mock the database driver and spy on calls. These tests are difficult to write and potentially bring very little value. You shouldn't assert whether a particular SQL statement was sent to the database, that is, implementation detail; **your tests should only care about behaviour**. Proving a specific SQL statement was compiled does not prove your code behaves how you need it to.

Contracts force you to decouple your tests from implementation details and focus on behaviour.

Follow the TDD approach described above to drive out your persistence needs.

The example repository has some examples of contracts, and how they're used to test in-memory and SQLite implementations of some persistence needs.

```
package inmemory_test

import (
    "github.com/quii/go-fakes-and-contracts/adapters/driven/persistence/inmemory"
    "github.com/quii/go-fakes-and-contracts/domain/planner"
    "testing"
)

func TestInMemoryPantry(t *testing.T) {
    planner.PantryContract{
        NewPantry: func() planner.Pantry {
            return inmemory.NewPantry()
        },
    }.Test(t)
}

package sqlite_test

import (
    "github.com/quii/go-fakes-and-contracts/adapters/driven/persistence/sqlite"
    "github.com/quii/go-fakes-and-contracts/domain/planner"
    "testing"
)
```

```
)  
  
func TestSQLitePantry(t *testing.T) {  
    client := sqlite.NewSQLiteClient()  
    t.Cleanup(func() {  
        if err := client.Close(); err != nil {  
            t.Error(err)  
        }  
    })  
  
    planner.PantryContract{  
        NewPantry: func() planner.Pantry {  
            return sqlite.NewPantry(client)  
        },  
    }.Test(t)  
}
```

Whilst Docker et al. do make running databases locally easier, they can still carry a significant performance overhead. Fakes with contracts allow you to use restrict the need to use the "heavier" dependency to only when you're validating the contract, and not needed for other kinds of tests.

Using in-memory fakes for acceptance and integration tests for the rest of the system provides a much faster and simpler developer experience.

Wrapping up

It's common for software projects to be organised with various teams building systems concurrently to try to reach a common goal.

This method of work requires a high degree of collaboration and communication. Many feel with an "API first" approach, we can define some API contracts (often on a wiki page!) and then work independently for six months and stick it all together. This rarely works well in practice because as we start writing code, we understand the domain and the problem better, which challenges our assumptions. We have to react to these changes in knowledge, which often require cross-team changes.

So, if you're in this situation, you need to structure and test your system optimally to deal with unpredictable changes, both inside and outside of the system you're working on.

“One of the defining characteristics of high-performing teams in software development is their ability to make progress and to change their minds, without asking for

permission from any person or group outside of their small team.”

Modern Software Engineering David Farley

Don’t rely on weekly meetings or Slack threads to flesh out changes. **Codify your assumptions in contracts.** Run those contracts against the systems in your build pipelines so you get fast feedback if new information comes to light. These contracts, in conjunction with **fakes**, mean you can work independently and manage external changes sustainably.

Your system as a collection of modules

Referring back to Farley’s book, I’m describing the idea of **incrementalism**. Building software is a constant learning exercise. Understanding the requirements we must solve for a given system to deliver value up-front is unrealistic. So, we have to optimise our systems and ways of work to **gather feedback quickly and experiment**.

You need a **modular system** to take advantage of the ideas discussed in this chapter. If you have modular code with reliable fakes, it allows you to experiment with your system via automated tests cheaply.

We found it extremely easy to translate weird, hypothetical (but possible) scenarios into self-contained tests to help us understand the problem and drive out more robust software by composing our modules together and trying out different data in different order, with some APIs failing, etc.

Well-defined, well-tested modules allow you to increment your system without changing and understanding everything at once.

But I’m working on something small with stable APIs

Even with stable APIs, you do not want your developer experience, builds and so on to be tightly coupled to other people’s code. When you get this approach right, you end up with a composable set of modules to piece together your system for production, running locally and writing different kinds of tests with doubles you trust.

It allows you to isolate the parts of your system you’re concerned about and write meaningful tests about the real problem you’re trying to solve.

Make your dependencies first-class citizens.

Of course, stubs and spies have their place. Simulating different behaviours of your dependencies ad-hocly in tests will always have its use, but be careful not to let the costs go out of control.

So many times in my career, I have seen carefully written software written by talented devs fall apart due to integration problems. Integration is challenging for engineers because it's hard to reproduce the exact behaviours of a system written by other engineers, who also change it simultaneously.

Some teams rely on everyone deploying to a shared environment and testing there. The problem is this doesn't give you **isolated** feedback, and the **feedback is slow**. You still won't be able to construct different experiments with how your system works with other dependencies, at least not efficiently.

We have to tame this complexity by adopting more sophisticated ways of modelling our dependencies to quickly test/experiment on our dev machines before it gets to production. Create realistic and manageable fakes of your dependencies, verified by contracts. Then, you can start writing more meaningful tests and experimenting with your system, making you more likely to succeed.

Build an application

Now that you have hopefully digested the Go Fundamentals section you have a solid grounding of a majority of Go's language features and how to do TDD.

This next section will involve building an application.

Each chapter will iterate on the previous one, expanding the application's functionality as our product owner dictates.

New concepts will be introduced to help facilitate writing great code but most of the new material will be learning what can be accomplished from Go's standard library.

By the end of this, you should have a strong grasp as to how to iteratively write an application in Go, backed by tests.

- [HTTP server](#) - We will create an application which listens to HTTP requests and responds to them.
- [JSON, routing and embedding](#) - We will make our endpoints return JSON and explore how to do routing.
- [IO and sorting](#) - We will persist and read our data from disk and we'll cover sorting data.

-
- [Command line & project structure](#) - Support multiple applications from one code base and read input from command line.
 - [Time](#) - using the time package to schedule activities.
 - [WebSockets](#) - learn how to write and test a server that uses Web-Sockets.

HTTP Server

[You can find all the code for this chapter here](#)

You have been asked to create a web server where users can track how many games players have won.

- GET /players/{name} should return a number indicating the total number of wins
- POST /players/{name} should record a win for that name, incrementing for every subsequent POST

We will follow the TDD approach, getting working software as quickly as we can and then making small iterative improvements until we have the solution. By taking this approach we

- Keep the problem space small at any given time
- Don't go down rabbit holes
- If we ever get stuck/lost, doing a revert wouldn't lose loads of work.

Red, green, refactor

Throughout this book, we have emphasised the TDD process of write a test & watch it fail (red), write the minimal amount of code to make it work (green) and then refactor.

This discipline of writing the minimal amount of code is important in terms of the safety TDD gives you. You should be striving to get out of "red" as soon as you can.

Kent Beck describes it as:

Make the test work quickly, committing whatever sins necessary in process.

You can commit these sins because you will refactor afterwards backed by the safety of the tests.

What if you don't do this?

The more changes you make while in red, the more likely you are to add more problems, not covered by tests.

The idea is to be iteratively writing useful code with small steps, driven by tests so that you don't fall into a rabbit hole for hours.

Chicken and egg

How can we incrementally build this? We can't GET a player without having stored something and it seems hard to know if POST has worked without the GET endpoint already existing.

This is where mocking shines.

- GET will need a PlayerStore thing to get scores for a player. This should be an interface so when we test we can create a simple stub to test our code without needing to have implemented any actual storage code.
- For POST we can spy on its calls to PlayerStore to make sure it stores players correctly. Our implementation of saving won't be coupled to retrieval.
- For having some working software quickly we can make a very simple in-memory implementation and then later we can create an implementation backed by whatever storage mechanism we prefer.

Write the test first

We can write a test and make it pass by returning a hard-coded value to get us started. Kent Beck refers this as "Faking it". Once we have a working test we can then write more tests to help us remove that constant.

By doing this very small step, we can make the important start of getting an overall project structure working correctly without having to worry too much about our application logic.

To create a web server in Go you will typically call [ListenAndServe](#).

```
func ListenAndServe(addr string, handler Handler) error
```

This will start a web server listening on a port, creating a goroutine for every request and running it against a [Handler](#).

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

A type implements the Handler interface by implementing the ServeHTTP method which expects two arguments, the first is where we write our response and the second is the HTTP request that was sent to the server.

Let's create a file named server_test.go and write a test for a function PlayerServer that takes in those two arguments. The request sent in will be to get a player's score, which we expect to be "20".

```
func TestGETPlayers(t *testing.T) {
    t.Run("returns Pepper's score", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/players/Pepper", nil)
        response := httptest.NewRecorder()

        PlayerServer(response, request)

        got := response.Body.String()
        want := "20"

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

In order to test our server, we will need a Request to send in and we'll want to spy on what our handler writes to the ResponseWriter.

- We use http.NewRequest to create a request. The first argument is the request's method and the second is the request's path. The nil argument refers to the request's body, which we don't need to set in this case.
- net/http/httptest has a spy already made for us called ResponseRecorder so we can use that. It has many helpful methods to inspect what has been written as a response.

Try to run the test

```
./server_test.go:13:2: undefined: PlayerServer
```

Write the minimal amount of code for the test to run and check the failing test output

The compiler is here to help, just listen to it.

Create a file named server.go and define PlayerServer

```
func PlayerServer() {}
```

Try again

```
./server_test.go:13:14: too many arguments in call to PlayerServer
    have (*httptest.ResponseRecorder, *http.Request)
    want ()
```

Add the arguments to our function

```
import "net/http"

func PlayerServer(w http.ResponseWriter, r *http.Request) {  
}
```

The code now compiles and the test fails

```
==== RUN TestGETPlayers/returns_Pepper's_score
--- FAIL: TestGETPlayers/returns_Pepper's_score (0.00s)
    server_test.go:20: got "", want '20'
```

Write enough code to make it pass

From the DI chapter, we touched on HTTP servers with a Greet function. We learned that net/http's ResponseWriter also implements io.Writer so we can use fmt.Fprint to send strings as HTTP responses.

```
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "20")
}
```

The test should now pass.

Complete the scaffolding

We want to wire this up into an application. This is important because

- We'll have actual working software, we don't want to write tests for the sake of it, it's good to see the code in action.
- As we refactor our code, it's likely we will change the structure of the program. We want to make sure this is reflected in our application too as part of the incremental approach.

Create a new main.go file for our application and put this code in

```
package main
```

```
import (
    "log"
    "net/http"
)
```

```
func main() {
    handler := http.HandlerFunc(PlayerServer)
    log.Fatal(http.ListenAndServe(":5000", handler))
}
```

So far all of our application code has been in one file, however, this isn't best practice for larger projects where you'll want to separate things into different files.

To run this, do go build which will take all the .go files in the directory and build you a program. You can then execute it with ./myprogram.

http.HandlerFunc

Earlier we explored that the Handler interface is what we need to implement in order to make a server. Typically we do that by creating a struct and make it implement the interface by implementing its own ServeHTTP method. However the use-case for structs is for holding data but currently we have no state, so it doesn't feel right to be creating one.

[HandlerFunc](#) lets us avoid this.

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler that calls f.

type HandlerFunc **func**(ResponseWriter, *Request)

From the documentation, we see that type HandlerFunc has already implemented the ServeHTTP method. By type casting our PlayerServer function with it, we have now implemented the required Handler.

http.ListenAndServe(":5000"...)

ListenAndServe takes a port to listen on a Handler. If there is a problem the web server will return an error, an example of that might be the port already being listened to. For that reason we wrap the call in log.Fatal to log the error to the user.

What we're going to do now is write another test to force us into making a positive change to try and move away from the hard-coded value.

Write the test first

We'll add another subtest to our suite which tries to get the score of a different player, which will break our hard-coded approach.

```
t.Run("returns Floyd's score", func(t *testing.T) {
    request, _ := http.NewRequest(http.MethodGet, "/players/Floyd", nil)
    response := httptest.NewRecorder()

    PlayerServer(response, request)

    got := response.Body.String()
    want := "10"

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

You may have been thinking

Surely we need some kind of concept of storage to control which player gets what score. It's weird that the values seem so arbitrary in our tests.

Remember we are just trying to take as small as steps as reasonably possible, so we're just trying to break the constant for now.

Try to run the test

```
==== RUN TestGETPlayers/returns_Pepper's_score
--- PASS: TestGETPlayers/returns_Pepper's_score (0.00s)
==== RUN TestGETPlayers/returns_Floyd's_score
--- FAIL: TestGETPlayers/returns_Floyd's_score (0.00s)
    server_test.go:34: got '20', want '10'
```

Write enough code to make it pass

```
//server.go
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")

    if player == "Pepper" {
        fmt.Fprint(w, "20")
        return
    }
```

```
if player == "Floyd" {
    fmt.Fprint(w, "10")
    return
}
}
```

This test has forced us to actually look at the request's URL and make a decision. So whilst in our heads, we may have been worrying about player stores and interfaces the next logical step actually seems to be about routing.

If we had started with the store code the amount of changes we'd have to do would be very large compared to this. **This is a smaller step towards our final goal and was driven by tests.**

We're resisting the temptation to use any routing libraries right now, just the smallest step to get our test passing.

r.URL.Path returns the path of the request which we can then use `strings.TrimPrefix` to trim away /players/ to get the requested player. It's not very robust but will do the trick for now.

Refactor

We can simplify the PlayerServer by separating out the score retrieval into a function

```
/server.go
func PlayerServer(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")

    fmt.Fprint(w, GetPlayerScore(player))
}

func GetPlayerScore(name string) string {
    if name == "Pepper" {
        return "20"
    }

    if name == "Floyd" {
        return "10"
    }

    return ""
}
```

And we can DRY up some of the code in the tests by making some helpers

```
/server_test.go
func TestGETPlayers(t *testing.T) {
    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()

        PlayerServer(response, request)

        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        PlayerServer(response, request)

        assertResponseBody(t, response.Body.String(), "10")
    })
}

func newGetScoreRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), nil)
    return req
}

func assertResponseBody(t testing.TB, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("response body is wrong, got %q want %q", got, want)
    }
}
```

However, we still shouldn't be happy. It doesn't feel right that our server knows the scores.

Our refactoring has made it pretty clear what to do.

We moved the score calculation out of the main body of our handler into a function GetPlayerScore. This feels like the right place to separate the concerns using interfaces.

Let's move our function we re-factored to be an interface instead

```
type PlayerStore interface {
    GetPlayerScore(name string) int
}
```

For our PlayerServer to be able to use a PlayerStore, it will need a ref-

erence to one. Now feels like the right time to change our architecture so that our PlayerServer is now a struct.

```
type PlayerServer struct {
    store PlayerStore
}
```

Finally, we will now implement the Handler interface by adding a method to our new struct and putting in our existing handler code.

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

The only other change is we now call our store.GetPlayerScore to get the score, rather than the local function we defined (which we can now delete).

Here is the full code listing of our server

```
//server.go
type PlayerStore interface {
    GetPlayerScore(name string) int
}

type PlayerServer struct {
    store PlayerStore
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

Fix the issues

This was quite a few changes and we know our tests and application will no longer compile, but just relax and let the compiler work through it.

```
./main.go:9:58: type PlayerServer is not an expression
```

We need to change our tests to instead create a new instance of our PlayerServer and then call its method ServeHTTP.

```
//server_test.go
func TestGETPlayers(t *testing.T) {
    server := &PlayerServer{}
```

```
t.Run("returns Pepper's score", func(t *testing.T) {
    request := newGetScoreRequest("Pepper")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertResponseBody(t, response.Body.String(), "20")
})

t.Run("returns Floyd's score", func(t *testing.T) {
    request := newGetScoreRequest("Floyd")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertResponseBody(t, response.Body.String(), "10")
})
}
```

Notice we're still not worrying about making stores just yet, we just want the compiler passing as soon as we can.

You should be in the habit of prioritising having code that compiles and then code that passes the tests.

By adding more functionality (like stub stores) whilst the code isn't compiling, we are opening ourselves up to potentially more compilation problems.

Now main.go won't compile for the same reason.

```
func main() {
    server := &PlayerServer{}
    log.Fatal(http.ListenAndServe(":5000", server))
}
```

Finally, everything is compiling but the tests are failing

```
==== RUN TestGETPlayers/returns_the_Pepper's_score
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
    panic: runtime error: invalid memory address or nil pointer dereference
```

This is because we have not passed in a PlayerStore in our tests. We'll need to make a stub one up.

```
//server_test.go
type StubPlayerStore struct {
    scores map[string]int
}
```

```
func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}
```

A map is a quick and easy way of making a stub key/value store for our tests. Now let's create one of these stores for our tests and send it into our PlayerServer.

```
//server_test.go
func TestGETPlayers(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{
            "Pepper": 20,
            "Floyd": 10,
        },
    }
    server := &PlayerServer{&store}

    t.Run("returns Pepper's score", func(t *testing.T) {
        request := newGetScoreRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "20")
    })

    t.Run("returns Floyd's score", func(t *testing.T) {
        request := newGetScoreRequest("Floyd")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertResponseBody(t, response.Body.String(), "10")
    })
}
```

Our tests now pass and are looking better. The intent behind our code is clearer now due to the introduction of the store. We're telling the reader that because we have this data in a PlayerStore that when you use it with a PlayerServer you should get the following responses.

Run the application

Now our tests are passing the last thing we need to do to complete this refactor is to check if our application is working. The program

should start up but you'll get a horrible response if you try and hit the server at `http://localhost:5000/players/Pepper`.

The reason for this is that we have not passed in a PlayerStore.

We'll need to make an implementation of one, but that's difficult right now as we're not storing any meaningful data so it'll have to be hard-coded for the time being.

```
//main.go
type InMemoryPlayerStore struct{}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return 123
}

func main() {
    server := &PlayerServer{&InMemoryPlayerStore{}}
    log.Fatal(http.ListenAndServe(":5000", server))
}
```

If you run go build again and hit the same URL you should get "123". Not great, but until we store data that's the best we can do. It also didn't feel great that our main application was starting up but not actually working. We had to manually test to see the problem.

We have a few options as to what to do next

- Handle the scenario where the player doesn't exist
- Handle the POST /players/{name} scenario

Whilst the POST scenario gets us closer to the "happy path", I feel it'll be easier to tackle the missing player scenario first as we're in that context already. We'll get to the rest later.

Write the test first

Add a missing player scenario to our existing suite

```
/server_test.go
t.Run("returns 404 on missing players", func(t *testing.T) {
    request := newGetScoreRequest("Apollo")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    got := response.Code
    want := http.StatusNotFound
}
```

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

Try to run the test

```
==== RUN TestGETPlayers/returns_404_on_missing_players
--- FAIL: TestGETPlayers/returns_404_on_missing_players (0.00s)
server_test.go:56: got status 200 want 404
```

Write enough code to make it pass

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    w.WriteHeader(http.StatusNotFound)

    fmt.Fprint(w, p.store.GetPlayerScore(player))
}
```

Sometimes I heavily roll my eyes when TDD advocates say "make sure you just write the minimal amount of code to make it pass" as it can feel very pedantic.

But this scenario illustrates the example well. I have done the bare minimum (knowing it is not correct), which is write a StatusNotFound on **all responses** but all our tests are passing!

By doing the bare minimum to make the tests pass it can highlight gaps in your tests. In our case, we are not asserting that we should be getting a StatusOK when players do exist in the store.

Update the other two tests to assert on the status and fix the code.

Here are the new tests

```
//server_test.go
func TestGETPlayers(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{
            "Pepper": 20,
            "Floyd": 10,
        },
    }
    server := &PlayerServer{&store}
```

```
t.Run("returns Pepper's score", func(t *testing.T) {
    request := newGetScoreRequest("Pepper")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertStatus(t, response.Code, http.StatusOK)
    assertResponseBody(t, response.Body.String(), "20")
})

t.Run("returns Floyd's score", func(t *testing.T) {
    request := newGetScoreRequest("Floyd")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertStatus(t, response.Code, http.StatusOK)
    assertResponseBody(t, response.Body.String(), "10")
})

t.Run("returns 404 on missing players", func(t *testing.T) {
    request := newGetScoreRequest("Apollo")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertStatus(t, response.Code, http.StatusNotFound)
})
}

func assertStatus(t testing.TB, got, want int) {
    t.Helper()
    if got != want {
        t.Errorf("did not get correct status, got %d, want %d", got, want)
    }
}

func newGetScoreRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/players/%s", name), nil)
    return req
}

func assertResponseBody(t testing.TB, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("response body is wrong, got %q want %q", got, want)
    }
}
```

```
    }
}
```

We're checking the status in all our tests now so I made a helper assertStatus to facilitate that.

Now our first two tests fail because of the 404 instead of 200, so we can fix PlayerServer to only return not found if the score is 0.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}
```

Storing scores

Now that we can retrieve scores from a store it now makes sense to be able to store new scores.

Write the test first

```
//server_test.go
func TestStoreWins(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{},
    }
    server := &PlayerServer{&store}

    t.Run("it returns accepted on POST", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodPost, "/players/Pepper", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusAccepted)
    })
}
```

For a start let's just check we get the correct status code if we hit the particular route with POST. This lets us drive out the functionality of accepting a different kind of request and handling it differently to GET /players/{name}. Once this works we can then start asserting on our handler's interaction with the store.

Try to run the test

```
==== RUN TestStoreWins/it_returns_accepted_on_POST
--- FAIL: TestStoreWins/it_returns_accepted_on_POST (0.00s)
    server_test.go:70: did not get correct status, got 404, want 202
```

Write enough code to make it pass

Remember we are deliberately committing sins, so an if statement based on the request's method will do the trick.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    if r.Method == http.MethodPost {
        w.WriteHeader(http.StatusAccepted)
        return
    }

    player := strings.TrimPrefix(r.URL.Path, "/players/")
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}
```

Refactor

The handler is looking a bit muddled now. Let's break the code up to make it easier to follow and isolate the different functionality into new functions.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    switch r.Method {
```

```
        case http.MethodPost:
            p.processWin(w)
        case http.MethodGet:
            p.showScore(w, r)
    }

}

func (p *PlayerServer) showScore(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter) {
    w.WriteHeader(http.StatusAccepted)
}
```

This makes the routing aspect of ServeHTTP a bit clearer and means our next iterations on storing can just be inside processWin.

Next, we want to check that when we do our POST /players/{name} that our PlayerStore is told to record the win.

Write the test first

We can accomplish this by extending our StubPlayerStore with a new RecordWin method and then spy on its invocations.

```
//server_test.go
type StubPlayerStore struct {
    scores map[string]int
    winCalls []string
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}

func (s *StubPlayerStore) RecordWin(name string) {
```

```
    s.winCalls = append(s.winCalls, name)
}
```

Now extend our test to check the number of invocations for a start

```
//server_test.go
func TestStoreWins(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{},
    }
    server := &PlayerServer{&store}

    t.Run("it records wins when POST", func(t *testing.T) {
        request := newPostWinRequest("Pepper")
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusAccepted)

        if len(store.winCalls) != 1 {
            t.Errorf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
        }
    })
}

func newPostWinRequest(name string) *http.Request {
    req, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/players/%s", name), nil)
    return req
}
```

Try to run the test

```
./server_test.go:26:20: too few values in struct initializer
./server_test.go:65:20: too few values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

We need to update our code where we create a StubPlayerStore as we've added a new field

```
//server_test.go
store := StubPlayerStore{
    map[string]int{},
```

```
    nil,
}
--- FAIL: TestStoreWins (0.00s)
--- FAIL: TestStoreWins/it_records_wins_when_POST (0.00s)
    server_test.go:80: got 0 calls to RecordWin want 1
```

Write enough code to make it pass

As we're only asserting the number of calls rather than the specific values it makes our initial iteration a little smaller.

We need to update PlayerServer's idea of what a PlayerStore is by changing the interface if we're going to be able to call RecordWin.

```
//server.go
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
}
```

By doing this main no longer compiles

```
.main.go:17:46: cannot use InMemoryPlayerStore literal (type *InMemoryPlayerStore) as type P
    *InMemoryPlayerStore does not implement PlayerStore (missing RecordWin method)
```

The compiler tells us what's wrong. Let's update InMemoryPlayerStore to have that method.

```
//main.go
type InMemoryPlayerStore struct{}`

func (i *InMemoryPlayerStore) RecordWin(name string) {}
```

Try and run the tests and we should be back to compiling code - but the test is still failing.

Now that PlayerStore has RecordWin we can call it within our PlayerServer

```
//server.go
func (p *PlayerServer) processWin(w http.ResponseWriter) {
    p.store.RecordWin("Bob")
    w.WriteHeader(http.StatusAccepted)
}
```

Run the tests and it should be passing! Obviously "Bob" isn't exactly what we want to send to RecordWin, so let's further refine the test.

Write the test first

```
//server_test.go
func TestStoreWins(t *testing.T) {
    store := StubPlayerStore{
        map[string]int{},
        nil,
    }
    server := &PlayerServer{&store}

    t.Run("it records wins on POST", func(t *testing.T) {
        player := "Pepper"

        request := newPostWinRequest(player)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusAccepted)

        if len(store.winCalls) != 1 {
            t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
        }

        if store.winCalls[0] != player {
            t.Errorf("did not store correct winner got %q want %q", store.winCalls[0], player)
        }
    })
}
```

Now that we know there is one element in our `winCalls` slice we can safely reference the first one and check it is equal to `player`.

Try to run the test

```
==== RUN TestStoreWins/it_records_wins_on_POST
--- FAIL: TestStoreWins/it_records_wins_on_POST (0.00s)
    server_test.go:86: did not store correct winner got 'Bob' want 'Pepper'
```

Write enough code to make it pass

```
//server.go
func (p *PlayerServer) processWin(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    p.store.RecordWin(player)
```

```
        w.WriteHeader(http.StatusAccepted)
    }
```

We changed processWin to take http.Request so we can look at the URL to extract the player's name. Once we have that we can call our store with the correct value to make the test pass.

Refactor

We can DRY up this code a bit as we're extracting the player name the same way in two places

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
    switch r.Method {
        case http.MethodPost:
            p.processWin(w, player)
        case http.MethodGet:
            p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}
```

Even though our tests are passing we don't really have working software. If you try and run main and use the software as intended it doesn't work because we haven't got round to implementing PlayerStore correctly. This is fine though; by focusing on our handler we have identified the interface that we need, rather than trying to design it up-front.

We could start writing some tests around our InMemoryPlayerStore

but it's only here temporarily until we implement a more robust way of persisting player scores (i.e. a database).

What we'll do for now is write an integration test between our PlayerServer and InMemoryPlayerStore to finish off the functionality. This will let us get to our goal of being confident our application is working, without having to directly test InMemoryPlayerStore. Not only that, but when we get around to implementing PlayerStore with a database, we can test that implementation with the same integration test.

Integration tests

Integration tests can be useful for testing that larger areas of your system work but you must bear in mind:

- They are harder to write
- When they fail, it can be difficult to know why (usually it's a bug within a component of the integration test) and so can be harder to fix
- They are sometimes slower to run (as they often are used with "real" components, like a database)

For that reason, it is recommended that you research The Test Pyramid.

Write the test first

In the interest of brevity, I am going to show you the final refactored integration test.

```
// server_integration_test.go
package main

import (
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    store := InMemoryPlayerStore{}
    server := PlayerServer{&store}
    player := "Pepper"

    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
```

```
server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))

response := httptest.NewRecorder()
server.ServeHTTP(response, newGetScoreRequest(player))
assertStatus(t, response.Code, http.StatusOK)

assertResponseBody(t, response.Body.String(), "3")
}



- We are creating our two components we are trying to integrate with: InMemoryPlayerStore and PlayerServer.
- We then fire off 3 requests to record 3 wins for player. We're not too concerned about the status codes in this test as it's not relevant to whether they are integrating well.
- The next response we do care about (so we store a variable response) because we are going to try and get the player's score.

```

Try to run the test

```
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
    server_integration_test.go:24: response body is wrong, got '123' want '3'
```

Write enough code to make it pass

I am going to take some liberties here and write more code than you may be comfortable with without writing a test.

This is allowed! We still have a test checking things should be working correctly but it is not around the specific unit we're working with (InMemoryPlayerStore).

If I were to get stuck in this scenario, I would revert my changes back to the failing test and then write more specific unit tests around InMemoryPlayerStore to help me drive out a solution.

```
//in_memory_player_store.go
func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
    store map[string]int
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}
```

```
func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}
```

- We need to store the data so I've added a map[string]int to the InMemoryPlayerStore struct
- For convenience I've made NewInMemoryPlayerStore to initialise the store, and updated the integration test to use it:
`//server_integration_test.go`
store := NewInMemoryPlayerStore()
server := PlayerServer{store}
- The rest of the code is just wrapping around the map

The integration test passes, now we just need to change main to use NewInMemoryPlayerStore()

```
// main.go
package main

import (
    "log"
    "net/http"
)

func main() {
    server := &PlayerServer{NewInMemoryPlayerStore()}
    log.Fatal(http.ListenAndServe(":5000", server))
}
```

Build it, run it and then use curl to test it out.

- Run this a few times, change the player names if you like curl -X POST http://localhost:5000/players/Pepper
- Check scores with curl http://localhost:5000/players/Pepper

Great! You've made a REST-ish service. To take this forward you'd want to pick a data store to persist the scores longer than the length of time the program runs.

- Pick a store (Bolt? Mongo? Postgres? File system?)
- Make PostgresPlayerStore implement PlayerStore
- TDD the functionality so you're sure it works
- Plug it into the integration test, check it's still ok
- Finally plug it into main

Refactor

We are almost there! Lets take some effort to prevent concurrency errors like these

fatal error: concurrent map read and map write

By adding mutexes, we enforce concurrency safety especially for the counter in our RecordWin function. Read more about mutexes in the sync chapter.

Wrapping up

http.Handler

- Implement this interface to create web servers
- Use http.HandlerFunc to turn ordinary functions into http.Handlers
- Use httptest.NewRecorder to pass in as a ResponseWriter to let you spy on the responses your handler sends
- Use http.NewRequest to construct the requests you expect to come in to your system

Interfaces, Mocking and DI

- Lets you iteratively build the system up in smaller chunks
- Allows you to develop a handler that needs a storage without needing actual storage
- TDD to drive out the interfaces you need

Commit sins, then refactor (and then commit to source control)

- You need to treat having failing compilation or failing tests as a red situation that you need to get out of as soon as you can.
- Write just the necessary code to get there. Then refactor and make the code nice.
- By trying to do too many changes whilst the code isn't compiling or the tests are failing puts you at risk of compounding the problems.
- Sticking to this approach forces you to write small tests, which means small changes, which helps keep working on complex systems manageable.

JSON, routing & embedding

[You can find all the code for this chapter here](#)

In the previous chapter we created a web server to store how many games players have won.

Our product owner has a new requirement; to have a new endpoint called /league which returns a list of all players stored. She would like this to be returned as JSON.

Here is the code we have so far

```
// server.go
package main

import (
    "fmt"
    "net/http"
    "strings"
)

type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
}

type PlayerServer struct {
    store PlayerStore
}

func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")

    switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
        p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}
```

```
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}

// in_memory_player_store.go
package main

func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
    store map[string]int
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}

// main.go
package main

import (
    "log"
    "net/http"
)

func main() {
    server := &PlayerServer{NewInMemoryPlayerStore()}
    log.Fatal(http.ListenAndServe(":5000", server))
}
```

You can find the corresponding tests in the link at the top of the chapter.

We'll start by making the league table endpoint.

Write the test first

We'll extend the existing suite as we have some useful test functions and a fake PlayerStore to use.

```
//server_test.go
func TestLeague(t *testing.T) {
    store := StubPlayerStore{}
    server := &PlayerServer{&store}

    t.Run("it returns 200 on /league", func(t *testing.T) {
        request, _ := http.NewRequest(http.MethodGet, "/league", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusOK)
    })
}
```

Before worrying about actual scores and JSON we will try and keep the changes small with the plan to iterate toward our goal. The simplest start is to check we can hit /league and get an OK back.

Try to run the test

```
--- FAIL: TestLeague/it_returns_200_on_/league (0.00s)
    server_test.go:101: status code is wrong: got 404, want 200
FAIL
FAIL    playerstore 0.221s
FAIL
```

Our PlayerServer returns a 404 Not Found, as if we were trying to get the wins for an unknown player. Looking at how server.go implements ServeHTTP, we realize that it always assumes to be called with a URL pointing to a specific player:

```
player := strings.TrimPrefix(r.URL.Path, "/players/")
```

In the previous chapter, we mentioned this was a fairly naive way of doing our routing. Our test informs us correctly that we need a concept how to deal with different request paths.

Write enough code to make it pass

Go has a built-in routing mechanism called [ServeMux](#) (request multiplexer) which lets you attach http.Handlers to particular request paths.

Let's commit some sins and get the tests passing in the quickest way we can, knowing we can refactor it with safety once we know the tests are passing.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    router := http.NewServeMux()

    router.Handle("/league", http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
    }))

    router.Handle("/players/", http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        player := strings.TrimPrefix(r.URL.Path, "/players/")

        switch r.Method {
        case http.MethodPost:
            p.processWin(w, player)
        case http.MethodGet:
            p.showScore(w, player)
        }
    }))

    router.ServeHTTP(w, r)
}
```

- When the request starts we create a router and then we tell it for x path use y handler.
- So for our new endpoint, we use http.HandlerFunc and an anonymous function to w.WriteHeader(http.StatusOK) when /league is requested to make our new test pass.
- For the /players/ route we just cut and paste our code into another http.HandlerFunc.
- Finally, we handle the request that came in by calling our new router's ServeHTTP (notice how ServeMux is also an http.Handler?)

The tests should now pass.

Refactor

ServeHTTP is looking quite big, we can separate things out a bit by refactoring our handlers into separate methods.

```
//server.go
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
```

```
router := http.NewServeMux()
router.Handle("/league", http.HandlerFunc(p.leagueHandler))
router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    router.ServeHTTP(w, r)
}

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) playersHandler(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")
}

switch r.Method {
    case http.MethodPost:
        p.processWin(w, player)
    case http.MethodGet:
        p.showScore(w, player)
}
}
```

It's quite odd (and inefficient) to be setting up a router as a request comes in and then calling it. What we ideally want to do is have some kind of NewPlayerServer function which will take our dependencies and do the one-time setup of creating the router. Each request can then just use that one instance of the router.

```
/server.go
type PlayerServer struct {
    store PlayerStore
    router *http.ServeMux
}

func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := &PlayerServer{
        store,
        http.NewServeMux(),
    }

    p.router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    p.router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    return p
}
```

```
func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {  
    p.router.ServeHTTP(w, r)  
}
```

- PlayerServer now needs to store a router.
- We have moved the routing creation out of ServeHTTP and into our NewPlayerServer so this only has to be done once, not per request.
- You will need to update all the test and production code where we used to do PlayerServer{&store} with NewPlayerServer(&store).

One final refactor

Try changing the code to the following.

```
type PlayerServer struct {  
    store PlayerStore  
    http.Handler  
}  
  
func NewPlayerServer(store PlayerStore) *PlayerServer {  
    p := new(PlayerServer)  
  
    p.store = store  
  
    router := http.NewServeMux()  
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))  
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))  
  
    p.Handler = router  
  
    return p  
}
```

Then replace server := &PlayerServer{&store} with server := NewPlayerServer(&store) in server_test.go, server_integration_test.go, and main.go.

Finally make sure you **delete** func (p *PlayerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) as it is no longer needed!

Embedding

We changed the second property of PlayerServer, removing the named property router http.ServeMux and replaced it with http.Handler; this is called embedding.

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to “borrow” pieces of an implementation by embedding types within a struct or interface.

Effective Go - Embedding

What this means is that our PlayerServer now has all the methods that http.Handler has, which is just ServeHTTP.

To “fill in” the http.Handler we assign it to the router we create in NewPlayerServer. We can do this because http.ServeMux has the method ServeHTTP.

This lets us remove our own ServeHTTP method, as we are already exposing one via the embedded type.

Embedding is a very interesting language feature. You can use it with interfaces to compose new interfaces.

```
type Animal interface {
    Eater
    Sleeper
}
```

And you can use it with concrete types too, not just interfaces. As you’d expect if you embed a concrete type you’ll have access to all its public methods and fields.

Any downsides?

You must be careful with embedding types because you will expose all public methods and fields of the type you embed. In our case, it is ok because we embedded just the interface that we wanted to expose (http.Handler).

If we had been lazy and embedded http.ServeMux instead (the concrete type) it would still work but users of PlayerServer would be able to add new routes to our server because Handle(path, handler) would be public.

When embedding types, really think about what impact that has on your public API.

It is a very common mistake to misuse embedding and end up polluting your APIs and exposing the internals of your type.

Now we’ve restructured our application we can easily add new routes and have the start of the /league endpoint. We now need to make it return some useful information.

We should return some JSON that looks something like this.

```
[  
  {  
    "Name": "Bill",  
    "Wins": 10  
  },  
  {  
    "Name": "Alice",  
    "Wins": 15  
  }  
]
```

Write the test first

We'll start by trying to parse the response into something meaningful.

```
//server_test.go  
func TestLeague(t *testing.T) {  
    store := StubPlayerStore{}  
    server := NewPlayerServer(&store)  
  
    t.Run("it returns 200 on /league", func(t *testing.T) {  
        request, _ := http.NewRequest(http.MethodGet, "/league", nil)  
        response := httptest.NewRecorder()  
  
        server.ServeHTTP(response, request)  
  
        var got []Player  
  
        err := json.NewDecoder(response.Body).Decode(&got)  
  
        if err != nil {  
            t.Fatalf("Unable to parse response from server %q into slice of Player, '%v'", response.B...  
        }  
  
        assertStatus(t, response.Code, http.StatusOK)  
    })  
}
```

Why not test the JSON string?

You could argue a simpler initial step would be just to assert that the response body has a particular JSON string.

In my experience tests that assert against JSON strings have the following problems.

-
- Brittleness. If you change the data-model your tests will fail.
 - Hard to debug. It can be tricky to understand what the actual problem is when comparing two JSON strings.
 - Poor intention. Whilst the output should be JSON, what's really important is exactly what the data is, rather than how it's encoded.
 - Re-testing the standard library. There is no need to test how the standard library outputs JSON, it is already tested. Don't test other people's code.

Instead, we should look to parse the JSON into data structures that are relevant for us to test with.

Data modelling

Given the JSON data model, it looks like we need an array of Player with some fields so we have created a new type to capture this.

```
//server.go
type Player struct {
    Name string
    Wins int
}
```

JSON decoding

```
//server_test.go
var got []Player
err := json.NewDecoder(response.Body).Decode(&got)
```

To parse JSON into our data model we create a Decoder from encoding/json package and then call its Decode method. To create a Decoder it needs an io.Reader to read from which in our case is our response spy's Body.

Decode takes the address of the thing we are trying to decode into which is why we declare an empty slice of Player the line before.

Parsing JSON can fail so Decode can return an error. There's no point continuing the test if that fails so we check for the error and stop the test with t.Fatal if it happens. Notice that we print the response body along with the error as it's important for someone running the test to see what string cannot be parsed.

Try to run the test

```
== RUN TestLeague/it_returns_200_on_/league
```

```
--- FAIL: TestLeague/it_returns_200_on_league (0.00s)
    server_test.go:107: Unable to parse response from server "" into slice of Player, 'unexpected end of JSON input'
```

Our endpoint currently does not return a body so it cannot be parsed into JSON.

Write enough code to make it pass

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    leagueTable := []Player{
        {"Chris", 20},
    }

    json.NewEncoder(w).Encode(leagueTable)

    w.WriteHeader(http.StatusOK)
}
```

The test now passes.

Encoding and Decoding

Notice the lovely symmetry in the standard library.

- To create an Encoder you need an io.Writer which is what http.ResponseWriter implements.
- To create a Decoder you need an io.Reader which the Body field of our response spy implements.

Throughout this book, we have used io.Writer and this is another demonstration of its prevalence in the standard library and how a lot of libraries easily work with it.

Refactor

It would be nice to introduce a separation of concern between our handler and getting the leagueTable as we know we're going to not hard-code that very soon.

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(p.getLeagueTable())
    w.WriteHeader(http.StatusOK)
}

func (p *PlayerServer) getLeagueTable() []Player {
```

```
return []Player{
    {"Chris", 20},
}
}
```

Next, we'll want to extend our test so that we can control exactly what data we want back.

Write the test first

We can update the test to assert that the league table contains some players that we will stub in our store.

Update StubPlayerStore to let it store a league, which is just a slice of Player. We'll store our expected data in there.

```
//server_test.go
type StubPlayerStore struct {
    scores map[string]int
    winCalls []string
    league []Player
}
```

Next, update our current test by putting some players in the league property of our stub and assert they get returned from our server.

```
//server_test.go
func TestLeague(t *testing.T) {

    t.Run("it returns the league table as JSON", func(t *testing.T) {
        wantedLeague := []Player{
            {"Cleo", 32},
            {"Chris", 20},
            {"Tiest", 14},
        }

        store := StubPlayerStore{nil, nil, wantedLeague}
        server := NewPlayerServer(&store)

        request, _ := http.NewRequest(http.MethodGet, "/league", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        var got []Player

        err := json.NewDecoder(response.Body).Decode(&got)
    })
}
```

```
if err != nil {
    t.Fatalf("Unable to parse response from server %q into slice of Player, '%v'", response.Body)
}

assertStatus(t, response.Code, http.StatusOK)

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

Try to run the test

```
./server_test.go:33:3: too few values in struct initializer
./server_test.go:70:3: too few values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

You'll need to update the other tests as we have a new field in StubPlayerStore; set it to nil for the other tests.

Try running the tests again and you should get

```
==== RUN TestLeague/it_returns_the_league_table_as_JSON
--- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
    server_test.go:124: got [{Chris 20}] want [{Cleo 32} {Chris 20} {Tiest 14}]
```

Write enough code to make it pass

We know the data is in our StubPlayerStore and we've abstracted that away into an interface PlayerStore. We need to update this so anyone passing us in a PlayerStore can provide us with the data for leagues.

```
//server.go
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() []Player
}
```

Now we can update our handler code to call that rather than returning a hard-coded list. Delete our method getLeagueTable() and then update leagueHandler to call GetLeague().

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(p.store.GetLeague())
    w.WriteHeader(http.StatusOK)
}
```

Try and run the tests.

```
# github.com/quii/learn-go-with-tests/json-and-io/v4
./main.go:9:50: cannot use NewInMemoryPlayerStore() (type *InMemoryPlayerStore) as type PlayerStore
    *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_integration_test.go:11:27: cannot use store (type *InMemoryPlayerStore) as type PlayerStore
    *InMemoryPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:36:28: cannot use &store (type *StubPlayerStore) as type PlayerStore in argument to PlayerStore
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:74:28: cannot use &store (type *StubPlayerStore) as type PlayerStore in argument to PlayerStore
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
./server_test.go:106:29: cannot use &store (type *StubPlayerStore) as type PlayerStore in argument to PlayerStore
    *StubPlayerStore does not implement PlayerStore (missing GetLeague method)
```

The compiler is complaining because InMemoryPlayerStore and StubPlayerStore do not have the new method we added to our interface.

For StubPlayerStore it's pretty easy, just return the league field we added earlier.

```
//server_test.go
func (s *StubPlayerStore) GetLeague() []Player {
    return s.league
}
```

Here's a reminder of how InMemoryStore is implemented.

```
//in_memory_player_store.go
type InMemoryPlayerStore struct {
    store map[string]int
}
```

Whilst it would be pretty straightforward to implement GetLeague "properly" by iterating over the map remember we are just trying to write the minimal amount of code to make the tests pass.

So let's just get the compiler happy for now and live with the uncomfortable feeling of an incomplete implementation in our InMemoryStore.

```
//in_memory_player_store.go
func (i *InMemoryPlayerStore) GetLeague() []Player {
    return nil
}
```

What this is really telling us is that later we're going to want to test this but let's park that for now.

Try and run the tests, the compiler should pass and the tests should be passing!

Refactor

The test code does not convey our intent very well and has a lot of boilerplate we can refactor away.

```
//server_test.go
t.Run("it returns the league table as JSON", func(t *testing.T) {
    wantedLeague := []Player{
        {"Cleo", 32},
        {"Chris", 20},
        {"Tiest", 14},
    }

    store := StubPlayerStore{nil, nil, wantedLeague}
    server := NewPlayerServer(&store)

    request := newLeagueRequest()
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    got := getLeagueFromResponse(t, response.Body)
    assertStatus(t, response.Code, http.StatusOK)
    assertLeague(t, got, wantedLeague)
})
```

Here are the new helpers

```
//server_test.go
func getLeagueFromResponse(t testing.TB, body io.Reader) ([]Player) {
    t.Helper()
    err := json.NewDecoder(body).Decode(&league)

    if err != nil {
        t.Fatalf("Unable to parse response from server %q into slice of Player, '%v'", body, err)
    }

    return
}

func assertLeague(t testing.TB, got, want []Player) {
```

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

func newLeagueRequest() *http.Request {
    req, _ := http.NewRequest(http.MethodGet, "/league", nil)
    return req
}
```

One final thing we need to do for our server to work is make sure we return a content-type header in the response so machines can recognise we are returning JSON.

Write the test first

Add this assertion to the existing test

```
//server_test.go
if response.Result().Header.Get("content-type") != "application/json" {
    t.Errorf("response did not have content-type of application/json, got %v", response.Result().H
}
```

Try to run the test

```
==== RUN TestLeague/it_returns_the_league_table_as_JSON
--- FAIL: TestLeague/it_returns_the_league_table_as_JSON (0.00s)
    server_test.go:124: response did not have content-type of application/json, got map[Content-Type: application/json]
```

Write enough code to make it pass

Update leagueHandler

```
//server.go
func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("content-type", "application/json")
    json.NewEncoder(w).Encode(p.store.GetLeague())
}
```

The test should pass.

Refactor

Create a constant for "application/json" and use it in leagueHandler

```
//server.go
const jsonContentType = "application/json"

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("content-type", jsonContentType)
    json.NewEncoder(w).Encode(p.store.GetLeague())
}
```

Then add a helper for assertContentType.

```
//server_test.go
func assertContentType(t testing.TB, response *httptest.ResponseRecorder, want string) {
    t.Helper()
    if response.Result().Header.Get("content-type") != want {
        t.Errorf("response did not have content-type of %s, got %v", want, response.Result().Header.Get("content-type"))
    }
}
```

Use it in the test.

```
//server_test.go
assertContentType(t, response, jsonContentType)
```

Now that we have sorted out PlayerServer for now we can turn our attention to InMemoryPlayerStore because right now if we tried to demo this to the product owner /league will not work.

The quickest way for us to get some confidence is to add to our integration test, we can hit the new endpoint and check we get back the correct response from /league.

Write the test first

We can use t.Run to break up this test a bit and we can reuse the helpers from our server tests - again showing the importance of refactoring tests.

```
//server_integration_test.go
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    store := NewInMemoryPlayerStore()
    server := NewPlayerServer(store)
    player := "Pepper"

    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))
    server.ServeHTTP(httptest.NewRecorder(), newPostWinRequest(player))

    t.Run("get score", func(t *testing.T) {
        response := httptest.NewRecorder()
```

```
server.ServeHTTP(response, newGetScoreRequest(player))
assertStatus(t, response.Code, http.StatusOK)

    assertResponseBody(t, response.Body.String(), "3")
}

t.Run("get league", func(t *testing.T) {
    response := httptest.NewRecorder()
    server.ServeHTTP(response, newLeagueRequest())
    assertStatus(t, response.Code, http.StatusOK)

    got := getLeagueFromResponse(t, response.Body)
    want := []Player{
        {"Pepper", 3},
    }
    assertLeague(t, got, want)
})
})
```

Try to run the test

```
==== RUN TestRecordingWinsAndRetrievingThem/get_league
--- FAIL: TestRecordingWinsAndRetrievingThem/get_league (0.00s)
    server_integration_test.go:35: got [] want [{Pepper 3}]
```

Write enough code to make it pass

InMemoryPlayerStore is returning nil when you call GetLeague() so we'll need to fix that.

```
//in_memory_player_store.go
func (i *InMemoryPlayerStore) GetLeague() []Player {
    var league []Player
    for name, wins := range i.store {
        league = append(league, Player{name, wins})
    }
    return league
}
```

All we need to do is iterate over the map and convert each key/value to a Player.

The test should now pass.

Wrapping up

We've continued to safely iterate on our program using TDD, making it support new endpoints in a maintainable way with a router and it can now return JSON for our consumers. In the next chapter, we will cover persisting the data and sorting our league.

What we've covered:

- **Routing.** The standard library offers you an easy to use type to do routing. It fully embraces the `http.Handler` interface in that you assign routes to Handlers and the router itself is also a Handler. It does not have some features you might expect though such as path variables (e.g `/users/{id}`). You can easily parse this information yourself but you might want to consider looking at other routing libraries if it becomes a burden. Most of the popular ones stick to the standard library's philosophy of also implementing `http.Handler`.
- **Type embedding.** We touched a little on this technique but you can [learn more about it from Effective Go](#). If there is one thing you should take away from this is that it can be extremely useful but always thinking about your public API, only expose what's appropriate.
- **JSON deserializing and serializing.** The standard library makes it very trivial to serialise and deserialise your data. It is also open to configuration and you can customise how these data transformations work if necessary.

IO and sorting

[You can find all the code for this chapter here](#)

In the previous chapter we continued iterating on our application by adding a new endpoint `/league`. Along the way we learned about how to deal with JSON, embedding types and routing.

Our product owner is somewhat perturbed by the software losing the scores when the server was restarted. This is because our implementation of our store is in-memory. She is also not pleased that we didn't interpret the `/league` endpoint should return the players ordered by the number of wins!

The code so far

```
// server.go  
package main
```

```
import (
    "encoding/json"
    "fmt"
    "net/http"
    "strings"
)

// PlayerStore stores score information about players
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() []Player
}

// Player stores a name with a number of wins
type Player struct {
    Name string
    Wins int
}

// PlayerServer is a HTTP interface for player information
type PlayerServer struct {
    store PlayerStore
    http.Handler
}

const jsonContentType = "application/json"

// NewPlayerServer creates a PlayerServer with routing configured
func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := new(PlayerServer)

    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    p.Handler = router

    return p
}

func (p *PlayerServer) leagueHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("content-type", jsonContentType)
```

```
        json.NewEncoder(w).Encode(p.store.GetLeague())
    }

func (p *PlayerServer) playersHandler(w http.ResponseWriter, r *http.Request) {
    player := strings.TrimPrefix(r.URL.Path, "/players/")

    switch r.Method {
        case http.MethodPost:
            p.processWin(w, player)
        case http.MethodGet:
            p.showScore(w, player)
    }
}

func (p *PlayerServer) showScore(w http.ResponseWriter, player string) {
    score := p.store.GetPlayerScore(player)

    if score == 0 {
        w.WriteHeader(http.StatusNotFound)
    }

    fmt.Fprint(w, score)
}

func (p *PlayerServer) processWin(w http.ResponseWriter, player string) {
    p.store.RecordWin(player)
    w.WriteHeader(http.StatusAccepted)
}

// in_memory_player_store.go
package main

func NewInMemoryPlayerStore() *InMemoryPlayerStore {
    return &InMemoryPlayerStore{map[string]int{}}
}

type InMemoryPlayerStore struct {
    store map[string]int
}

func (i *InMemoryPlayerStore) GetLeague() []Player {
    var league []Player
    for name, wins := range i.store {
        league = append(league, Player{name, wins})
    }
    return league
}
```

```
}

func (i *InMemoryPlayerStore) RecordWin(name string) {
    i.store[name]++
}

func (i *InMemoryPlayerStore) GetPlayerScore(name string) int {
    return i.store[name]
}

// main.go
package main

import (
    "log"
    "net/http"
)

func main() {
    server := NewPlayerServer(NewInMemoryPlayerStore())
    log.Fatal(http.ListenAndServe(":5000", server))
}
```

You can find the corresponding tests in the link at the top of the chapter.

Store the data

There are dozens of databases we could use for this but we're going to go for a very simple approach. We're going to store the data for this application in a file as JSON.

This keeps the data very portable and is relatively simple to implement.

It won't scale especially well but given this is a prototype it'll be fine for now. If our circumstances change and it's no longer appropriate it'll be simple to swap it out for something different because of the PlayerStore abstraction we have used.

We will keep the InMemoryPlayerStore for now so that the integration tests keep passing as we develop our new store. Once we are confident our new implementation is sufficient to make the integration test pass we will swap it in and then delete InMemoryPlayerStore.

Write the test first

By now you should be familiar with the interfaces around the standard library for reading data (`io.Reader`), writing data (`io.Writer`) and how we can use the standard library to test these functions without having to use real files.

For this work to be complete we'll need to implement `PlayerStore` so we'll write tests for our store calling the methods we need to implement. We'll start with `GetLeague`.

```
//file_system_store_test.go
func TestFileSystemStore(t *testing.T) {

    t.Run("league from a reader", func(t *testing.T) {
        database := strings.NewReader(`[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)

        store := FileSystemPlayerStore{database}

        got := store.GetLeague()

        want := []Player{
            {"Cleo", 10},
            {"Chris", 33},
        }

        assertLeague(t, got, want)
    })
}
```

We're using `strings.NewReader` which will return us a Reader, which is what our `FileSystemPlayerStore` will use to read data. In main we will open a file, which is also a Reader.

Try to run the test

```
# github.com/quii/learn-go-with-tests/io/v1
./file_system_store_test.go:15:12: undefined: FileSystemPlayerStore
```

Write the minimal amount of code for the test to run and check the failing test output

Let's define `FileSystemPlayerStore` in a new file

```
//file_system_store.go
type FileSystemPlayerStore struct{}
```

Try again

```
# github.com/quii/learn-go-with-tests/io/v1
./file_system_store_test.go:15:28: too many values in struct initializer
./file_system_store_test.go:17:15: store.GetLeague undefined (type FileSystemPlayerStore has no field or method GetLeague)
```

It's complaining because we're passing in a Reader but not expecting one and it doesn't have GetLeague defined yet.

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database io.Reader
}

func (f *FileSystemPlayerStore) GetLeague() []Player {
    return nil
}
```

One more try...

```
==== RUN TestFileSystemStore//league_from_a_reader
--- FAIL: TestFileSystemStore//league_from_a_reader (0.00s)
    file_system_store_test.go:24: got [] want [{Cleo 10} {Chris 33}]
```

Write enough code to make it pass

We've read JSON from a reader before

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetLeague() []Player {
    var league []Player
    json.NewDecoder(f.database).Decode(&league)
    return league
}
```

The test should pass.

Refactor

We have done this before! Our test code for the server had to decode the JSON from the response.

Let's try DRYing this up into a function.

Create a new file called league.go and put this inside.

```
//league.go
func NewLeague(rdr io.Reader) ([]Player, error) {
    var league []Player
    err := json.NewDecoder(rdr).Decode(&league)
    if err != nil {
        err = fmt.Errorf("problem parsing league, %v", err)
    }

    return league, err
}
```

Call this in our implementation and in our test helper getLeagueFromResponse in server_test.go

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetLeague() []Player {
    league, _ := NewLeague(f.database)
    return league
}
```

We haven't got a strategy yet for dealing with parsing errors but let's press on.

Seeking problems

There is a flaw in our implementation. First of all, let's remind ourselves how io.Reader is defined.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

With our file, you can imagine it reading through byte by byte until the end. What happens if you try to Read a second time?

Add the following to the end of our current test.

```
//file_system_store_test.go
```

```
// read again
got = store.GetLeague()
assertLeague(t, got, want)
```

We want this to pass, but if you run the test it doesn't.

The problem is our Reader has reached the end so there is nothing more to read. We need a way to tell it to go back to the start.

[ReadSeeker](#) is another interface in the standard library that can help.

```
type ReadSeeker interface {
    Reader
    Seeker
}
```

Remember embedding? This is an interface comprised of Reader and Seeker

```
type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}
```

This sounds good, can we change FileSystemPlayerStore to take this interface instead?

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database io.ReadSeeker
}

func (f *FileSystemPlayerStore) GetLeague() []Player {
    f.database.Seek(0, 0)
    league, _ := NewLeague(f.database)
    return league
}
```

Try running the test, it now passes! Happily for us strings.NewReader that we used in our test also implements ReadSeeker so we didn't have to make any other changes.

Next we'll implement GetPlayerScore.

Write the test first

```
//file_system_store_test.go
t.Run("get player score", func(t *testing.T) {
    database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}`])

    store := FileSystemPlayerStore{database}

    got := store.GetPlayerScore("Chris")

    want := 33

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

```
    }
})
```

Try to run the test

```
./file_system_store_test.go:38:15: store.GetPlayerScore undefined (type FileSystemPlayerStore has no field or method GetPlayerScore)
```

Write the minimal amount of code for the test to run and check the failing test output

We need to add the method to our new type to get the test to compile.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
    return 0
}
```

Now it compiles and the test fails

```
==== RUN  TestFileSystemStore/get_player_score
--- FAIL: TestFileSystemStore//get_player_score (0.00s)
    file_system_store_test.go:43: got 0 want 33
```

Write enough code to make it pass

We can iterate over the league to find the player and return their score

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
    var wins int

    for _, player := range f.GetLeague() {
        if player.Name == name {
            wins = player.Wins
            break
        }
    }

    return wins
}
```

Refactor

You will have seen dozens of test helper refactorings so I'll leave this to you to make it work

```
//file_system_store_test.go
t.Run("get player score", func(t *testing.T) {
    database := strings.NewReader(`[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}`))

    store := FileSystemPlayerStore{database}

    got := store.GetPlayerScore("Chris")
    want := 33
    assertScoreEquals(t, got, want)
})
```

Finally, we need to start recording scores with RecordWin.

Write the test first

Our approach is fairly short-sighted for writes. We can't (easily) just update one "row" of JSON in a file. We'll need to store the whole new representation of our database on every write.

How do we write? We'd normally use a Writer but we already have our ReadSeeker. Potentially we could have two dependencies but the standard library already has an interface for us ReadWriteSeeker which lets us do all the things we'll need to do with a file.

Let's update our type

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database io.ReadWriteSeeker
}
```

See if it compiles

```
./file_system_store_test.go:15:34: cannot use database (type *strings.Reader) as type io.ReadW
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
./file_system_store_test.go:36:34: cannot use database (type *strings.Reader) as type io.ReadW
    *strings.Reader does not implement io.ReadWriteSeeker (missing Write method)
```

It's not too surprising that strings.Reader does not implement ReadWriteSeeker so what do we do?

We have two choices

- Create a temporary file for each test. *os.File implements ReadWriteSeeker. The pro of this is it becomes more of an integration test, we're really reading and writing from the file system so it will give us a very high level of confidence. The cons are we prefer unit tests because they are faster and generally

simpler. We will also need to do more work around creating temporary files and then making sure they're removed after the test.

- We could use a third party library. [Mattetti](#) has written a library [filebuffer](#) which implements the interface we need and doesn't touch the file system.

I don't think there's an especially wrong answer here, but by choosing to use a third party library I would have to explain dependency management! So we will use files instead.

Before adding our test we need to make our other tests compile by replacing the strings.Reader with an os.File.

Let's create some helper functions which will create a temporary file with some data inside it, and abstract our score tests

```
//file_system_store_test.go
func createTempFile(t testing.TB, initialData string) (io.ReadWriteSeeker, func() {
    t.Helper()

    tmpfile, err := os.CreateTemp("", "db")

    if err != nil {
        t.Fatalf("could not create temp file %v", err)
    }

    tmpfile.Write([]byte(initialData))

    removeFile := func() {
        tmpfile.Close()
        os.Remove(tmpfile.Name())
    }

    return tmpfile, removeFile
}

func assertScoreEquals(t testing.TB, got, want int) {
    t.Helper()
    if got != want {
        t.Errorf("got %d want %d", got, want)
    }
}
```

`CreateTemp` creates a temporary file for us to use. The "db" value we've passed in is a prefix put on a random file name it will create. This is to ensure it won't clash with other files by accident.

You'll notice we're not only returning our ReadWriteSeeker (the file)

but also a function. We need to make sure that the file is removed once the test is finished. We don't want to leak details of the files into the test as it's prone to error and uninteresting for the reader. By returning a removeFile function, we can take care of the details in our helper and all the caller has to do is run defer cleanDatabase().

```
//file_system_store_test.go
func TestFileSystemStore(t *testing.T) {

    t.Run("league from a reader", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store := FileSystemPlayerStore{database}

        got := store.GetLeague()

        want := []Player{
            {"Cleo", 10},
            {"Chris", 33},
        }

        assertLeague(t, got, want)

        // read again
        got = store.GetLeague()
        assertLeague(t, got, want)
    })

    t.Run("get player score", func(t *testing.T) {
        database, cleanDatabase := createTempFile(t, `[
            {"Name": "Cleo", "Wins": 10},
            {"Name": "Chris", "Wins": 33}]`)
        defer cleanDatabase()

        store := FileSystemPlayerStore{database}

        got := store.GetPlayerScore("Chris")
        want := 33
        assertScoreEquals(t, got, want)
    })
}
```

Run the tests and they should be passing! There were a fair amount of changes but now it feels like we have our interface definition complete

and it should be very easy to add new tests from now.

Let's get the first iteration of recording a win for an existing player

```
//file_system_store_test.go
t.Run("store wins for existing players", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
    defer cleanDatabase()

    store := FileSystemPlayerStore{database}

    store.RecordWin("Chris")

    got := store.GetPlayerScore("Chris")
    want := 34
    assertScoreEquals(t, got, want)
})
```

Try to run the test

```
./file_system_store_test.go:67:8: store.RecordWin undefined (type
FileSystemPlayerStore has no field or method RecordWin)
```

Write the minimal amount of code for the test to run and check the failing test output

Add the new method

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {
}

==== RUN TestFileSystemStore/store_wins_for_existing_players
--- FAIL: TestFileSystemStore/store_wins_for_existing_players (0.00s)
    file_system_store_test.go:71: got 33 want 34
```

Our implementation is empty so the old score is getting returned.

Write enough code to make it pass

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()
```

```
for i, player := range league {
    if player.Name == name {
        league[i].Wins++
    }
}

f.database.Seek(0, 0)
json.NewEncoder(f.database).Encode(league)
}
```

You may be asking yourself why I am doing `league[i].Wins++` rather than `player.Wins++`.

When you range over a slice you are returned the current index of the loop (in our case `i`) and a copy of the element at that index. Changing the `Wins` value of a copy won't have any effect on the `league` slice that we iterate on. For that reason, we need to get the reference to the actual value by doing `league[i]` and then changing that value instead.

If you run the tests, they should now be passing.

Refactor

In `GetPlayerScore` and `RecordWin`, we are iterating over `[]Player` to find a player by name.

We could refactor this common code in the internals of `FileSystemStore` but to me, it feels like this is maybe useful code we can lift into a new type. Working with a "League" so far has always been with `[]Player` but we can create a new type called `League`. This will be easier for other developers to understand and then we can attach useful methods onto that type for us to use.

Inside `league.go` add the following

```
//league.go
type League []Player

func (l League) Find(name string) *Player {
    for i, p := range l {
        if p.Name == name {
            return &l[i]
        }
    }
    return nil
}
```

Now if anyone has a `League` they can easily find a given player.

Change our PlayerStore interface to return League rather than []Player. Try to re-run the tests, you'll get a compilation problem because we've changed the interface but it's very easy to fix; just change the return type from []Player to League.

This lets us simplify our methods in file_system_store.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
    player := f.GetLeague().Find(name)

    if player != nil {
        return player.Wins
    }

    return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()
    player := league.Find(name)

    if player != nil {
        player.Wins++
    }

    f.database.Seek(0, 0)
    json.NewEncoder(f.database).Encode(league)
}
```

This is looking much better and we can see how we might be able to find other useful functionality around League that can be refactored.

We now need to handle the scenario of recording wins of new players.

Write the test first

```
//file_system_store_test.go
t.Run("store wins for new players", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[
        {"Name": "Cleo", "Wins": 10},
        {"Name": "Chris", "Wins": 33}]`)
    defer cleanDatabase()

    store := FileSystemPlayerStore{database}
```

```
    store.RecordWin("Pepper")

    got := store.GetPlayerScore("Pepper")
    want := 1
    assertScoreEquals(t, got, want)
)}
```

Try to run the test

```
==== RUN TestFileSystemStore/store_wins_for_new_players#01
--- FAIL: TestFileSystemStore/store_wins_for_new_players#01 (0.00s)
    file_system_store_test.go:86: got 0 want 1
```

Write enough code to make it pass

We just need to handle the scenario where Find returns nil because it couldn't find the player.

```
//file_system_store.go
func (f *FileSystemPlayerStore) RecordWin(name string) {
    league := f.GetLeague()
    player := league.Find(name)

    if player != nil {
        player.Wins++
    } else {
        league = append(league, Player{name, 1})
    }

    f.database.Seek(0, 0)
    json.NewEncoder(f.database).Encode(league)
}
```

The happy path is looking ok so we can now try using our new Store in the integration test. This will give us more confidence that the software works and then we can delete the redundant InMemoryPlayerStore.

In TestRecordingWinsAndRetrievingThem replace the old store.

```
//server_integration_test.go
database, cleanDatabase := createTempFile(t, "")
defer cleanDatabase()
store := &FileSystemPlayerStore{database}
```

If you run the test it should pass and now we can delete InMemoryPlayerStore. main.go will now have compilation problems which will motivate us to now use our new store in the "real" code.

```
// main.go
package main

import (
    "log"
    "net/http"
    "os"
)

const dbFileName = "game.db.json"

func main() {
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store := &FileSystemPlayerStore{db}
    server := NewPlayerServer(store)

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

- We create a file for our database.
- The 2nd argument to os.OpenFile lets you define the permissions for opening the file, in our case O_RDWR means we want to read and write and os.O_CREATE means create the file if it doesn't exist.
- The 3rd argument means sets permissions for the file, in our case, all users can read and write the file. ([See superuser.com for a more detailed explanation](#)).

Running the program now persists the data in a file in between restarts, hooray!

More refactoring and performance concerns

Every time someone calls GetLeague() or GetPlayerScore() we are reading the entire file and parsing it into JSON. We should not have to do that because FileSystemStore is entirely responsible for the state of the league; it should only need to read the file when the program starts up and only need to update the file when data changes.

We can create a constructor which can do some of this initialisation

for us and store the league as a value in our FileSystemStore to be used on the reads instead.

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database io.ReadWriteSeeker
    league   League
}

func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
    database.Seek(0, 0)
    league, _ := NewLeague(database)
    return &FileSystemPlayerStore{
        database: database,
        league:   league,
    }
}
```

This way we only have to read from disk once. We can now replace all of our previous calls to getting the league from disk and just use f.league instead.

```
//file_system_store.go
func (f *FileSystemPlayerStore) GetLeague() League {
    return f.league
}

func (f *FileSystemPlayerStore) GetPlayerScore(name string) int {
    player := f.league.Find(name)

    if player != nil {
        return player.Wins
    }

    return 0
}

func (f *FileSystemPlayerStore) RecordWin(name string) {
    player := f.league.Find(name)

    if player != nil {
        player.Wins++
    } else {
        f.league = append(f.league, Player{name, 1})
    }
}
```

```
f.database.Seek(0, 0)
json.NewEncoder(f.database).Encode(f.league)
}
```

If you try to run the tests it will now complain about initialising FileSystemPlayerStore so just fix them by calling our new constructor.

Another problem

There is some more naivety in the way we are dealing with files which could create a very nasty bug down the line.

When we RecordWin, we Seek back to the start of the file and then write the new data—but what if the new data was smaller than what was there before?

In our current case, this is impossible. We never edit or delete scores so the data can only get bigger. However, it would be irresponsible for us to leave the code like this; it's not unthinkable that a delete scenario could come up.

How will we test for this though? What we need to do is first refactor our code so we separate out the concern of the kind of data we write, from the writing. We can then test that separately to check it works how we hope.

We'll create a new type to encapsulate our "when we write we go from the beginning" functionality. I'm going to call it Tape. Create a new file with the following:

```
// tape.go
package main

import "io"

type tape struct {
    file io.ReadWriteSeeker
}

func (t *tape) Write(p []byte) (n int, err error) {
    t.file.Seek(0, 0)
    return t.file.Write(p)
}
```

Notice that we're only implementing Write now, as it encapsulates the Seek part. This means our FileSystemStore can just have a reference to a Writer instead.

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database io.Writer
    league   League
}

Update the constructor to use Tape

//file_system_store.go
func NewFileSystemPlayerStore(database io.ReadWriteSeeker) *FileSystemPlayerStore {
    database.Seek(0, 0)
    league, _ := NewLeague(database)

    return &FileSystemPlayerStore{
        database: &tape{database},
        league:   league,
    }
}
```

Finally, we can get the amazing payoff we wanted by removing the Seek call from RecordWin. Yes, it doesn't feel much, but at least it means if we do any other kind of writes we can rely on our Write to behave how we need it to. Plus it will now let us test the potentially problematic code separately and fix it.

Let's write the test where we want to update the entire contents of a file with something that is smaller than the original contents.

Write the test first

Our test will create a file with some content, try to write to it using the tape, and read it all again to see what's in the file. In tape_test.go:

```
//tape_test.go
func TestTape_Write(t *testing.T) {
    file, clean := createTempFile(t, "12345")
    defer clean()

    tape := &tape{file}

    tape.Write([]byte("abc"))

    file.Seek(0, 0)
    newFileContents, _ := io.ReadAll(file)

    got := string(newFileContents)
    want := "abc"
```

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

Try to run the test

```
==== RUN TestTape_Write
--- FAIL: TestTape_Write (0.00s)
    tape_test.go:23: got 'abc45' want 'abc'
```

As we thought! It writes the data we want, but leaves the rest of the original data remaining.

Write enough code to make it pass

`os.File` has a `truncate` function that will let us effectively empty the file. We should be able to just call this to get what we want.

Change `tape` to the following:

```
//tape.go
type tape struct {
    file *os.File
}

func (t *tape) Write(p []byte) (n int, err error) {
    t.file.Truncate(0)
    t.file.Seek(0, 0)
    return t.file.Write(p)
}
```

The compiler will fail in a number of places where we are expecting an `io.ReadWriteSeeker` but we are sending in `*os.File`. You should be able to fix these problems yourself by now but if you get stuck just check the source code.

Once you get it refactoring our `TestTape_Write` test should be passing!

One other small refactor

In `RecordWin` we have the line `json.NewEncoder(f.database).Encode(f.league)`.

We don't need to create a new encoder every time we write, we can initialise one in our constructor and use that instead.

Store a reference to an `Encoder` in our type and initialise it in the constructor:

```
//file_system_store.go
type FileSystemPlayerStore struct {
    database *json.Encoder
    league  League
}

func NewFileSystemPlayerStore(file *os.File) *FileSystemPlayerStore {
    file.Seek(0, 0)
    league, _ := NewLeague(file)

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }
}
```

Use it in RecordWin.

```
func (f *FileSystemPlayerStore) RecordWin(name string) {
    player := f.league.Find(name)

    if player != nil {
        player.Wins++
    } else {
        f.league = append(f.league, Player{name, 1})
    }

    f.database.Encode(f.league)
}
```

Didn't we just break some rules there? Testing private things? No interfaces?

On testing private types

It's true that in general you should favour not testing private things as that can sometimes lead to your tests being too tightly coupled to the implementation, which can hinder refactoring in future.

However, we must not forget that tests should give us confidence.

We were not confident that our implementation would work if we added any kind of edit or delete functionality. We did not want to leave the code like that, especially if this was being worked on by more than one person who may not be aware of the shortcomings of our initial approach.

Finally, it's just one test! If we decide to change the way it works it

won't be a disaster to just delete the test but we have at the very least captured the requirement for future maintainers.

Interfaces

We started off the code by using `io.Reader` as that was the easiest path for us to unit test our new `PlayerStore`. As we developed the code we moved on to `io.ReadWriter` and then `io.ReadWriteSeeker`. We then found out there was nothing in the standard library that actually implemented that apart from `*os.File`. We could've taken the decision to write our own or use an open source one but it felt pragmatic just to make temporary files for the tests.

Finally, we needed `Truncate` which is also on `*os.File`. It would've been an option to create our own interface capturing these requirements.

```
type ReadWriteSeekTruncate interface {
    io.ReadWriteSeeker
    Truncate(size int64) error
}
```

But what is this really giving us? Bear in mind we are not mocking and it is unrealistic for a **file system** store to take any type other than an `*os.File` so we don't need the polymorphism that interfaces give us.

Don't be afraid to chop and change types and experiment like we have here. The great thing about using a statically typed language is the compiler will help you with every change.

Error handling

Before we start working on sorting we should make sure we're happy with our current code and remove any technical debt we may have. It's an important principle to get to working software as quickly as possible (stay out of the red state) but that doesn't mean we should ignore error cases!

If we go back to `FileSystemStore.go` we have `league, _ := NewLeague(f.database)` in our constructor.

`NewLeague` can return an error if it is unable to parse the league from the `io.Reader` that we provide.

It was pragmatic to ignore that at the time as we already had failing tests. If we had tried to tackle it at the same time, we would have been juggling two things at once.

Let's make it so our constructor is capable of returning an error.

```
//file_system_store.go
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {
    file.Seek(0, 0)
    league, err := NewLeague(file)

    if err != nil {
        return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name(), err)
    }

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }, nil
}
```

Remember it is very important to give helpful error messages (just like your tests). People on the internet jokingly say that most Go code is:

```
if err != nil {
    return err
}
```

That is 100% not idiomatic. Adding contextual information (i.e what you were doing to cause the error) to your error messages makes operating your software far easier.

If you try to compile you'll get some errors.

```
./main.go:18:35: multiple-value NewFileSystemPlayerStore() in single-value context
./file_system_store_test.go:35:36: multiple-value NewFileSystemPlayerStore() in single-value context
./file_system_store_test.go:57:36: multiple-value NewFileSystemPlayerStore() in single-value context
./file_system_store_test.go:70:36: multiple-value NewFileSystemPlayerStore() in single-value context
./file_system_store_test.go:85:36: multiple-value NewFileSystemPlayerStore() in single-value context
./server_integration_test.go:12:35: multiple-value NewFileSystemPlayerStore() in single-value context
```

In main we'll want to exit the program, printing the error.

```
//main.go
store, err := NewFileSystemPlayerStore(db)

if err != nil {
    log.Fatalf("problem creating file system player store, %v ", err)
}
```

In the tests we should assert there is no error. We can make a helper to help with this.

```
//file_system_store_test.go
func assertNoError(t testing.TB, err error) {
    t.Helper()
```

```
if err != nil {
    t.Fatalf("didn't expect an error but got one, %v", err)
}
}
```

Work through the other compilation problems using this helper. Finally, you should have a failing test:

```
==== RUN TestRecordingWinsAndRetrievingThem
--- FAIL: TestRecordingWinsAndRetrievingThem (0.00s)
    server_integration_test.go:14: didn't expect an error but got one, problem loading player stor
```

We cannot parse the league because the file is empty. We weren't getting errors before because we always just ignored them.

Let's fix our big integration test by putting some valid JSON in it:

```
//server_integration_test.go
func TestRecordingWinsAndRetrievingThem(t *testing.T) {
    database, cleanDatabase := createTempFile(t, `[])
    //etc...
}
```

Now that all the tests are passing, we need to handle the scenario where the file is empty.

Write the test first

```
//file_system_store_test.go
t.Run("works with an empty file", func(t *testing.T) {
    database, cleanDatabase := createTempFile(t, "")
    defer cleanDatabase()

    _, err := NewFileSystemPlayerStore(database)
    assert.NoError(t, err)
})
```

Try to run the test

```
==== RUN TestFileSystemStore/works_with_an_empty_file
--- FAIL: TestFileSystemStore/works_with_an_empty_file (0.00s)
    file_system_store_test.go:108: didn't expect an error but got one, problem loading player s
```

Write enough code to make it pass

Change our constructor to the following

```
//file_system_store.go
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {

    file.Seek(0, 0)

    info, err := file.Stat()

    if err != nil {
        return nil, fmt.Errorf("problem getting file info from file %s, %v", file.Name(), err)
    }

    if info.Size() == 0 {
        file.Write([]byte("[]"))
        file.Seek(0, 0)
    }
}

league, err := NewLeague(file)

if err != nil {
    return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name(), err)
}

return &FileSystemPlayerStore{
    database: json.NewEncoder(&tape{file}),
    league: league,
}, nil
}
```

file.Stat returns stats on our file, which lets us check the size of the file. If it's empty, we Write an empty JSON array and Seek back to the start, ready for the rest of the code.

Refactor

Our constructor is a bit messy now, so let's extract the initialise code into a function:

```
//file_system_store.go
func initialisePlayerDBFile(file *os.File) error {
    file.Seek(0, 0)

    info, err := file.Stat()

    if err != nil {
        return fmt.Errorf("problem getting file info from file %s, %v", file.Name(), err)
    }
```

```
if info.Size() == 0 {
    file.Write([]byte("[]"))
    file.Seek(0, 0)
}

return nil
}

//file_system_store.go
func NewFileSystemPlayerStore(file *os.File) (*FileSystemPlayerStore, error) {
    err := initialisePlayerDBFile(file)

    if err != nil {
        return nil, fmt.Errorf("problem initialising player db file, %v", err)
    }

    league, err := NewLeague(file)

    if err != nil {
        return nil, fmt.Errorf("problem loading player store from file %s, %v", file.Name(), err)
    }

    return &FileSystemPlayerStore{
        database: json.NewEncoder(&tape{file}),
        league:   league,
    }, nil
}
```

Sorting

Our product owner wants /league to return the players sorted by their scores, from highest to lowest.

The main decision to make here is where in the software should this happen. If we were using a "real" database we would use things like ORDER BY so the sorting is super fast. For that reason, it feels like implementations of PlayerStore should be responsible.

Write the test first

We can update the assertion on our first test in TestFileSystemStore:

```
//file_system_store_test.go
t.Run("league sorted", func(t *testing.T) {
```

```
database, cleanDatabase := createTempFile(t, `[
    {"Name": "Cleo", "Wins": 10},
    {"Name": "Chris", "Wins": 33}`)
defer cleanDatabase()

store, err := NewFileSystemPlayerStore(database)
assert.NoError(t, err)

got := store.GetLeague()

want := League{
    {"Chris", 33},
    {"Cleo", 10},
}

assertLeague(t, got, want)

// read again
got = store.GetLeague()
assertLeague(t, got, want)
})
```

The order of the JSON coming in is in the wrong order and our want will check that it is returned to the caller in the correct order.

Try to run the test

```
==== RUN TestFileSystemStore/league_from_a_reader,_sorted
--- FAIL: TestFileSystemStore/league_from_a_reader,_sorted (0.00s)
    file_system_store_test.go:46: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cleo 10}]
    file_system_store_test.go:51: got [{Cleo 10} {Chris 33}] want [{Chris 33} {Cleo 10}]
```

Write enough code to make it pass

```
func (f *FileSystemPlayerStore) GetLeague() League {
    sort.Slice(f.league, func(i, j int) bool {
        return f.league[i].Wins > f.league[j].Wins
    })
    return f.league
}
```

sort.Slice

Slice sorts the provided slice given the provided less function.

Easy!

Wrapping up

What we've covered

- The Seeker interface and its relation to Reader and Writer.
- Working with files.
- Creating an easy to use helper for testing with files that hides all the messy stuff.
- sort.Slice for sorting slices.
- Using the compiler to help us safely make structural changes to the application.

Breaking rules

- Most rules in software engineering aren't really rules, just best practices that work 80% of the time.
- We discovered a scenario where one of our previous "rules" of not testing internal functions was not helpful for us so we broke the rule.
- It's important when breaking rules to understand the trade-off you are making. In our case, we were ok with it because it was just one test and would've been very difficult to exercise the scenario otherwise.
- In order to be able to break the rules **you must understand them first**. An analogy is with learning guitar. It doesn't matter how creative you think you are, you must understand and practice the fundamentals.

Where our software is at

- We have an HTTP API where you can create players and increment their score.
- We can return a league of everyone's scores as JSON.
- The data is persisted as a JSON file.

Command line and project structure

You can find all the code for this chapter here

Our product owner now wants to pivot by introducing a second application - a command line application.

For now, it will just need to be able to record a player's win when the user types Ruth wins. The intention is to eventually be a tool for helping users play poker.

The product owner wants the database to be shared amongst the two applications so that the league updates according to wins recorded in the new application.

A reminder of the code

We have an application with a main.go file that launches an HTTP server. The HTTP server won't be interesting to us for this exercise but the abstraction it uses will. It depends on a PlayerStore.

```
type PlayerStore interface {
    GetPlayerScore(name string) int
    RecordWin(name string)
    GetLeague() League
}
```

In the previous chapter, we made a FileSystemPlayerStore which implements that interface. We should be able to re-use some of this for our new application.

Some project refactoring first

Our project now needs to create two binaries, our existing web server and the command line app.

Before we get stuck into our new work we should structure our project to accommodate this.

So far all the code has lived in one folder, in a path looking like this

```
$GOPATH/src/github.com/your-name/my-app
```

In order for you to make an application in Go, you need a main function inside a package main. So far all of our "domain" code has lived inside package main and our func main can reference everything.

This was fine so far and it is good practice not to go over-the-top with package structure. If you take the time to look through the standard library you will see very little in the way of lots of folders and structure.

Thankfully it's pretty straightforward to add structure when you need it.

Inside the existing project create a cmd directory with a webserver directory inside that (e.g mkdir -p cmd/webserver).

Move the main.go inside there.

If you have tree installed you should run it and your structure should look like this

```
.  
|-- file_system_store.go  
|-- file_system_store_test.go  
|-- cmd  
|   |-- webserver  
|       |-- main.go  
|-- league.go  
|-- server.go  
|-- server_integration_test.go  
|-- server_test.go  
|-- tape.go  
|-- tape_test.go
```

We now effectively have a separation between our application and the library code but we now need to change some package names. Remember when you build a Go application its package must be main.

Change all the other code to have a package called poker.

Finally, we need to import this package into main.go so we can use it to create our web server. Then we can use our library code by using poker.FunctionName.

The paths will be different on your computer, but it should be similar to this:

```
// cmd/webserver/main.go  
package main  
  
import (  
    "github.com/quii/learn-go-with-tests/command-line/v1"  
    "log"  
    "net/http"  
    "os"  
)  
  
const dbFileName = "game.db.json"  
  
func main() {  
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)  
  
    if err != nil {  
        log.Fatalf("problem opening %s %v", dbFileName, err)  
    }
```

```
store, err := poker.NewFileSystemPlayerStore(db)

if err != nil {
    log.Fatalf("problem creating file system player store, %v", err)
}

server := poker.NewPlayerServer(store)

log.Fatal(http.ListenAndServe(":5000", server))
}
```

The full path may seem a bit jarring, but this is how you can import any publicly available library into your code.

By separating our domain code into a separate package and committing it to a public repo like GitHub any Go developer can write their own code which imports that package the features we've written available. The first time you try and run it will complain it is not existing but all you need to do is run go get.

In addition, users can view [the documentation at pkg.go.dev](#).

Final checks

- Inside the root run go test and check they're still passing
- Go inside our cmd/webserver and do go run main.go
 - Visit <http://localhost:5000/league> and you should see it's still working

Walking skeleton

Before we get stuck into writing tests, let's add a new application that our project will build. Create another directory inside cmd called cli (command line interface) and add a main.go with the following

```
// cmd/cli/main.go
package main

import "fmt"

func main() {
    fmt.Println("Let's play poker")
}
```

The first requirement we'll tackle is recording a win when the user types {PlayerName} wins.

Write the test first

We know we need to make something called CLI which will allow us to Play poker. It'll need to read user input and then record wins to a PlayerStore.

Before we jump too far ahead though, let's just write a test to check it integrates with the PlayerStore how we'd like.

Inside CLI_test.go (in the root of the project, not inside cmd)

```
// CLI_test.go
package poker

import "testing"

func TestCLI(t *testing.T) {
    playerStore := &StubPlayerStore{}
    cli := &CLI{playerStore}
    cli.PlayPoker()

    if len(playerStore.winCalls) != 1 {
        t.Fatal("expected a win call but didn't get any")
    }
}
```

- We can use our StubPlayerStore from other tests
- We pass in our dependency into our not yet existing CLI type
- Trigger the game by an unwritten PlayPoker method
- Check that a win is recorded

Try to run the test

```
# github.com/quii/learn-go-with-tests/command-line/v2
./cli_test.go:25:10: undefined: CLI
```

Write the minimal amount of code for the test to run and check the failing test output

At this point, you should be comfortable enough to create our new CLI struct with the respective field for our dependency and add a method.

You should end up with code like this

```
// CLI.go
package poker

type CLI struct {
```

```
    playerStore PlayerStore
}
```

```
func (cli *CLI) PlayPoker() {}
```

Remember we're just trying to get the test running so we can check the test fails how we'd hope

```
--- FAIL: TestCLI (0.00s)
  cli_test.go:30: expected a win call but didn't get any
FAIL
```

Write enough code to make it pass

```
//CLI.go
func (cli *CLI) PlayPoker() {
    cli.playerStore.RecordWin("Cleo")
}
```

That should make it pass.

Next, we need to simulate reading from Stdin (the input from the user) so that we can record wins for specific players.

Let's extend our test to exercise this.

Write the test first

```
//CLI_test.go
func TestCLI(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &StubPlayerStore{}

    cli := &CLI{playerStore, in}
    cli.PlayPoker()

    if len(playerStore.winCalls) != 1 {
        t.Fatal("expected a win call but didn't get any")
    }

    got := playerStore.winCalls[0]
    want := "Chris"

    if got != want {
        t.Errorf("didn't record correct winner, got %q, want %q", got, want)
    }
}
```

`os.Stdin` is what we'll use in `main` to capture the user's input. It is a `*File` under the hood which means it implements `io.Reader` which as we know by now is a handy way of capturing text.

We create an `io.Reader` in our test using the handy `strings.NewReader`, filling it with what we expect the user to type.

Try to run the test

```
./CLI_test.go:12:32: too many values in struct initializer
```

Write the minimal amount of code for the test to run and check the failing test output

We need to add our new dependency into `CLI`.

```
//CLI.go
type CLI struct {
    playerStore PlayerStore
    in         io.Reader
}
```

Write enough code to make it pass

```
--- FAIL: TestCLI (0.00s)
    CLI_test.go:23: didn't record the correct winner, got 'Cleo', want 'Chris'
FAIL
```

Remember to do the strictly easiest thing first

```
func (cli *CLI) PlayPoker() {
    cli.playerStore.RecordWin("Chris")
}
```

The test passes. We'll add another test to force us to write some real code next, but first, let's refactor.

Refactor

In `server_test` we earlier did checks to see if wins are recorded as we have here. Let's DRY that assertion up into a helper

```
//server_test.go
func assertPlayerWin(t testing.TB, store *StubPlayerStore, winner string) {
    t.Helper()

    if len(store.winCalls) != 1 {
```

```
t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
}

if store.winCalls[0] != winner {
    t.Errorf("did not store correct winner got %q want %q", store.winCalls[0], winner)
}
}
```

Now replace the assertions in both server_test.go and CLI_test.go.

The test should now read like so

```
//CLI_test.go
func TestCLI(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &StubPlayerStore{}

    cli := &CLI{playerStore, in}
    cli.PlayPoker()

    assertPlayerWin(t, playerStore, "Chris")
}
```

Now let's write another test with different user input to force us into actually reading it.

Write the test first

```
//CLI_test.go
func TestCLI(t *testing.T) {

    t.Run("record chris win from user input", func(t *testing.T) {
        in := strings.NewReader("Chris wins\n")
        playerStore := &StubPlayerStore{}

        cli := &CLI{playerStore, in}
        cli.PlayPoker()

        assertPlayerWin(t, playerStore, "Chris")
    })

    t.Run("record cleo win from user input", func(t *testing.T) {
        in := strings.NewReader("Cleo wins\n")
        playerStore := &StubPlayerStore{}

        cli := &CLI{playerStore, in}
        cli.PlayPoker()
    })
}
```

```
        assertPlayerWin(t, playerStore, "Cleo")
    })
}
```

Try to run the test

```
==== RUN TestCLI
--- FAIL: TestCLI (0.00s)
==== RUN TestCLI/record_chris_win_from_user_input
    --- PASS: TestCLI/record_chris_win_from_user_input (0.00s)
==== RUN TestCLI/record_cleo_win_from_user_input
    --- FAIL: TestCLI/record_cleo_win_from_user_input (0.00s)
        CLI_test.go:27: did not store correct winner got 'Chris' want 'Cleo'
FAIL
```

Write enough code to make it pass

We'll use a [bufio.Scanner](#) to read the input from the `io.Reader`.

Package `bufio` implements buffered I/O. It wraps an `io.Reader` or `io.Writer` object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.

Update the code to the following

```
//CLI.go
type CLI struct {
    playerStore PlayerStore
    in         io.Reader
}

func (cli *CLI) PlayPoker() {
    reader := bufio.NewScanner(cli.in)
    reader.Scan()
    cli.playerStore.RecordWin(extractWinner(reader.Text()))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}
```

The tests will now pass.

- `Scanner.Scan()` will read up to a newline.

-
- We then use Scanner.Text() to return the string the scanner read to.

Now that we have some passing tests, we should wire this up into main. Remember we should always strive to have fully-integrated working software as quickly as we can.

In main.go add the following and run it. (you may have to adjust the path of the second dependency to match what's on your computer)

```
package main

import (
    "fmt"
    "github.com/quii/learn-go-with-tests/command-line/v3"
    "log"
    "os"
)

const dbFileName = "game.db.json"

func main() {
    fmt.Println("Let's play poker")
    fmt.Println("Type {Name} wins to record a win")

    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store, err := poker.NewFileSystemPlayerStore(db)

    if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
    }

    game := poker.CLI{store, os.Stdin}
    game.PlayPoker()
}
```

You should get an error

```
command-line/v3/cmd/cli/main.go:32:25: implicit assignment of unexported field 'playerStore' in
command-line/v3/cmd/cli/main.go:32:34: implicit assignment of unexported field 'in' in poker.CLI
```

What's happening here is because we are trying to assign to the fields playerStore and in in CLI. These are unexported (private) fields. We could do this in our test code because our test is in the same package

as CLI (poker). But our main is in package main so it does not have access.

This highlights the importance of integrating your work. We rightfully made the dependencies of our CLI private (because we don't want them exposed to users of CLIs) but haven't made a way for users to construct it.

Is there a way to have caught this problem earlier?

package mypackage_test

In all other examples so far, when we make a test file we declare it as being in the same package that we are testing.

This is fine and it means on the odd occasion where we want to test something internal to the package we have access to the unexported types.

But given we have advocated for not testing internal things generally, can Go help enforce that? What if we could test our code where we only have access to the exported types (like our main does)?

When you're writing a project with multiple packages I would strongly recommend that your test package name has _test at the end. When you do this you will only be able to have access to the public types in your package. This would help with this specific case but also helps enforce the discipline of only testing public APIs. If you still wish to test internals you can make a separate test with the package you want to test.

An adage with TDD is that if you cannot test your code then it is probably hard for users of your code to integrate with it. Using package foo_test will help with this by forcing you to test your code as if you are importing it like users of your package will.

Before fixing main let's change the package of our test inside CLI_test.go to poker_test.

If you have a well-configured IDE you will suddenly see a lot of red! If you run the compiler you'll get the following errors

```
./CLI_test.go:12:19: undefined: StubPlayerStore
./CLI_test.go:17:3: undefined: assertPlayerWin
./CLI_test.go:22:19: undefined: StubPlayerStore
./CLI_test.go:27:3: undefined: assertPlayerWin
```

We have now stumbled into more questions on package design. In order to test our software we made unexported stubs and helper functions which are no longer available for us to use in our CLI_test be-

cause the helpers are defined in the _test.go files in the poker package.

Do we want to have our stubs and helpers 'public'? This is a subjective discussion. One could argue that you do not want to pollute your package's API with code to facilitate tests.

In the presentation "[Advanced Testing with Go](#)" by Mitchell Hashimoto, it is described how at HashiCorp they advocate doing this so that users of the package can write tests without having to re-invent the wheel writing stubs. In our case, this would mean anyone using our poker package won't have to create their own stub PlayerStore if they wish to work with our code.

Anecdotally I have used this technique in other shared packages and it has proved extremely useful in terms of users saving time when integrating with our packages.

So let's create a file called testing.go and add our stub and our helpers.

```
// testing.go
package poker

import "testing"

type StubPlayerStore struct {
    scores map[string]int
    winCalls []string
    league []Player
}

func (s *StubPlayerStore) GetPlayerScore(name string) int {
    score := s.scores[name]
    return score
}

func (s *StubPlayerStore) RecordWin(name string) {
    s.winCalls = append(s.winCalls, name)
}

func (s *StubPlayerStore) GetLeague() League {
    return s.league
}

func AssertPlayerWin(t testing.TB, store *StubPlayerStore, winner string) {
    t.Helper()
}
```

```
if len(store.winCalls) != 1 {
    t.Fatalf("got %d calls to RecordWin want %d", len(store.winCalls), 1)
}

if store.winCalls[0] != winner {
    t.Errorf("did not store correct winner got %q want %q", store.winCalls[0], winner)
}
}

// todo for you - the rest of the helpers
```

You'll need to make the helpers public (remember exporting is done with a capital letter at the start) if you want them to be exposed to importers of our package.

In our CLI test you'll need to call the code as if you were using it within a different package.

```
/CLI_test.go
func TestCLI(t *testing.T) {

    t.Run("record chris win from user input", func(t *testing.T) {
        in := strings.NewReader("Chris wins\n")
        playerStore := &poker.StubPlayerStore{}

        cli := &poker.CLI{playerStore, in}
        cli.PlayPoker()

        poker.AssertPlayerWin(t, playerStore, "Chris")
    })

    t.Run("record cleo win from user input", func(t *testing.T) {
        in := strings.NewReader("Cleo wins\n")
        playerStore := &poker.StubPlayerStore{}

        cli := &poker.CLI{playerStore, in}
        cli.PlayPoker()

        poker.AssertPlayerWin(t, playerStore, "Cleo")
    })
}
```

You'll now see we have the same problems as we had in main

```
./CLI_test.go:15:26: implicit assignment of unexported field 'playerStore' in poker.CLI literal
./CLI_test.go:15:39: implicit assignment of unexported field 'in' in poker.CLI literal
```

```
./CLI_test.go:25:26: implicit assignment of unexported field 'playerStore' in poker.CLI literal
./CLI_test.go:25:39: implicit assignment of unexported field 'in' in poker.CLI literal
```

The easiest way to get around this is to make a constructor as we have for other types. We'll also change CLI so it stores a bufio.Scanner instead of the reader as it's now automatically wrapped at construction time.

```
//CLI.go
type CLI struct {
    playerStore PlayerStore
    in          *bufio.Scanner
}

func NewCLI(store PlayerStore, in io.Reader) *CLI {
    return &CLI{
        playerStore: store,
        in:          bufio.NewScanner(in),
    }
}
```

By doing this, we can then simplify and refactor our reading code

```
//CLI.go
func (cli *CLI) PlayPoker() {
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

Change the test to use the constructor instead and we should be back to the tests passing.

Finally, we can go back to our new main.go and use the constructor we just made

```
//cmd/cli/main.go
game := poker.NewCLI(store, os.Stdin)
```

Try and run it, type "Bob wins".

Refactor

We have some repetition in our respective applications where we are opening a file and creating a file_system_store from its contents. This feels like a slight weakness in our package's design so we should make a function in it to encapsulate opening a file from a path and returning you the PlayerStore.

```
//file_system_store.go
func FileSystemPlayerStoreFromFile(path string) (*FileSystemPlayerStore, func(), error) {
    db, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        return nil, nil, fmt.Errorf("problem opening %s %v", path, err)
    }

    closeFunc := func() {
        db.Close()
    }

    store, err := NewFileSystemPlayerStore(db)

    if err != nil {
        return nil, nil, fmt.Errorf("problem creating file system player store, %v ", err)
    }

    return store, closeFunc, nil
}
```

Now refactor both of our applications to use this function to create the store.

CLI application code

```
// cmd/cli/main.go
package main

import (
    "fmt"
    "github.com/quii/learn-go-with-tests/command-line/v3"
    "log"
    "os"
)

const dbFileName = "game.db.json"

func main() {
```

```
store, close, err := poker.FileSystemPlayerStoreFromFile(dbFileName)

if err != nil {
    log.Fatal(err)
}
defer close()

fmt.Println("Let's play poker")
fmt.Println("Type {Name} wins to record a win")
poker.NewCLI(store, os.Stdin).PlayPoker()
}
```

Web server application code

```
// cmd/webserver/main.go
package main

import (
    "github.com/quii/learn-go-with-tests/command-line/v3"
    "log"
    "net/http"
)

const dbFileName = "game.db.json"

func main() {
    store, close, err := poker.FileSystemPlayerStoreFromFile(dbFileName)

    if err != nil {
        log.Fatal(err)
    }
    defer close()

    server := poker.NewPlayerServer(store)

    if err := http.ListenAndServe(":5000", server); err != nil {
        log.Fatalf("could not listen on port 5000 %v", err)
    }
}
```

Notice the symmetry: despite being different user interfaces the setup is almost identical. This feels like good validation of our design so far. And notice also that `FileSystemPlayerStoreFromFile` returns a closing function, so we can close the underlying file once we are done using the Store.

Wrapping up

Package structure

This chapter meant we wanted to create two applications, re-using the domain code we've written so far. In order to do this, we needed to update our package structure so that we had separate folders for our respective mains.

By doing this we ran into integration problems due to unexported values so this further demonstrates the value of working in small "slices" and integrating often.

We learned how `mypackage_test` helps us create a testing environment which is the same experience for other packages integrating with your code, to help you catch integration problems and see how easy (or not!) your code is to work with.

Reading user input

We saw how reading from `os.Stdin` is very easy for us to work with as it implements `io.Reader`. We used `bufio.Scanner` to easily read line by line user input.

Simple abstractions leads to simpler code re-use

It was almost no effort to integrate `PlayerStore` into our new application (once we had made the package adjustments) and subsequently testing was very easy too because we decided to expose our stub version too.

Time

You can find all the code for this chapter here

The product owner wants us to expand the functionality of our command line application by helping a group of people play Texas-Holdem Poker.

Just enough information on poker

You won't need to know much about poker, only that at certain time intervals all the players need to be informed of a steadily increasing "blind" value.

Our application will help keep track of when the blind should go up, and how much it should be.

-
- When it starts it asks how many players are playing. This determines the amount of time there is before the "blind" bet goes up.
 - There is a base amount of time of 5 minutes.
 - For every player, 1 minute is added.
 - e.g 6 players equals 11 minutes for the blind.
 - After the blind time expires the game should alert the players the new amount the blind bet is.
 - The blind starts at 100 chips, then 200, 400, 600, 1000, 2000 and continue to double until the game ends (our previous functionality of "Ruth wins" should still finish the game)

Reminder of the code

In the previous chapter we made our start to the command line application which already accepts a command of {name} wins. Here is what the current CLI code looks like, but be sure to familiarise yourself with the other code too before starting.

```
type CLI struct {
    playerStore PlayerStore
    in         *bufio.Scanner
}

func NewCLI(store PlayerStore, in io.Reader) *CLI {
    return &CLI{
        playerStore: store,
        in:         bufio.NewScanner(in),
    }
}

func (cli *CLI) PlayPoker() {
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

time.AfterFunc

We want to be able to schedule our program to print the blind bet values at certain durations dependant on the number of players.

To limit the scope of what we need to do, we'll forget about the number of players part for now and just assume there are 5 players so we'll test that every 10 minutes the new value of the blind bet is printed.

As usual the standard library has us covered with `func AfterFunc(d Duration, f func()) *Timer`

AfterFunc waits for the duration to elapse and then calls f in its own goroutine. It returns a Timer that can be used to cancel the call using its Stop method.

time.Duration

A Duration represents the elapsed time between two instants as an int64 nanosecond count.

The time library has a number of constants to let you multiply those nanoseconds so they're a bit more readable for the kind of scenarios we'll be doing

`5 * time.Second`

When we call `PlayPoker` we'll schedule all of our blind alerts.

Testing this may be a little tricky though. We'll want to verify that each time period is scheduled with the correct blind amount but if you look at the signature of `time.AfterFunc` its second argument is the function it will run. You cannot compare functions in Go so we'd be unable to test what function has been sent in. So we'll need to write some kind of wrapper around `time.AfterFunc` which will take the time to run and the amount to print so we can spy on that.

Write the test first

Add a new test to our suite

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(playerStore, in, blindAlerter)
    cli.PlayPoker()
```

```
    if len(blindAlerter.alerts) != 1 {
        t.Fatal("expected a blind alert to be scheduled")
    }
})
```

You'll notice we've made a SpyBlindAlerter which we are trying to inject into our CLI and then checking that after we call PlayPoker that an alert is scheduled.

(Remember we are just going for the simplest scenario first and then we'll iterate.)

Here's the definition of SpyBlindAlerter

```
type SpyBlindAlerter struct {
    alerts []struct {
        scheduledAt time.Duration
        amount      int
    }
}

func (s *SpyBlindAlerter) ScheduleAlertAt(duration time.Duration, amount int) {
    s.alerts = append(s.alerts, struct {
        scheduledAt time.Duration
        amount      int
    }{duration, amount})
}
```

Try to run the test

```
./CLI_test.go:32:27: too many arguments in call to poker.NewCLI
    have (*poker.StubPlayerStore, *strings.Reader, *SpyBlindAlerter)
    want (poker.PlayerStore, io.Reader)
```

Write the minimal amount of code for the test to run and check the failing test output

We have added a new argument and the compiler is complaining. Strictly speaking the minimal amount of code is to make NewCLI accept a *SpyBlindAlerter but let's cheat a little and just define the dependency as an interface.

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}
```

And then add it to the constructor

```
func NewCLI(store PlayerStore, in io.Reader, alerter BlindAlerter) *CLI
```

Your other tests will now fail as they don't have a BlindAlerter passed in to NewCLI.

Spying on BlindAlerter is not relevant for the other tests so in the test file add

```
var dummySpyAlerter = &SpyBlindAlerter{}
```

Then use that in the other tests to fix the compilation problems. By labelling it as a "dummy" it is clear to the reader of the test that it is not important.

> Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

The tests should now compile and our new test fails.

```
==== RUN TestCLI
==== RUN TestCLI/it_schedules_printing_of_blind_values
--- FAIL: TestCLI (0.00s)
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values (0.00s)
        CLI_test.go:38: expected a blind alert to be scheduled
```

Write enough code to make it pass

We'll need to add the BlindAlerter as a field on our CLI so we can reference it in our PlayPoker method.

```
type CLI struct {
    playerStore PlayerStore
    in         *bufio.Scanner
    alerter    BlindAlerter
}

func NewCLI(store PlayerStore, in io.Reader, alerter BlindAlerter) *CLI {
    return &CLI{
        playerStore: store,
        in:         bufio.NewScanner(in),
        alerter:    alerter,
    }
}
```

To make the test pass, we can call our BlindAlerter with anything we like

```
func (cli *CLI) PlayPoker() {
    cli.alerter.ScheduleAlertAt(5*time.Second, 100)
    userInput := cli.readLine()
```

```
        cli.playerStore.RecordWin(extractWinner(userInput))
    }
```

Next we'll want to check it schedules all the alerts we'd hope for, for 5 players

Write the test first

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(playerStore, in, blindAlerter)
    cli.PlayPoker()

    cases := []struct {
        expectedScheduleTime time.Duration
        expectedAmount      int
    }{
        {0 * time.Second, 100},
        {10 * time.Minute, 200},
        {20 * time.Minute, 300},
        {30 * time.Minute, 400},
        {40 * time.Minute, 500},
        {50 * time.Minute, 600},
        {60 * time.Minute, 800},
        {70 * time.Minute, 1000},
        {80 * time.Minute, 2000},
        {90 * time.Minute, 4000},
        {100 * time.Minute, 8000},
    }

    for i, c := range cases {
        t.Run(fmt.Sprintf("%d scheduled for %v", c.expectedAmount, c.expectedScheduleTime), func(t *testing.T) {
            if len(blindAlerter.alerts) <= i {
                t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
            }

            alert := blindAlerter.alerts[i]

            amountGot := alert.amount
            if amountGot != c.expectedAmount {
                t.Errorf("got amount %d, want %d", amountGot, c.expectedAmount)
            }
        })
    }
})
```

```
        gotScheduledTime := alert.scheduledAt
        if gotScheduledTime != c.expectedScheduleTime {
            t.Errorf("got scheduled time of %v, want %v", gotScheduledTime, c.expectedScheduleTime)
        }
    })
}
})
```

Table-based test works nicely here and clearly illustrate what our requirements are. We run through the table and check the SpyBlindAlerter to see if the alert has been scheduled with the correct values.

Try to run the test

You should have a lot of failures looking like this

```
==== RUN TestCLI
--- FAIL: TestCLI (0.00s)
==== RUN TestCLI/it_schedules_printing_of_blind_values
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values (0.00s)
==== RUN TestCLI/it_schedules_printing_of_blind_values/100_scheduled_for_0s
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values/100_scheduled_for_0s (0.00s)
        CLI_test.go:71: got scheduled time of 5s, want 0s
==== RUN TestCLI/it_schedules_printing_of_blind_values/200_scheduled_for_10m0s
    --- FAIL: TestCLI/it_schedules_printing_of_blind_values/200_scheduled_for_10m0s (0.00s)
        CLI_test.go:59: alert 1 was not scheduled [{5000000000 100}]
```

Write enough code to make it pass

```
func (cli *CLI) PlayPoker() {
    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        cli.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + 10*time.Minute
    }

    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}
```

It's not a lot more complicated than what we already had. We're just now iterating over an array of blinds and calling the scheduler on an increasing blindTime

Refactor

We can encapsulate our scheduled alerts into a method just to make PlayPoker read a little clearer.

```
func (cli *CLI) PlayPoker() {
    cli.scheduleBlindAlerts()
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func (cli *CLI) scheduleBlindAlerts() {
    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        cli.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + 10*time.Minute
    }
}
```

Finally our tests are looking a little clunky. We have two anonymous structs representing the same thing, a ScheduledAlert. Let's refactor that into a new type and then make some helpers to compare them.

```
type scheduledAlert struct {
    at    time.Duration
    amount int
}

func (s scheduledAlert) String() string {
    return fmt.Sprintf("%d chips at %v", s.amount, s.at)
}

type SpyBlindAlerter struct {
    alerts []scheduledAlert
}

func (s *SpyBlindAlerter) ScheduleAlertAt(at time.Duration, amount int) {
    s.alerts = append(s.alerts, scheduledAlert{at, amount})
}
```

We've added a String() method to our type so it prints nicely if the test fails

Update our test to use our new type

```
t.Run("it schedules printing of blind values", func(t *testing.T) {
    in := strings.NewReader("Chris wins\n")
    playerStore := &poker.StubPlayerStore{}
```

```
blindAlerter := &SpyBlindAlerter{}

cli := poker.NewCLI(playerStore, in, blindAlerter)
cli.PlayPoker()

cases := []scheduledAlert{
    {0 * time.Second, 100},
    {10 * time.Minute, 200},
    {20 * time.Minute, 300},
    {30 * time.Minute, 400},
    {40 * time.Minute, 500},
    {50 * time.Minute, 600},
    {60 * time.Minute, 800},
    {70 * time.Minute, 1000},
    {80 * time.Minute, 2000},
    {90 * time.Minute, 4000},
    {100 * time.Minute, 8000},
}

for i, want := range cases {
    t.Run(fmt.Sprintf("want %d", i), func(t *testing.T) {
        if len(blindAlerter.alerts) <= i {
            t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
        }

        got := blindAlerter.alerts[i]
        assertScheduledAlert(t, got, want)
    })
}
}
```

Implement assertScheduledAlert yourself.

We've spent a fair amount of time here writing tests and have been somewhat naughty not integrating with our application. Let's address that before we pile on any more requirements.

Try running the app and it won't compile, complaining about not enough args to NewCLI.

Let's create an implementation of BlindAlerter that we can use in our application.

Create blind_alerter.go and move our BlindAlerter interface and add the new things below

```
package poker
```

```
import (
    "fmt"
    "os"
    "time"
)

type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}

type BlindAlerterFunc func(duration time.Duration, amount int)

func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int) {
    a(duration, amount)
}

func StdOutAlerter(duration time.Duration, amount int) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(os.Stdout, "Blind is now %d\n", amount)
    })
}
```

Remember that any type can implement an interface, not just structs. If you are making a library that exposes an interface with one function defined it is a common idiom to also expose a MyInterfaceFunc type.

This type will be a func which will also implement your interface. That way users of your interface have the option to implement your interface with just a function; rather than having to create an empty struct type.

We then create the function StdOutAlerter which has the same signature as the function and just use time.AfterFunc to schedule it to print to os.Stdout.

Update main where we create NewCLI to see this in action

```
poker.NewCLI(store, os.Stdin, poker.BlindAlerterFunc(poker.StdOutAlerter)).PlayPoker()
```

Before running you might want to change the blindTime increment in CLI to be 10 seconds rather than 10 minutes just so you can see it in action.

You should see it print the blind values as we'd expect every 10 seconds. Notice how you can still type Shaun wins into the CLI and it will stop the program how we'd expect.

The game won't always be played with 5 people so we need to prompt the user to enter a number of players before the game starts.

Write the test first

To check we are prompting for the number of players we'll want to record what is written to StdOut. We've done this a few times now, we know that os.Stdout is an io.Writer so we can check what is written if we use dependency injection to pass in a bytes.Buffer in our test and see what our code will write.

We don't care about our other collaborators in this test just yet so we've made some dummies in our test file.

We should be a little wary that we now have 4 dependencies for CLI, that feels like maybe it is starting to have too many responsibilities. Let's live with it for now and see if a refactoring emerges as we add this new functionality.

```
var dummyBlindAlerter = &SpyBlindAlerter{}
var dummyPlayerStore = &poker.StubPlayerStore{}
var dummyStdIn = &bytes.Buffer{}
var dummyStdOut = &bytes.Buffer{}
```

Here is our new test

```
t.Run("it prompts the user to enter the number of players", func(t *testing.T) {
    stdout := &bytes.Buffer{}
    cli := poker.NewCLI(dummyPlayerStore, dummyStdIn, stdout, dummyBlindAlerter)
    cli.PlayPoker()

    got := stdout.String()
    want := "Please enter the number of players: "

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

We pass in what will be os.Stdout in main and see what is written.

Try to run the test

```
./CLI_test.go:38:27: too many arguments in call to poker.NewCLI
    have (*poker.StubPlayerStore, *bytes.Buffer, *bytes.Buffer, *SpyBlindAlerter)
    want (poker.PlayerStore, io.Reader, poker.BlindAlerter)
```

Write the minimal amount of code for the test to run and check the failing test output

We have a new dependency so we'll have to update NewCLI

```
func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI
```

Now the other tests will fail to compile because they don't have an io.Writer being passed into NewCLI.

Add dummyStdout for the other tests.

The new test should fail like so

```
==== RUN TestCLI
--- FAIL: TestCLI (0.00s)
==== RUN TestCLI/it_prompts_the_user_to_enter_the_number_of_players
    --- FAIL: TestCLI/it_prompts_the_user_to_enter_the_number_of_players (0.00s)
        CLI_test.go:46: got "", want 'Please enter the number of players: '
FAIL
```

Write enough code to make it pass

We need to add our new dependency to our CLI so we can reference it in PlayPoker

```
type CLI struct {
    playerStore PlayerStore
    in         *bufio.Scanner
    out        io.Writer
    alerter    BlindAlerter
}
```

```
func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI {
    return &CLI{
        playerStore: store,
        in:         bufio.NewScanner(in),
        out:        out,
        alerter:    alerter,
    }
}
```

Then finally we can write our prompt at the start of the game

```
func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, "Please enter the number of players: ")
    cli.scheduleBlindAlerts()
    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}
```

Refactor

We have a duplicate string for the prompt which we should extract into a constant

```
const PlayerPrompt = "Please enter the number of players: "
```

Use this in both the test code and CLI.

Now we need to send in a number and extract it out. The only way we'll know if it has had the desired effect is by seeing what blind alerts were scheduled.

Write the test first

```
t.Run("it prompts the user to enter the number of players", func(t *testing.T) {
    stdout := &bytes.Buffer{}
    in := strings.NewReader("7\n")
    blindAlerter := &SpyBlindAlerter{}

    cli := poker.NewCLI(dummyPlayerStore, in, stdout, blindAlerter)
    cli.PlayPoker()

    got := stdout.String()
    want := poker.PlayerPrompt

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

    cases := []scheduledAlert{
        {0 * time.Second, 100},
        {12 * time.Minute, 200},
        {24 * time.Minute, 300},
        {36 * time.Minute, 400},
    }

    for i, want := range cases {
        t.Run(fmt.Sprintf(want), func(t *testing.T) {

            if len(blindAlerter.alerts) <= i {
                t.Fatalf("alert %d was not scheduled %v", i, blindAlerter.alerts)
            }

            got := blindAlerter.alerts[i]
            assertScheduledAlert(t, got, want)
        })
    }
})
```

```
    }
})
```

Ouch! A lot of changes.

- We remove our dummy for StdIn and instead send in a mocked version representing our user entering 7
- We also remove our dummy on the blind alerter so we can see that the number of players has had an effect on the scheduling
- We test what alerts are scheduled

Try to run the test

The test should still compile and fail reporting that the scheduled times are wrong because we've hard-coded for the game to be based on having 5 players

```
==== RUN TestCLI
--- FAIL: TestCLI (0.00s)
==== RUN TestCLI/it_prompts_the_user_to_enter_the_number_of_players
    --- FAIL: TestCLI/it_prompts_the_user_to_enter_the_number_of_players (0.00s)
==== RUN TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chips_at_0s
    --- PASS: TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chips_at_0s (0.00s)
==== RUN TestCLI/it_prompts_the_user_to_enter_the_number_of_players/200_chips_at_12m0s
```

Write enough code to make it pass

Remember, we are free to commit whatever sins we need to make this work. Once we have working software we can then work on refactoring the mess we're about to make!

```
func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, PlayerPrompt)

    numberOfPlayers, _ := strconv.Atoi(cli.readLine())

    cli.scheduleBlindAlerts(numberOfPlayers)

    userInput := cli.readLine()
    cli.playerStore.RecordWin(extractWinner(userInput))
}

func (cli *CLI) scheduleBlindAlerts(numberOfPlayers int) {
    blindIncrement := time.Duration(5+numberOfPlayers) * time.Minute

    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
```

```
for _, blind := range blinds {
    cli.alerter.ScheduleAlertAt(blindTime, blind)
    blindTime = blindTime + blindIncrement
}
}
```

- We read in the `numberOfPlayersInput` into a string
- We use `cli.readLine()` to get the input from the user and then call `Atoi` to convert it into an integer - ignoring any error scenarios. We'll need to write a test for that scenario later.
- From here we change `scheduleBlindAlerts` to accept a number of players. We then calculate a `blindIncrement` time to use to add to `blindTime` as we iterate over the blind amounts

While our new test has been fixed, a lot of others have failed because now our system only works if the game starts with a user entering a number. You'll need to fix the tests by changing the user inputs so that a number followed by a newline is added (this is highlighting yet more flaws in our approach right now).

Refactor

This all feels a bit horrible right? Let's **listen to our tests**.

- In order to test that we are scheduling some alerts we set up 4 different dependencies. Whenever you have a lot of dependencies for a thing in your system, it implies it's doing too much. Visually we can see it in how cluttered our test is.
- To me it feels like **we need to make a cleaner abstraction between reading user input and the business logic we want to do**
- A better test would be given this user input, do we call a new type `Game` with the correct number of players.
- We would then extract the testing of the scheduling into the tests for our new `Game`.

We can refactor toward our `Game` first and our test should continue to pass. Once we've made the structural changes we want we can think about how we can refactor the tests to reflect our new separation of concerns

Remember when making changes in refactoring try to keep them as small as possible and keep re-running the tests.

Try it yourself first. Think about the boundaries of what a `Game` would offer and what our CLI should be doing.

For now **don't** change the external interface of `NewCLI` as we don't want to change the test code and the client code at the same time as

that is too much to juggle and we could end up breaking things.

This is what I came up with:

```
// game.go
type Game struct {
    alerter BlindAlerter
    store PlayerStore
}

func (p *Game) Start(numberOfPlayers int) {
    blindIncrement := time.Duration(5+numberOfPlayers) * time.Minute

    blinds := []int{100, 200, 300, 400, 500, 600, 800, 1000, 2000, 4000, 8000}
    blindTime := 0 * time.Second
    for _, blind := range blinds {
        p.alerter.ScheduleAlertAt(blindTime, blind)
        blindTime = blindTime + blindIncrement
    }
}

func (p *Game) Finish(winner string) {
    p.store.RecordWin(winner)
}

// cli.go
type CLI struct {
    in *bufio.Scanner
    out io.Writer
    game *Game
}

func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CLI {
    return &CLI{
        in: bufio.NewScanner(in),
        out: out,
        game: &Game{
            alerter: alerter,
            store: store,
        },
    }
}

const PlayerPrompt = "Please enter the number of players: "

func (cli *CLI) PlayPoker() {
    fmt.Fprint(cli.out, PlayerPrompt)
```

```
    numberOfPlayersInput := cli.readLine()
    numberOfPlayers, _ := strconv.Atoi(strings.Trim(numberOfPlayersInput, "\n"))

    cli.game.Start(numberOfPlayers)

    winnerInput := cli.readLine()
    winner := extractWinner(winnerInput)

    cli.game.Finish(winner)
}

func extractWinner(userInput string) string {
    return strings.Replace(userInput, " wins\n", "", 1)
}

func (cli *CLI) readLine() string {
    cli.in.Scan()
    return cli.in.Text()
}
```

From a "domain" perspective:

- We want to Start a Game, indicating how many people are playing
- We want to Finish a Game, declaring the winner

The new Game type encapsulates this for us.

With this change we've passed BlindAlerter and PlayerStore to Game as it is now responsible for alerting and storing results.

Our CLI is now just concerned with:

- Constructing Game with its existing dependencies (which we'll refactor next)
- Interpreting user input as method invocations for Game

We want to try to avoid doing "big" refactors which leave us in a state of failing tests for extended periods as that increases the chances of mistakes. (If you are working in a large/distributed team this is extra important)

The first thing we'll do is refactor Game so that we inject it into CLI. We'll do the smallest changes in our tests to facilitate that and then we'll see how we can break up the tests into the themes of parsing user input and game management.

All we need to do right now is change NewCLI

```
func NewCLI(in io.Reader, out io.Writer, game *Game) *CLI {
    return &CLI{
        in:  bufio.NewScanner(in),
        out: out,
        game: game,
    }
}
```

This feels like an improvement already. We have less dependencies and our dependency list is reflecting our overall design goal of CLI being concerned with input/output and delegating game specific actions to a Game.

If you try and compile there are problems. You should be able to fix these problems yourself. Don't worry about making any mocks for Game right now, just initialise real Games just to get everything compiling and tests green.

To do this you'll need to make a constructor

```
func NewGame(alerter BlindAlerter, store PlayerStore) *Game {
    return &Game{
        alerter: alerter,
        store: store,
    }
}
```

Here's an example of one of the setups for the tests being fixed

```
stdout := &bytes.Buffer{}
in := strings.NewReader("7\n")
blindAlerter := &SpyBlindAlerter{}
game := poker.NewGame(blindAlerter, dummyPlayerStore)

cli := poker.NewCLI(in, stdout, game)
cli.PlayPoker()
```

It shouldn't take much effort to fix the tests and be back to green again (that's the point!) but make sure you fix main.go too before the next stage.

```
// main.go
game := poker.NewGame(poker.BlindAlerterFunc(poker.StdOutAlerter), store)
cli := poker.NewCLI(os.Stdin, os.Stdout, game)
cli.PlayPoker()
```

Now that we have extracted out Game we should move our game specific assertions into tests separate from CLI.

This is just an exercise in copying our CLI tests but with less dependencies

```
func TestGame_Start(t *testing.T) {
    t.Run("schedules alerts on game start for 5 players", func(t *testing.T) {
        blindAlerter := &poker.SpyBlindAlerter{}
        game := poker.NewGame(blindAlerter, dummyPlayerStore)

        game.Start(5)

        cases := []poker.ScheduledAlert{
            {At: 0 * time.Second, Amount: 100},
            {At: 10 * time.Minute, Amount: 200},
            {At: 20 * time.Minute, Amount: 300},
            {At: 30 * time.Minute, Amount: 400},
            {At: 40 * time.Minute, Amount: 500},
            {At: 50 * time.Minute, Amount: 600},
            {At: 60 * time.Minute, Amount: 800},
            {At: 70 * time.Minute, Amount: 1000},
            {At: 80 * time.Minute, Amount: 2000},
            {At: 90 * time.Minute, Amount: 4000},
            {At: 100 * time.Minute, Amount: 8000},
        }

        checkSchedulingCases(cases, t, blindAlerter)
    })

    t.Run("schedules alerts on game start for 7 players", func(t *testing.T) {
        blindAlerter := &poker.SpyBlindAlerter{}
        game := poker.NewGame(blindAlerter, dummyPlayerStore)

        game.Start(7)

        cases := []poker.ScheduledAlert{
            {At: 0 * time.Second, Amount: 100},
            {At: 12 * time.Minute, Amount: 200},
            {At: 24 * time.Minute, Amount: 300},
            {At: 36 * time.Minute, Amount: 400},
        }

        checkSchedulingCases(cases, t, blindAlerter)
    })

}

func TestGame_Finish(t *testing.T) {
    store := &poker.StubPlayerStore{}
    game := poker.NewGame(dummyBlindAlerter, store)
    winner := "Ruth"
```

```
    game.Finish(winner)
    poker.AssertPlayerWin(t, store, winner)
}
```

The intent behind what happens when a game of poker starts is now much clearer.

Make sure to also move over the test for when the game ends.

Once we are happy we have moved the tests over for game logic we can simplify our CLI tests so they reflect our intended responsibilities clearer

- Process user input and call Game's methods when appropriate
- Send output
- Crucially it doesn't know about the actual workings of how games work

To do this we'll have to make it so CLI no longer relies on a concrete Game type but instead accepts an interface with Start(numberOfPlayers) and Finish(winner). We can then create a spy of that type and verify the correct calls are made.

It's here we realise that naming is awkward sometimes. Rename Game to TexasHoldem (as that's the kind of game we're playing) and the new interface will be called Game. This keeps faithful to the notion that our CLI is oblivious to the actual game we're playing and what happens when you Start and Finish.

```
type Game interface {
    Start(numberOfPlayers int)
    Finish(winner string)
}
```

Replace all references to *Game inside CLI and replace them with Game (our new interface). As always keep re-running tests to check everything is green while we are refactoring.

Now that we have decoupled CLI from TexasHoldem we can use spies to check that Start and Finish are called when we expect them to, with the correct arguments.

Create a spy that implements Game

```
type GameSpy struct {
    StartedWith int
    FinishedWith string
}
```

```
func (g *GameSpy) Start(numberOfPlayers int) {
```

```
        g.StartedWith = numberOfPlayers
    }

func (g *GameSpy) Finish(winner string) {
    g.FinishedWith = winner
}
```

Replace any CLI test which is testing any game specific logic with checks on how our GameSpy is called. This will then reflect the responsibilities of CLI in our tests clearly.

Here is an example of one of the tests being fixed; try and do the rest yourself and check the source code if you get stuck.

```
t.Run("it prompts the user to enter the number of players and starts the game", func(t *testing.T) {
    stdout := &bytes.Buffer{}
    in := strings.NewReader("7\n")
    game := &GameSpy{}

    cli := poker.NewCLI(in, stdout, game)
    cli.PlayPoker()

    gotPrompt := stdout.String()
    wantPrompt := poker.PlayerPrompt

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

    if game.StartedWith != 7 {
        t.Errorf("wanted Start called with 7 but got %d", game.StartedWith)
    }
})
```

Now that we have a clean separation of concerns, checking edge cases around IO in our CLI should be easier.

We need to address the scenario where a user puts a non numeric value when prompted for the number of players:

Our code should not start the game and it should print a handy error to the user and then exit.

Write the test first

We'll start by making sure the game doesn't start

```
t.Run("it prints an error when a non numeric value is entered and does not start the game", func(t *testing.T) {
    stdout := &bytes.Buffer{}}
```

```
in := strings.NewReader("Pies\n")
game := &GameSpy{}

cli := poker.NewCLI(in, stdout, game)
cli.PlayPoker()

if game.StartCalled {
    t.Errorf("game should not have started")
}
})
```

You'll need to add to our GameSpy a field StartCalled which only gets set if Start is called

Try to run the test

```
==== RUN TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start
--- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start
CLI_test.go:62: game should not have started
```

Write enough code to make it pass

Around where we call Atoi we just need to check for the error

```
numberOfPlayers, err := strconv.Atoi(cli.readLine())

if err != nil {
    return
}
```

Next we need to inform the user of what they did wrong so we'll assert on what is printed to stdout.

Write the test first

We've asserted on what was printed to stdout before so we can copy that code for now

```
gotPrompt := stdout.String()

wantPrompt := poker.PlayerPrompt + "you're so silly"

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

We are storing everything that gets written to stdout so we still expect the poker.PlayerPrompt. We then just check an additional thing gets printed. We're not too bothered about the exact wording for now, we'll address it when we refactor.

Try to run the test

```
==== RUN TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start_with_a_number
--- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does_not_start_with_a_number (0.00s)
    CLI_test.go:70: got 'Please enter the number of players: ', want 'Please enter the number of players: 1'
```

Write enough code to make it pass

Change the error handling code

```
if err != nil {
    fmt.Fprint(cli.out, "you're so silly")
    return
}
```

Refactor

Now refactor the message into a constant like PlayerPrompt

```
wantPrompt := poker.PlayerPrompt + poker.BadPlayerInputErrMsg
```

and put in a more appropriate message

```
const BadPlayerInputErrMsg = "Bad value received for number of players, please try again with a number"
```

Finally our testing around what has been sent to stdout is quite verbose, let's write an assert function to clean it up.

```
func assertMessagesSentToUser(t testing.TB, stdout *bytes.Buffer, messages ...string) {
    t.Helper()
    want := strings.Join(messages, "\n")
    got := stdout.String()
    if got != want {
        t.Errorf("got %q sent to stdout but expected %v", got, messages)
    }
}
```

Using the vararg syntax (...string) is handy here because we need to assert on varying amounts of messages.

Use this helper in both of the tests where we assert on messages sent to the user.

There are a number of tests that could be helped with some assertX functions so practice your refactoring by cleaning up our tests so they read nicely.

Take some time and think about the value of some of the tests we've driven out. Remember we don't want more tests than necessary, can you refactor/remove some of them and still be confident it all works ?

Here is what I came up with

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

    t.Run("start game with 3 players and finish game with 'Chris' as winner", func(t *testing.T) {
        game := &GameSpy{}
        stdout := &bytes.Buffer{}

        in := userSends("3", "Chris wins")
        cli := poker.NewCLI(in, stdout, game)

        cli.PlayPoker()

        assertMessagesSentToUser(t, stdout, poker.PlayerPrompt)
        assertGameStartedWith(t, game, 3)
        assertFinishCalledWith(t, game, "Chris")
    })

    t.Run("start game with 8 players and record 'Cleo' as winner", func(t *testing.T) {
        game := &GameSpy{}

        in := userSends("8", "Cleo wins")
        cli := poker.NewCLI(in, dummyStdOut, game)

        cli.PlayPoker()

        assertGameStartedWith(t, game, 8)
        assertFinishCalledWith(t, game, "Cleo")
    })

    t.Run("it prints an error when a non numeric value is entered and does not start the game", func(t *testing.T) {
        game := &GameSpy{}

        stdout := &bytes.Buffer{}
        in := userSends("pies")

        cli := poker.NewCLI(in, stdout, game)
        cli.PlayPoker()
    })
}
```

```
        assertGameNotStarted(t, game)
        assertMessagesSentToUser(t, stdout, poker.PlayerPrompt, poker.BadPlayerInputErrMsg)
    })
}
```

The tests now reflect the main capabilities of CLI, it is able to read user input in terms of how many people are playing and who won and handles when a bad value is entered for number of players. By doing this it is clear to the reader what CLI does, but also what it doesn't do.

What happens if instead of putting Ruth wins the user puts in Lloyd is a killer ?

Finish this chapter by writing a test for this scenario and making it pass.

Wrapping up

A quick project recap

For the past 5 chapters we have slowly TDD'd a fair amount of code

- We have two applications, a command line application and a web server.
- Both these applications rely on a PlayerStore to record winners
- The web server can also display a league table of who is winning the most games
- The command line app helps players play a game of poker by tracking what the current blind value is.

time.Afterfunc

A very handy way of scheduling a function call after a specific duration. It is well worth investing time [looking at the documentation for time](#) as it has a lot of time saving functions and methods for you to work with.

Some of my favourites are

- `time.After(duration)` returns a chan Time when the duration has expired. So if you wish to do something after a specific time, this can help.
- `time.NewTicker(duration)` returns a Ticker which is similar to the above in that it returns a channel but this one "ticks" every duration, rather than just once. This is very handy if you want to execute some code every N duration.

More examples of good separation of concerns

Generally it is good practice to separate the responsibilities of dealing with user input and responses away from domain code. You see that here in our command line application and also our web server.

Our tests got messy. We had too many assertions (check this input, schedules these alerts, etc) and too many dependencies. We could visually see it was cluttered; it is **so important to listen to your tests.**

- If your tests look messy try and refactor them.
- If you've done this and they're still a mess it is very likely pointing to a flaw in your design
- This is one of the real strengths of tests.

Even though the tests and the production code was a bit cluttered we could freely refactor backed by our tests.

Remember when you get into these situations to always take small steps and re-run the tests after every change.

It would've been dangerous to refactor both the test code and the production code at the same time, so we first refactored the production code (in the current state we couldn't improve the tests much) without changing its interface so we could rely on our tests as much as we could while changing things. Then we refactored the tests after the design improved.

After refactoring the dependency list reflected our design goal. This is another benefit of DI in that it often documents intent. When you rely on global variables responsibilities become very unclear.

An example of a function implementing an interface

When you define an interface with one method in it you might want to consider defining a `MyInterfaceFunc` type to complement it so users can implement your interface with just a function.

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}

// BlindAlerterFunc allows you to implement BlindAlerter with a function
type BlindAlerterFunc func(duration time.Duration, amount int)

// ScheduleAlertAt is BlindAlerterFunc implementation of BlindAlerter
func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int) {
```

```
    a(duration, amount)
}
```

By doing this, people using your library can implement your interface with just a function. They can use [Type Conversion](#) to convert their function into a `BlindAlerterFunc` and then use it as a `BlindAlerter` (as `BlindAlerterFunc` implements `BlindAlerter`).

```
game := poker.NewTexasHoldem(poker.BlindAlerterFunc(poker.StdOutAlerter), store)
```

The broader point here is, in Go you can add methods to types, not just structs. This is a very powerful feature, and you can use it to implement interfaces in more convenient ways.

Consider that you can not only define types of functions, but also define types around other types, so that you can add methods to them.

```
type Blog map[string]string
```

```
func (b Blog) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, b[r.URL.Path])
}
```

Here we've created an HTTP handler that implements a very simple "blog" where it will use URL paths as keys to posts stored in a map.

WebSockets

[You can find all the code for this chapter here](#)

In this chapter we'll learn how to use WebSockets to improve our application.

Project recap

We have two applications in our poker codebase

- Command line app. Prompts the user to enter the number of players in a game. From then on informs the players of what the "blind bet" value is, which increases over time. At any point a user can enter "{Playername} wins" to finish the game and record the victor in a store.
- Web app. Allows users to record winners of games and displays a league table. Shares the same store as the command line app.

Next steps

The product owner is thrilled with the command line application but would prefer it if we could bring that functionality to the browser. She imagines a web page with a text box that allows the user to enter the number of players and when they submit the form the page displays the blind value and automatically updates it when appropriate. Like the command line application the user can declare the winner and it'll get saved in the database.

On the face of it, it sounds quite simple but as always we must emphasise taking an iterative approach to writing software.

First we will need to serve HTML. So far all of our HTTP endpoints have returned either plaintext or JSON. We could use the same techniques we know (as they're all ultimately strings) but we can also use the [html/template](#) package for a cleaner solution.

We also need to be able to asynchronously send messages to the user saying The blind is now *y* without having to refresh the browser. We can use [WebSockets](#) to facilitate this.

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection

Given we are taking on a number of techniques it's even more important we do the smallest amount of useful work possible first and then iterate.

For that reason the first thing we'll do is create a web page with a form for the user to record a winner. Rather than using a plain form, we will use WebSockets to send that data to our server for it to record.

After that we'll work on the blind alerts by which point we will have a bit of infrastructure code set up.

What about tests for the JavaScript ?

There will be some JavaScript written to do this but I won't go in to writing tests.

It is of course possible but for the sake of brevity I won't be including any explanations for it.

Sorry folks. Lobby O'Reilly to pay me to make a "Learn JavaScript with tests".

Write the test first

First thing we need to do is serve up some HTML to users when they hit /game.

Here's a reminder of the pertinent code in our web server

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
}

const jsonContentType = "application/json"

func NewPlayerServer(store PlayerStore) *PlayerServer {
    p := new(PlayerServer)

    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))

    p.Handler = router

    return p
}
```

The easiest thing we can do for now is check when we GET /game that we get a 200.

```
func TestGame(t *testing.T) {
    t.Run("GET /game returns 200", func(t *testing.T) {
        server := NewPlayerServer(&StubPlayerStore{})

        request, _ := http.NewRequest(http.MethodGet, "/game", nil)
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response.Code, http.StatusOK)
    })
}
```

Try to run the test

```
--- FAIL: TestGame (0.00s)
==== RUN TestGame/GET_/game_returns_200
```

```
--- FAIL: TestGame/GET_/game_returns_200 (0.00s)
server_test.go:109: did not get correct status, got 404, want 200
```

Write enough code to make it pass

Our server has a router setup so it's relatively easy to fix.

To our router add

```
router.Handle("/game", http.HandlerFunc(p.game))
```

And then write the game method

```
func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
}
```

Refactor

The server code is already fine due to us slotting in more code into the existing well-factored code very easily.

We can tidy up the test a little by adding a test helper function newGameRequest to make the request to /game. Try writing this yourself.

```
func TestGame(t *testing.T) {
    t.Run("GET /game returns 200", func(t *testing.T) {
        server := NewPlayerServer(&StubPlayerStore{})

        request := newGameRequest()
        response := httptest.NewRecorder()

        server.ServeHTTP(response, request)

        assertStatus(t, response, http.StatusOK)
    })
}
```

You'll also notice I changed assertStatus to accept response rather than response.Code as I feel it reads better.

Now we need to make the endpoint return some HTML, here it is

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
<title>Let's play poker</title>
</head>
<body>
<section id="game">
  <div id="declare-winner">
    <label for="winner">Winner</label>
    <input type="text" id="winner"/>
    <button id="winner-button">Declare winner</button>
  </div>
</section>
</body>
<script type="application/javascript">

  const submitWinnerButton = document.getElementById('winner-button')
  const winnerInput = document.getElementById('winner')

  if (window['WebSocket']) {
    const conn = new WebSocket('ws://' + document.location.host + '/ws')

    submitWinnerButton.onclick = event => {
      conn.send(winnerInput.value)
    }
  }
</script>
</html>
```

We have a very simple web page

- A text input for the user to enter the winner into
- A button they can click to declare the winner.
- Some JavaScript to open a WebSocket connection to our server and handle the submit button being pressed

WebSocket is built into most modern browsers so we don't need to worry about bringing in any libraries. The web page won't work for older browsers, but we're ok with that for this scenario.

How do we test we return the correct markup?

There are a few ways. As has been emphasised throughout the book, it is important that the tests you write have sufficient value to justify the cost.

1. Write a browser based test, using something like Selenium. These tests are the most "realistic" of all approaches because they start an actual web browser of some kind and simulates a user interacting with it. These tests can give you a lot of

confidence your system works but are more difficult to write than unit tests and much slower to run. For the purposes of our product this is overkill.

2. Do an exact string match. This can be ok but these kind of tests end up being very brittle. The moment someone changes the markup you will have a test failing when in practice nothing has actually broken.
3. Check we call the correct template. We will be using a templating library from the standard lib to serve the HTML (discussed shortly) and we could inject in the thing to generate the HTML and spy on its call to check we're doing it right. This would have an impact on our code's design but doesn't actually test a great deal; other than we're calling it with the correct template file. Given we will only have the one template in our project the chance of failure here seems low.

So in the book "Learn Go with Tests" for the first time, we're not going to write a test.

Put the markup in a file called game.html

Next change the endpoint we just wrote to the following

```
func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {  
    tmpl, err := template.ParseFiles("game.html")  
  
    if err != nil {  
        http.Error(w, fmt.Sprintf("problem loading template %s", err.Error()), http.StatusInternalServerError)  
        return  
    }  
  
    tmpl.Execute(w, nil)  
}
```

[html/template](#) is a Go package for creating HTML. In our case we call `template.ParseFiles`, giving the path of our html file. Assuming there is no error you can then `Execute` the template, which writes it to an `io.Writer`. In our case we want it to Write to the internet, so we give it our `http.ResponseWriter`.

As we have not written a test, it would be prudent to manually test our web server just to make sure things are working as we'd hope. Go to cmd/webserver and run the `main.go` file. Visit `http://localhost:5000/game`.

You should have got an error about not being able to find the template. You can either change the path to be relative to your folder, or you can have a copy of the `game.html` in the `cmd/webserver` directory. I chose to create a symlink (`ln -s ../../game.html game.html`) to the file

inside the root of the project so if I make changes they are reflected when running the server.

If you make this change and run again you should see our UI.

Now we need to test that when we get a string over a WebSocket connection to our server that we declare it as a winner of a game.

Write the test first

For the first time we are going to use an external library so that we can work with WebSockets.

Run go get github.com/gorilla/websocket

This will fetch the code for the excellent [Gorilla WebSocket](#) library. Now we can update our tests for our new requirement.

```
t.Run("when we get a message over a websocket it is a winner of a game", func(t *testing.T) {
    store := &StubPlayerStore{}
    winner := "Ruth"
    server := httptest.NewServer(NewPlayerServer(store))
    defer server.Close()

    wsURL := "ws" + strings.TrimPrefix(server.URL, "http") + "/ws"

    ws, _, err := websocket.DefaultDialer.Dial(wsURL, nil)
    if err != nil {
        t.Fatalf("could not open a ws connection on %s %v", wsURL, err)
    }
    defer ws.Close()

    if err := ws.WriteMessage(websocket.TextMessage, []byte(winner)); err != nil {
        t.Fatalf("could not send message over ws connection %v", err)
    }

    AssertPlayerWin(t, store, winner)
})
```

Make sure that you have an import for the `websocket` library. My IDE automatically did it for me, so should yours.

To test what happens from the browser we have to open up our own WebSocket connection and write to it.

Our previous tests around our server just called methods on our server but now we need to have a persistent connection to our server. To do that we use `httptest.NewServer` which takes a `http.Handler` and will spin it up and listen for connections.

Using `websocket.DefaultDialer.Dial` we try to dial in to our server and then we'll try and send a message with our winner.

Finally, we assert on the player store to check the winner was recorded.

Try to run the test

```
==== RUN TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
--- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game (0
server_test.go:124: could not open a ws connection on ws://127.0.0.1:55838/ws websocket
```

We have not changed our server to accept WebSocket connections on `/ws` so we're not shaking hands yet.

Write enough code to make it pass

Add another listing to our router

```
router.Handle("/ws", http.HandlerFunc(p.webSocket))
```

Then add our new webSocket handler

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    upgrader := websocket.Upgrader{
        ReadBufferSize: 1024,
        WriteBufferSize: 1024,
    }
    upgrader.Upgrade(w, r, nil)
}
```

To accept a WebSocket connection we Upgrade the request. If you now re-run the test you should move on to the next error.

```
==== RUN TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game
--- FAIL: TestGame/when_we_get_a_message_over_a_websocket_it_is_a_winner_of_a_game (0
server_test.go:132: got 0 calls to RecordWin want 1
```

Now that we have a connection opened, we'll want to listen for a message and then record it as the winner.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    upgrader := websocket.Upgrader{
        ReadBufferSize: 1024,
        WriteBufferSize: 1024,
    }
    conn, _ := upgrader.Upgrade(w, r, nil)
    _, winnerMsg, _ := conn.ReadMessage()
    p.store.RecordWin(string(winnerMsg))
}
```

(Yes, we're ignoring a lot of errors right now!)

conn.ReadMessage() blocks on waiting for a message on the connection. Once we get one we use it to RecordWin. This would finally close the WebSocket connection.

If you try and run the test, it's still failing.

The issue is timing. There is a delay between our WebSocket connection reading the message and recording the win and our test finishes before it happens. You can test this by putting a short time.Sleep before the final assertion.

Let's go with that for now but acknowledge that putting in arbitrary sleeps into tests **is very bad practice**.

```
time.Sleep(10 * time.Millisecond)  
AssertPlayerWin(t, store, winner)
```

Refactor

We committed many sins to make this test work both in the server code and the test code but remember this is the easiest way for us to work.

We have nasty, horrible, working software backed by a test, so now we are free to make it nice and know we won't break anything accidentally.

Let's start with the server code.

We can move the upgrader to a private value inside our package because we don't need to redeclare it on every WebSocket connection request

```
var wsUpgrader = websocket.Upgrader{  
    ReadBufferSize: 1024,  
    WriteBufferSize: 1024,  
}  
  
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {  
    conn, _ := wsUpgrader.Upgrade(w, r, nil)  
    _, winnerMsg, _ := conn.ReadMessage()  
    p.store.RecordWin(string(winnerMsg))  
}
```

Our call to template.ParseFiles("game.html") will run on every GET /game which means we'll go to the file system on every request even though we have no need to re-parse the template. Let's refactor our code so that we parse the template once in NewPlayerServer instead.

We'll have to make it so this function can now return an error in case we have problems fetching the template from disk or parsing it.

Here's the relevant changes to PlayerServer

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
    template *template.Template
}

const htmlTemplatePath = "game.html"

func NewPlayerServer(store PlayerStore) (*PlayerServer, error) {
    p := new(PlayerServer)

    tmpl, err := template.ParseFiles(htmlTemplatePath)

    if err != nil {
        return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err)
    }

    p.template = tmpl
    p.store = store

    router := http.NewServeMux()
    router.Handle("/league", http.HandlerFunc(p.leagueHandler))
    router.Handle("/players/", http.HandlerFunc(p.playersHandler))
    router.Handle("/game", http.HandlerFunc(p.game))
    router.Handle("/ws", http.HandlerFunc(p.webSocket))

    p.Handler = router

    return p, nil
}

func (p *PlayerServer) game(w http.ResponseWriter, r *http.Request) {
    p.template.Execute(w, nil)
}
```

By changing the signature of NewPlayerServer we now have compilation problems. Try and fix them yourself or refer to the source code if you struggle.

For the test code I made a helper called mustMakePlayerServer(`*testing.T, store PlayerStore`) `*PlayerServer` so that I could hide the error noise away from the tests.

```
func mustMakePlayerServer(t *testing.T, store PlayerStore) *PlayerServer {
    server, err := NewPlayerServer(store)
    if err != nil {
        t.Fatal("problem creating player server", err)
    }
    return server
}
```

Similarly, I created another helper mustDialWS so that I could hide nasty error noise when creating the WebSocket connection.

```
func mustDialWS(t *testing.T, url string) *websocket.Conn {
    ws, _, err := websocket.DefaultDialer.Dial(url, nil)
    if err != nil {
        t.Fatalf("could not open a ws connection on %s %v", url, err)
    }
    return ws
}
```

Finally, in our test code we can create a helper to tidy up sending messages

```
func writeWSMessage(t testing.TB, conn *websocket.Conn, message string) {
    t.Helper()
    if err := conn.WriteMessage(websocket.TextMessage, []byte(message)); err != nil {
        t.Fatalf("could not send message over ws connection %v", err)
    }
}
```

Now the tests are passing try running the server and declare some winners in /game. You should see them recorded in /league. Remember that every time we get a winner we close the connection, you will need to refresh the page to open the connection again.

We've made a trivial web form that lets users record the winner of a game. Let's iterate on it to make it so the user can start a game by providing a number of players and the server will push messages to the client informing them of what the blind value is as time passes.

First update game.html to update our client side code for the new requirements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Lets play poker</title>
</head>
```

```
<body>
<section id="game">
  <div id="game-start">
    <label for="player-count">Number of players</label>
    <input type="number" id="player-count"/>
    <button id="start-game">Start</button>
  </div>

  <div id="declare-winner">
    <label for="winner">Winner</label>
    <input type="text" id="winner"/>
    <button id="winner-button">Declare winner</button>
  </div>

  <div id="blind-value"/>
</section>

<section id="game-end">
  <h1>Another great game of poker everyone!</h1>
  <p><a href="/league">Go check the league table</a></p>
</section>

</body>
<script type="application/javascript">
  const startGame = document.getElementById('game-start')

  const declareWinner = document.getElementById('declare-winner')
  const submitWinnerButton = document.getElementById('winner-button')
  const winnerInput = document.getElementById('winner')

  const blindContainer = document.getElementById('blind-value')

  const gameContainer = document.getElementById('game')
  const gameEndContainer = document.getElementById('game-end')

  declareWinner.hidden = true
  gameEndContainer.hidden = true

  document.getElementById('start-game').addEventListener('click', event => {
    startGame.hidden = true
    declareWinner.hidden = false

    const numberOfPlayers = document.getElementById('player-count').value

    if (window['WebSocket']) {
      const conn = new WebSocket('ws://' + document.location.host + '/ws')
```

```

submitWinnerButton.onclick = event => {
  conn.send(winnerInput.value)
  gameEndContainer.hidden = false
  gameContainer.hidden = true
}

conn.onclose = evt => {
  blindContainer.innerText = 'Connection closed'
}

conn.onmessage = evt => {
  blindContainer.innerText = evt.data
}

conn.onopen = function () {
  conn.send(numberOfPlayers)
}
})
</script>
</html>

```

The main changes is bringing in a section to enter the number of players and a section to display the blind value. We have a little logic to show/hide the user interface depending on the stage of the game.

Any message we receive via conn.onmessage we assume to be blind alerts and so we set the blindContainer.innerText accordingly.

How do we go about sending the blind alerts? In the previous chapter we introduced the idea of Game so our CLI code could call a Game and everything else would be taken care of including scheduling blind alerts. This turned out to be a good separation of concern.

```

type Game interface {
  Start(numberOfPlayers int)
  Finish(winner string)
}

```

When the user was prompted in the CLI for number of players it would Start the game which would kick off the blind alerts and when the user declared the winner they would Finish. This is the same requirements we have now, just a different way of getting the inputs; so we should look to re-use this concept if we can.

Our "real" implementation of Game is TexasHoldem

```

type TexasHoldem struct {

```

```
    alerter BlindAlerter
    store PlayerStore
}
```

By sending in a BlindAlerter TexasHoldem can schedule blind alerts to be sent to wherever

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int)
}
```

And as a reminder, here is our implementation of the BlindAlerter we use in the CLI.

```
func StdOutAlerter(duration time.Duration, amount int) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(os.Stdout, "Blind is now %d\n", amount)
    })
}
```

This works in CLI because we always want to send the alerts to os.Stdout but this won't work for our web server. For every request we get a new http.ResponseWriter which we then upgrade to *websocket.Conn. So we can't know when constructing our dependencies where our alerts need to go.

For that reason we need to change BlindAlerter.ScheduleAlertAt so that it takes a destination for the alerts so that we can re-use it in our webserver.

Open blind_alerter.go and add the parameter to io.Writer

```
type BlindAlerter interface {
    ScheduleAlertAt(duration time.Duration, amount int, to io.Writer)
}
```

```
type BlindAlerterFunc func(duration time.Duration, amount int, to io.Writer)
```

```
func (a BlindAlerterFunc) ScheduleAlertAt(duration time.Duration, amount int, to io.Writer) {
    a(duration, amount, to)
}
```

The idea of a StdoutAlerter doesn't fit our new model so just rename it to Alerter

```
func Alerter(duration time.Duration, amount int, to io.Writer) {
    time.AfterFunc(duration, func() {
        fmt.Fprintf(to, "Blind is now %d\n", amount)
    })
}
```

If you try and compile, it will fail in TexasHoldem because it is calling ScheduleAlertAt without a destination, to get things compiling again for now hard-code it to os.Stdout.

Try and run the tests and they will fail because SpyBlindAlerter no longer implements BlindAlerter, fix this by updating the signature of ScheduleAlertAt, run the tests and we should still be green.

It doesn't make any sense for TexasHoldem to know where to send blind alerts. Let's now update Game so that when you start a game you declare where the alerts should go.

```
type Game interface {
    Start(numberOfPlayers int, alertsDestination io.Writer)
    Finish(winner string)
}
```

Let the compiler tell you what you need to fix. The change isn't so bad:

- Update TexasHoldem so it properly implements Game
- In CLI when we start the game, pass in our out property (cli.game.Start(numberOfPlayers, cli.out))
- In TexasHoldem's test i use game.Start(5, io.Discard) to fix the compilation problem and configure the alert output to be discarded

If you've got everything right, everything should be green! Now we can try and use Game within Server.

Write the test first

The requirements of CLI and Server are the same! It's just the delivery mechanism is different.

Let's take a look at our CLI test for inspiration.

```
t.Run("start game with 3 players and finish game with 'Chris' as winner", func(t *testing.T) {
    game := &GameSpy{}

    out := &bytes.Buffer{}
    in := userSends("3", "Chris wins")

    poker.NewCLI(in, out, game).PlayPoker()

    assertMessagesSentToUser(t, out, poker.PlayerPrompt)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, "Chris")
})
```

It looks like we should be able to test drive out a similar outcome using GameSpy

Replace the old websocket test with the following

```
t.Run("start a game with 3 players and declare Ruth the winner", func(t *testing.T) {
    game := &poker.GameSpy{}
    winner := "Ruth"
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()

    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(10 * time.Millisecond)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)
})
```

- As discussed we create a spy Game and pass it into mustMakePlayerServer (be sure to update the helper to support this).
- We then send the web socket messages for a game.
- Finally we assert that the game is started and finished with what we expect.

Try to run the test

You'll have a number of compilation errors around mustMakePlayerServer in other tests. Introduce an unexported variable dummyGame and use it through all the tests that aren't compiling

```
var (
    dummyGame = &GameSpy{}
```

The final error is where we are trying to pass in Game to NewPlayerServer but it doesn't support it yet

```
./server_test.go:21:38: too many arguments in call to "github.com/quii/learn-go-with-tests/WebS
    have ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore, "github.com/quii/lear
    want ("github.com/quii/learn-go-with-tests/WebSockets/v2".PlayerStore)
```

Write the minimal amount of code for the test to run and check the failing test output

Just add it as an argument for now just to get the test running

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error)
```

Finally!

```
==== RUN TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner
--- FAIL: TestGame (0.01s)
    --- FAIL: TestGame/start_a_game_with_3_players_and_declare_Ruth_the_winner (0.01s)
        server_test.go:146: wanted Start called with 3 but got 0
        server_test.go:147: expected finish called with 'Ruth' but got ""
FAIL
```

Write enough code to make it pass

We need to add Game as a field to PlayerServer so that it can use it when it gets requests.

```
type PlayerServer struct {
    store PlayerStore
    http.Handler
    template *template.Template
    game    Game
}
```

(We already have a method called game so rename that to playGame)

Next lets assign it in our constructor

```
func NewPlayerServer(store PlayerStore, game Game) (*PlayerServer, error) {
    p := new(PlayerServer)

    tmpl, err := template.ParseFiles(htmlTemplatePath)

    if err != nil {
        return nil, fmt.Errorf("problem opening %s %v", htmlTemplatePath, err)
    }

    p.game = game

    // etc
}
```

Now we can use our Game within webSocket.

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    conn, _ := wsUpgrader.Upgrade(w, r, nil)
```

```
_ , numberOfPlayersMsg, _ := conn.ReadMessage()
numberOfPlayers, _ := strconv.Atoi(string(numberOfPlayersMsg))
p.game.Start(numberOfPlayers, io.Discard) //todo: Don't discard the blinds messages!

_ , winner, _ := conn.ReadMessage()
p.game.Finish(string(winner))
}
```

Hooray! The tests pass.

We are not going to send the blind messages anywhere just yet as we need to have a think about that. When we call game.Start we send in io.Discard which will just discard any messages written to it.

For now start the web server up. You'll need to update the main.go to pass a Game to the PlayerServer

```
func main() {
    db, err := os.OpenFile(dbFileName, os.O_RDWR|os.O_CREATE, 0666)

    if err != nil {
        log.Fatalf("problem opening %s %v", dbFileName, err)
    }

    store, err := poker.NewFileSystemPlayerStore(db)

    if err != nil {
        log.Fatalf("problem creating file system player store, %v ", err)
    }

    game := poker.NewTexasHoldem(poker.BlindAlerterFunc(poker.Alerter), store)

    server, err := poker.NewPlayerServer(store, game)

    if err != nil {
        log.Fatalf("problem creating player server %v", err)
    }

    log.Fatal(http.ListenAndServe(":5000", server))
}
```

Discounting the fact we're not getting blind alerts yet, the app does work! We've managed to re-use Game with PlayerServer and it has taken care of all the details. Once we figure out how to send our blind alerts through to the web sockets rather than discarding them it should all work.

Before that though, let's tidy up some code.

Refactor

The way we're using WebSockets is fairly basic and the error handling is fairly naive, so I wanted to encapsulate that in a type just to remove that messiness from the server code. We may wish to revisit it later but for now this'll tidy things up a bit

```
type playerServerWS struct {
    *websocket.Conn
}

func newPlayerServerWS(w http.ResponseWriter, r *http.Request) *playerServerWS {
    conn, err := wsUpgrader.Upgrade(w, r, nil)

    if err != nil {
        log.Printf("problem upgrading connection to WebSockets %v\n", err)
    }

    return &playerServerWS{conn}
}

func (w *playerServerWS) WaitForMsg() string {
    _, msg, err := w.ReadMessage()
    if err != nil {
        log.Printf("error reading from websocket %v\n", err)
    }
    return string(msg)
}
```

Now the server code is a bit simplified

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    ws := newPlayerServerWS(w, r)

    numberOfPlayersMsg := ws.WaitForMsg()
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
    p.game.Start(numberOfPlayers, io.Discard) //todo: Don't discard the blinds messages!

    winner := ws.WaitForMsg()
    p.game.Finish(winner)
}
```

Once we figure out how to not discard the blind messages we're done.

Let's not write a test!

Sometimes when we're not sure how to do something, it's best just to play around and try things out! Make sure your work is committed

first because once we've figured out a way we should drive it through a test.

The problematic line of code we have is

```
p.game.Start(numberOfPlayers, io.Discard) //todo: Don't discard the blinds messages!
```

We need to pass in an io.Writer for the game to write the blind alerts to.

Wouldn't it be nice if we could pass in our playerServerWS from before? It's our wrapper around our WebSocket so it feels like we should be able to send that to our Game to send messages to.

Give it a go:

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {  
    ws := newPlayerServerWS(w, r)  
  
    numberOfPlayersMsg := ws.WaitForMsg()  
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)  
    p.game.Start(numberOfPlayers, ws)  
    //etc...  
}
```

The compiler complains

```
./server.go:71:14: cannot use ws (type *playerServerWS) as type io.Writer in argument to p.game.Start  
    *playerServerWS does not implement io.Writer (missing Write method)
```

It seems the obvious thing to do, would be to make it so playerServerWS does implement io.Writer. To do so we use the underlying *websocket.Conn to use WriteMessage to send the message down the websocket

```
func (w *playerServerWS) Write(p []byte) (n int, err error) {  
    err = w.WriteMessage(websocket.TextMessage, p)  
  
    if err != nil {  
        return 0, err  
    }  
  
    return len(p), nil  
}
```

This seems too easy! Try and run the application and see if it works.

Beforehand edit TexasHoldem so that the blind increment time is shorter so you can see it in action

```
blindIncrement := time.Duration(5+numberOfPlayers) * time.Second // (rather than a minute)
```

You should see it working! The blind amount increments in the browser as if by magic.

Now let's revert the code and think how to test it. In order to implement it all we did was pass through to StartGame was playerServerWS rather than io.Discard so that might make you think we should perhaps spy on the call to verify it works.

Spying is great and helps us check implementation details but we should always try and favour testing the real behaviour if we can because when you decide to refactor it's often spy tests that start failing because they are usually checking implementation details that you're trying to change.

Our test currently opens a websocket connection to our running server and sends messages to make it do things. Equally we should be able to test the messages our server sends back over the websocket connection.

Write the test first

We'll edit our existing test.

Currently, our GameSpy does not send any data to out when you call Start. We should change it so we can configure it to send a canned message and then we can check that message gets sent to the websocket. This should give us confidence that we have configured things correctly whilst still exercising the real behaviour we want.

```
type GameSpy struct {
    StartCalled    bool
    StartCalledWith int
    BlindAlert    []byte

    FinishedCalled bool
    FinishCalledWith string
}
```

Add BlindAlert field.

Update GameSpy Start to send the canned message to out.

```
func (g *GameSpy) Start(numberOfPlayers int, out io.Writer) {
    g.StartCalled = true
    g.StartCalledWith = numberOfPlayers
    out.Write(g.BlindAlert)
}
```

This now means when we exercise PlayerServer when it tries to Start the game it should end up sending messages through the websocket

if things are working right.

Finally, we can update the test

```
t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth the winner")
    wantedBlindAlert := "Blind is 100"
    winner := "Ruth"

    game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()

    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(10 * time.Millisecond)
    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)

    _, gotBlindAlert, _ := ws.ReadMessage()

    if string(gotBlindAlert) != wantedBlindAlert {
        t.Errorf("got blind alert %q, want %q", string(gotBlindAlert), wantedBlindAlert)
    }
}

• We've added a wantedBlindAlert and configured our GameSpy to send it to out if Start is called.
• We hope it gets sent in the websocket connection so we've added a call to ws.ReadMessage() to wait for a message to be sent and then check it's the one we expected.
```

Try to run the test

You should find the test hangs forever. This is because ws.ReadMessage() will block until it gets a message, which it never will.

Write the minimal amount of code for the test to run and check the failing test output

We should never have tests that hang so let's introduce a way of handling code that we want to timeout.

```

func within(t testing.TB, d time.Duration, assert func()) {
    t.Helper()

    done := make(chan struct{}, 1)

    go func() {
        assert()
        done <- struct{}{}
    }()

    select {
        case <-time.After(d):
            t.Error("timed out")
        case <-done:
    }
}

```

What within does is take a function assert as an argument and then runs it in a go routine. If/When the function finishes it will signal it is done via the done channel.

While that happens we use a select statement which lets us wait for a channel to send a message. From here it is a race between the assert function and time.After which will send a signal when the duration has occurred.

Finally, I made a helper function for our assertion just to make things a bit neater

```

func assertWebsocketGotMsg(t *testing.T, ws *websocket.Conn, want string) {
    _, msg, _ := ws.ReadMessage()
    if string(msg) != want {
        t.Errorf(`got "%s", want "%s"`, string(msg), want)
    }
}

```

Here's how the test reads now

```

t.Run("start a game with 3 players, send some blind alerts down WS and declare Ruth the winner",
    wantedBlindAlert := "Blind is 100"
    winner := "Ruth"

    game := &GameSpy{BlindAlert: []byte(wantedBlindAlert)}
    server := httptest.NewServer(mustMakePlayerServer(t, dummyPlayerStore, game))
    ws := mustDialWS(t, "ws"+strings.TrimPrefix(server.URL, "http")+"/ws")

    defer server.Close()
    defer ws.Close()
)

```

```
    writeWSMessage(t, ws, "3")
    writeWSMessage(t, ws, winner)

    time.Sleep(tenMS)

    assertGameStartedWith(t, game, 3)
    assertFinishCalledWith(t, game, winner)
    within(t, tenMS, func() { assertWebsocketGotMsg(t, ws, wantedBlindAlert) })
})
```

Now if you run the test...

```
==== RUN TestGame
==== RUN TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_and_de
--- FAIL: TestGame (0.02s)
--- FAIL: TestGame/start_a_game_with_3_players,_send_some_blind_alerts_down_WS_and_de
    server_test.go:143: timed out
    server_test.go:150: got "", want "Blind is 100"
```

Write enough code to make it pass

Finally, we can now change our server code, so it sends our Web-
Socket connection to the game when it starts

```
func (p *PlayerServer) webSocket(w http.ResponseWriter, r *http.Request) {
    ws := newPlayerServerWS(w, r)

    numberOfPlayersMsg := ws.WaitForMsg()
    numberOfPlayers, _ := strconv.Atoi(numberOfPlayersMsg)
    p.game.Start(numberOfPlayers, ws)

    winner := ws.WaitForMsg()
    p.game.Finish(winner)
}
```

Refactor

The server code was a very small change so there's not a lot to change here but the test code still has a time.Sleep call because we have to wait for our server to do its work asynchronously.

We can refactor our helpers assertGameStartedWith and assertFinishCalledWith so that they can retry their assertions for a short period before failing.

Here's how you can do it for assertFinishCalledWith and you can use the same approach for the other helper.

```
func assertFinishCalledWith(t testing.TB, game *GameSpy, winner string) {
    t.Helper()

    passed := retryUntil(500*time.Millisecond, func() bool {
        return game.FinishCalledWith == winner
    })

    if !passed {
        t.Errorf("expected finish called with %q but got %q", winner, game.FinishCalledWith)
    }
}
```

Here is how retryUntil is defined

```
func retryUntil(d time.Duration, f func() bool) bool {
    deadline := time.Now().Add(d)
    for time.Now().Before(deadline) {
        if f() {
            return true
        }
    }
    return false
}
```

Wrapping up

Our application is now complete. A game of poker can be started via a web browser and the users are informed of the blind bet value as time goes by via WebSockets. When the game finishes they can record the winner which is persisted using code we wrote a few chapters ago. The players can find out who is the best (or luckiest) poker player using the website's /league endpoint.

Through the journey we have made mistakes but with the TDD flow we have never been very far away from working software. We were free to keep iterating and experimenting.

The final chapter will retrospect on the approach, the design we've arrived at and tie up some loose ends.

We covered a few things in this chapter

WebSockets

- Convenient way of sending messages between clients and servers that does not require the client to keep polling the server. Both the client and server code we have is very simple.

-
- Trivial to test, but you have to be wary of the asynchronous nature of the tests

Handling code in tests that can be delayed or never finish

- Create helper functions to retry assertions and add timeouts.
- We can use go routines to ensure the assertions don't block anything and then use channels to let them signal that they have finished, or not.
- The time package has some helpful functions which also send signals via channels about events in time so we can set timeouts

OS Exec

You can find all the code here

keith6014 asks on [reddit](#)

I am executing a command using `os/exec.Command()` which generated XML data. The command will be executed in a function called `GetData()`.

In order to test `GetData()`, I have some testdata which I created.

In my `_test.go` I have a `TestGetData` which calls `GetData()` but that will use `os.exec`, instead I would like for it to use my testdata.

What is a good way to achieve this? When calling `GetData` should I have a "test" flag mode so it will read a file ie `GetData(mode string)?`

A few things

- When something is difficult to test, it's often due to the separation of concerns not being quite right
- Don't add "test modes" into your code, instead use [Dependency Injection](#) so that you can model your dependencies and separate concerns.

I have taken the liberty of guessing what the code might look like

```
type Payload struct {
    Message string `xml:"message"`
}

func GetData() string {
    cmd := exec.Command("cat", "msg.xml")
```

```

out, _ := cmd.StdoutPipe()
var payload Payload
decoder := xml.NewDecoder(out)

// these 3 can return errors but I'm ignoring for brevity
cmd.Start()
decoder.Decode(&payload)
cmd.Wait()

return strings.ToUpper(payload.Message)
}

```

- It uses exec.Command which allows you to execute an external command to the process
- We capture the output in cmd.StdoutPipe which returns us a io.ReadCloser (this will become important)
- The rest of the code is more or less copy and pasted from the [excellent documentation](#).
 - We capture any output from stdout into an io.ReadCloser and then we Start the command and then wait for all the data to be read by calling Wait. In between those two calls we decode the data into our Payload struct.

Here is what is contained inside msg.xml

```

<payload>
  <message>Happy New Year!</message>
</payload>

```

I wrote a simple test to show it in action

```

func TestGetData(t *testing.T) {
  got := GetData()
  want := "HAPPY NEW YEAR!"

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

```

Testable code

Testable code is decoupled and single purpose. To me it feels like there are two main concerns for this code

1. Retrieving the raw XML data

-
2. Decoding the XML data and applying our business logic (in this case strings.ToUpper on the <message>)

The first part is just copying the example from the standard lib.

The second part is where we have our business logic and by looking at the code we can see where the "seam" in our logic starts; it's where we get our io.ReadCloser. We can use this existing abstraction to separate concerns and make our code testable.

The problem with GetData is the business logic is coupled with the means of getting the XML. To make our design better we need to decouple them

Our TestGetData can act as our integration test between our two concerns so we'll keep hold of that to make sure it keeps working.

Here is what the newly separated code looks like

```
type Payload struct {
    Message string `xml:"message"`
}

func GetData(data io.Reader) string {
    var payload Payload
    xml.NewDecoder(data).Decode(&payload)
    return strings.ToUpper(payload.Message)
}

func getXMLFromCommand() io.Reader {
    cmd := exec.Command("cat", "msg.xml")
    out, _ := cmd.StdoutPipe()

    cmd.Start()
    data, _ := io.ReadAll(out)
    cmd.Wait()

    return bytes.NewReader(data)
}

func TestGetDataIntegration(t *testing.T) {
    got := GetData(getXMLFromCommand())
    want := "HAPPY NEW YEAR!"

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

Now that GetData takes its input from just an io.Reader we have made it testable and it is no longer concerned how the data is retrieved; people can re-use the function with anything that returns an io.Reader (which is extremely common). For example we could start fetching the XML from a URL instead of the command line.

```
func TestGetData(t *testing.T) {
    input := strings.NewReader(`<payload>
        <message>Cats are the best animal</message>
    </payload>`)

    got := GetData(input)
    want := "CATS ARE THE BEST ANIMAL"

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

Here is an example of a unit test for GetData.

By separating the concerns and using existing abstractions within Go testing our important business logic is a breeze.

Error types

[You can find all the code here](#)

Creating your own types for errors can be an elegant way of tidying up your code, making your code easier to use and test.

Pedro on the Gopher Slack asks

If I'm creating an error like fmt.Errorf("%s must be foo, got %s", bar, baz), is there a way to test equality without comparing the string value?

Let's make up a function to help explore this idea.

```
// DumbGetter will get the string body of url if it gets a 200
func DumbGetter(url string) (string, error) {
    res, err := http.Get(url)

    if err != nil {
        return "", fmt.Errorf("problem fetching from %s, %v", url, err)
    }
```

```
if res.StatusCode != http.StatusOK {
    return "", fmt.Errorf("did not get 200 from %s, got %d", url, res.StatusCode)
}

defer res.Body.Close()
body, _ := io.ReadAll(res.Body) // ignoring err for brevity

return string(body), nil
}
```

It's not uncommon to write a function that might fail for different reasons and we want to make sure we handle each scenario correctly.

As Pedro says, we could write a test for the status error like so.

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {
    svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req *http.Request) {
        res.WriteHeader(http.StatusTeapot)
    }))
    defer svr.Close()

    _, err := DumbGetter(svr.URL)

    if err == nil {
        t.Fatal("expected an error")
    }

    want := fmt.Sprintf("did not get 200 from %s, got %d", svr.URL, http.StatusTeapot)
    got := err.Error()

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

This test creates a server which always returns StatusTeapot and then we use its URL as the argument to DumbGetter so we can see it handles non 200 responses correctly.

Problems with this way of testing

This book tries to emphasise listen to your tests and this test doesn't feel good:

- We're constructing the same string as production code does to test it
- It's annoying to read and write

-
- Is the exact error message string what we're actually concerned with ?

What does this tell us? The ergonomics of our test would be reflected on another bit of code trying to use our code.

How does a user of our code react to the specific kind of errors we return? The best they can do is look at the error string which is extremely error prone and horrible to write.

What we should do

With TDD we have the benefit of getting into the mindset of:

How would I want to use this code?

What we could do for DumbGetter is provide a way for users to use the type system to understand what kind of error has happened.

What if DumbGetter could return us something like

```
type BadStatusError struct {
    URL  string
    Status int
}
```

Rather than a magical string, we have actual data to work with.

Let's change our existing test to reflect this need

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {
    svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req *http.Request) {
        res.WriteHeader(http.StatusTeapot)
    }))
    defer svr.Close()

    _, err := DumbGetter(svr.URL)

    if err == nil {
        t.Fatal("expected an error")
    }

    got, isStatusErr := err.(BadStatusError)

    if !isStatusErr {
        t.Fatalf("was not a BadStatusError, got %T", err)
    }

    want := BadStatusError{URL: svr.URL, Status: http.StatusTeapot}
})
```

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

We'll have to make `BadStatusError` implement the error interface.

```
func (b BadStatusError) Error() string {
    return fmt.Sprintf("did not get 200 from %s, got %d", b.URL, b.Status)
}
```

What does the test do?

Instead of checking the exact string of the error, we are doing a [type assertion](#) on the error to see if it is a `BadStatusError`. This reflects our desire for the kind of error clearer. Assuming the assertion passes we can then check the properties of the error are correct.

When we run the test, it tells us we didn't return the right kind of error

```
--- FAIL: TestDumbGetter (0.00s)
    --- FAIL: TestDumbGetter/when_you_dont_get_a_200_you_get_a_status_error (0.00s)
        error-types_test.go:56: was not a BadStatusError, got *errors.errorString
```

Let's fix `DumbGetter` by updating our error handling code to use our type

```
if res.StatusCode != http.StatusOK {
    return "", BadStatusError{URL: url, Status: res.StatusCode}
}
```

This change has had some real positive effects

- Our `DumbGetter` function has become simpler, it's no longer concerned with the intricacies of an error string, it just creates a `BadStatusError`.
- Our tests now reflect (and document) what a user of our code could do if they decided they wanted to do some more sophisticated error handling than just logging. Just do a type assertion and then you get easy access to the properties of the error.
- It is still "just" an error, so if they choose to they can pass it up the call stack or log it like any other error.

Wrapping up

If you find yourself testing for multiple error conditions don't fall in to the trap of comparing the error messages.

This leads to flaky and difficult to read/write tests and it reflects the difficulties the users of your code will have if they also need to start doing things differently depending on the kind of errors that have occurred.

Always make sure your tests reflect how you'd like to use your code, so in this respect consider creating error types to encapsulate your kinds of errors. This makes handling different kinds of errors easier for users of your code and also makes writing your error handling code simpler and easier to read.

Addendum

As of Go 1.13 there are new ways to work with errors in the standard library which is covered in the [Go Blog](#)

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {  
    svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req *http.Request)  
        res.WriteHeader(http.StatusTeapot)  
    )))  
    defer svr.Close()  
  
    _, err := DumbGetter(svr.URL)  
  
    if err == nil {  
        t.Fatal("expected an error")  
    }  
  
    var got BadStatusError  
    isBadStatusError := errors.As(err, &got)  
    want := BadStatusError{URL: svr.URL, Status: http.StatusTeapot}  
  
    if !isBadStatusError {  
        t.Fatalf("was not a BadStatusError, got %T", err)  
    }  
  
    if got != want {  
        t.Errorf("got %v, want %v", got, want)  
    }  
})
```

In this case we are using `errors.As` to try and extract our error into our custom type. It returns a bool to denote success and extracts it into `got` for us.

Context-aware readers

[You can find all the code here](#)

This chapter demonstrates how to test-drive a context aware io.Reader as written by Mat Ryer and David Hernandez in [The Pace Dev Blog](#).

Context aware reader?

First of all, a quick primer on io.Reader.

If you've read other chapters in this book you will have ran into io.Reader when we've opened files, encoded JSON and various other common tasks. It's a simple abstraction over reading data from something

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

By using io.Reader you can gain a lot of re-use from the standard library, it's a very commonly used abstraction (along with its counterpart io.Writer)

Context aware?

[In a previous chapter](#) we discussed how we can use context to provide cancellation. This is especially useful if you're performing tasks which may be computationally expensive and you want to be able to stop them.

When you're using an io.Reader you have no guarantees over speed, it could take 1 nanosecond or hundreds of hours. You might find it useful to be able to cancel these kind of tasks in your own application and that's what Mat and David wrote about.

They combined two simple abstractions (context.Context and io.Reader) to solve this problem.

Let's try and TDD some functionality so that we can wrap an io.Reader so it can be cancelled.

Testing this poses an interesting challenge. Normally when using an io.Reader you're usually supplying it to some other function and you don't really concern yourself with the details; such as json.NewDecoder or io.ReadAll.

What we want to demonstrate is something like

Given an io.Reader with "ABCDEF", when I send a cancel signal half-way through I when I try to continue to read I get nothing else so all I get is "ABC"

Let's look at the interface again.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

The Reader's Read method will read the contents it has into a []byte that we supply.

So rather than reading everything, we could:

- Supply a fixed-size byte array that doesn't fit all the contents
- Send a cancel signal
- Try and read again and this should return an error with 0 bytes read

For now, let's just write a "happy path" test where there is no cancellation, just so we can get familiar with the problem without having to write any production code yet.

```
func TestContextAwareReader(t *testing.T) {
    t.Run("lets just see how a normal reader works", func(t *testing.T) {
        rdr := strings.NewReader("123456")
        got := make([]byte, 3)
        _, err := rdr.Read(got)

        if err != nil {
            t.Fatal(err)
        }

        assertBufferHas(t, got, "123")

        _, err = rdr.Read(got)

        if err != nil {
            t.Fatal(err)
        }

        assertBufferHas(t, got, "456")
    })
}

func assertBufferHas(t testing.TB, buf []byte, want string) {
    t.Helper()
    got := string(buf)
```

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

• Make an io.Reader from a string with some data
• A byte array to read into which is smaller than the contents of
  the reader
• Call read, check the contents, repeat.
```

From this we can imagine sending some kind of cancel signal before the second read to change behaviour.

Now we've seen how it works we'll TDD the rest of the functionality.

Write the test first

We want to be able to compose an io.Reader with a context.Context.

With TDD it's best to start with imagining your desired API and write a test for it.

From there let the compiler and failing test output can guide us to a solution

```
t.Run("behaves like a normal reader", func(t *testing.T) {
    rdr := NewCancellableReader(strings.NewReader("123456"))
    got := make([]byte, 3)
    _, err := rdr.Read(got)

    if err != nil {
        t.Fatal(err)
    }

    assertBufferHas(t, got, "123")
    _, err = rdr.Read(got)

    if err != nil {
        t.Fatal(err)
    }

    assertBufferHas(t, got, "456")
})
```

Try to run the test

```
./cancel_readers_test.go:12:10: undefined: NewCancellableReader
```

Write the minimal amount of code for the test to run and check the failing test output

We'll need to define this function and it should return an io.Reader

```
func NewCancellableReader(rdr io.Reader) io.Reader {  
    return nil  
}
```

If you try and run it

```
==== RUN  TestCancelReaders  
==== RUN  TestCancelReaders/behaves_like_a_normal_reader  
panic: runtime error: invalid memory address or nil pointer dereference [recovered]  
    panic: runtime error: invalid memory address or nil pointer dereference  
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x10f8fb5]
```

As expected

Write enough code to make it pass

For now, we'll just return the io.Reader we pass in

```
func NewCancellableReader(rdr io.Reader) io.Reader {  
    return rdr  
}
```

The test should now pass.

I know, I know, this seems silly and pedantic but before charging in to the fancy work it is important that we have some verification that we haven't broken the "normal" behaviour of an io.Reader and this test will give us confidence as we move forward.

Write the test first

Next we need to try and cancel.

```
t.Run("stops reading when cancelled", func(t *testing.T) {  
    ctx, cancel := context.WithCancel(context.Background())  
    rdr := NewCancellableReader(ctx, strings.NewReader("123456"))  
    got := make([]byte, 3)  
    _, err := rdr.Read(got)  
  
    if err != nil {  
        t.Fatal(err)  
    }  
  
    assertBufferHas(t, got, "123")
```

```
cancel()

n, err := rdr.Read(got)

if err == nil {
    t.Error("expected an error after cancellation but didnt get one")
}

if n > 0 {
    t.Errorf("expected 0 bytes to be read after cancellation but %d were read", n)
}
})
```

We can more or less copy the first test but now we're:

- Creating a context.Context with cancellation so we can cancel after the first read
- For our code to work we'll need to pass ctx to our function
- We then assert that post-cancel nothing was read

Try to run the test

```
./cancel_readers_test.go:33:30: too many arguments in call to NewCancellableReader
have (context.Context, *strings.Reader)
want (io.Reader)
```

Write the minimal amount of code for the test to run and check the failing test output

The compiler is telling us what to do; update our signature to accept a context

```
func NewCancellableReader(ctx context.Context, rdr io.Reader) io.Reader {
    return rdr
}
```

(You'll need to update the first test to pass in context.Background too)

You should now see a very clear failing test output

```
==== RUN TestCancelReaders
==== RUN TestCancelReaders/stops_reading_when_cancelled
--- FAIL: TestCancelReaders (0.00s)
    --- FAIL: TestCancelReaders/stops_reading_when_cancelled (0.00s)
        cancel_readers_test.go:48: expected an error but didnt get one
        cancel_readers_test.go:52: expected 0 bytes to be read after cancellation but 3 were read
```

Write enough code to make it pass

At this point, it's copy and paste from the original post by Mat and David but we'll still take it slowly and iteratively.

We know we need to have a type that encapsulates the io.Reader that we read from and the context.Context so let's create that and try and return it from our function instead of the original io.Reader

```
func NewCancellableReader(ctx context.Context, rdr io.Reader) io.Reader {
    return &readerCtx{
        ctx:   ctx,
        delegate: rdr,
    }
}

type readerCtx struct {
    ctx   context.Context
    delegate io.Reader
}
```

As I have stressed many times in this book, go slowly and let the compiler help you

```
./cancel_readers_test.go:60:3: cannot use &readerCtx literal (type *readerCtx) as type io.Reader
    *readerCtx does not implement io.Reader (missing Read method)
```

The abstraction feels right, but it doesn't implement the interface we need (io.Reader) so let's add the method.

```
func (r *readerCtx) Read(p []byte) (n int, err error) {
    panic("implement me")
}
```

Run the tests and they should compile but panic. This is still progress.

Let's make the first test pass by just delegating the call to our underlying io.Reader

```
func (r readerCtx) Read(p []byte) (n int, err error) {
    return r.delegate.Read(p)
}
```

At this point we have our happy path test passing again and it feels like we have our stuff abstracted nicely

To make our second test pass we need to check the context.Context to see if it has been cancelled.

```
func (r readerCtx) Read(p []byte) (n int, err error) {
    if err := r.ctx.Err(); err != nil {
        return 0, err
    }
    // ...
}
```

```
    }
    return r.delegate.Read(p)
}
```

All tests should now pass. You'll notice how we return the error from the context.Context. This allows callers of the code to inspect the various reasons cancellation has occurred and this is covered more in the original post.

Wrapping up

- Small interfaces are good and are easily composed
- When you're trying to augment one thing (e.g io.Reader) with another you usually want to reach for the [delegation pattern](#)

In software engineering, the delegation pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance.

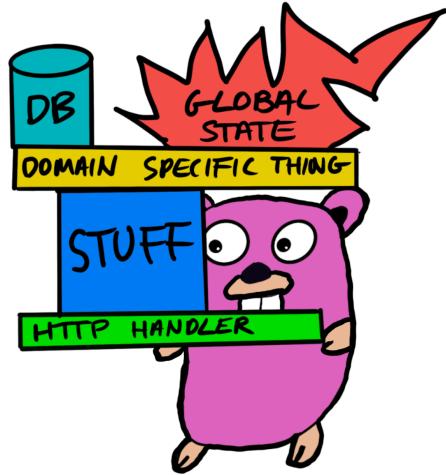
- An easy way to start this kind of work is to wrap your delegate and write a test that asserts it behaves how the delegate normally does before you start composing other parts to change behaviour. This will help you to keep things working correctly as you code toward your goal

HTTP Handlers Revisited

[You can find all the code here](#)

This book already has a chapter on [testing a HTTP handler](#) but this will feature a broader discussion on designing them, so they are simple to test.

We'll take a look at a real example and how we can improve how it's designed by applying principles such as single responsibility principle and separation of concerns. These principles can be realised by using [interfaces](#) and [dependency injection](#). By doing this we'll show how testing handlers is actually quite trivial.



"HOW DO I TEST MY HTTP HANDLER?"

Testing HTTP handlers seems to be a recurring question in the Go community, and I think it points to a wider problem of people misunderstanding how to design them.

So often people's difficulties with testing stems from the design of their code rather than the actual writing of tests. As I stress so often in this book:

If your tests are causing you pain, listen to that signal and think about the design of your code.

An example

Santosh Kumar tweeted me

How do I test a http handler which has mongodb dependency?

Here is the code

```
func Registration(w http.ResponseWriter, r *http.Request) {
    var res model.ResponseResult
    var user model.User

    w.Header().Set("Content-Type", "application/json")

    jsonDecoder := json.NewDecoder(r.Body)
    jsonDecoder.DisallowUnknownFields()
    defer r.Body.Close()

    // check if there is proper json body or error
    if err := jsonDecoder.Decode(&user); err != nil {
```

```
res.Error = err.Error()
// return 400 status codes
w.WriteHeader(http.StatusBadRequest)
json.NewEncoder(w).Encode(res)
return
}

// Connect to mongodb
client, _ := mongo.NewClient(options.Client().ApplyURI("mongodb://127.0.0.1:27017"))
ctx, _ := context.WithTimeout(context.Background(), 10*time.Second)
err := client.Connect(ctx)
if err != nil {
    panic(err)
}
defer client.Disconnect(ctx)
// Check if username already exists in users datastore, if so, 400
// else insert user right away
collection := client.Database("test").Collection("users")
filter := bson.D{{"username", user.Username}}
var foundUser model.User
err = collection.FindOne(context.TODO(), filter).Decode(&foundUser)
if foundUser.Username == user.Username {
    res.Error = UserExists
    // return 400 status codes
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(res)
    return
}

pass, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)
if err != nil {
    res.Error = err.Error()
    // return 400 status codes
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(res)
    return
}
user.Password = string(pass)

insertResult, err := collection.InsertOne(context.TODO(), user)
if err != nil {
    res.Error = err.Error()
    // return 400 status codes
    w.WriteHeader(http.StatusBadRequest)
    json.NewEncoder(w).Encode(res)
    return
}
```

```
    }

// return 200
w.WriteHeader(http.StatusOK)
res.Result = fmt.Sprintf("%s: %s", UserCreated, insertResult.InsertedID)
json.NewEncoder(w).Encode(res)
return
}
```

Let's just list all the things this one function has to do:

1. Write HTTP responses, send headers, status codes, etc.
2. Decode the request's body into a User
3. Connect to a database (and all the details around that)
4. Query the database and applying some business logic depending on the result
5. Generate a password
6. Insert a record

This is too much.

What is a HTTP Handler and what should it do ?

Forgetting specific Go details for a moment, no matter what language I've worked in what has always served me well is thinking about the [separation of concerns](#) and the [single responsibility principle](#).

This can be quite tricky to apply depending on the problem you're solving. What exactly is a responsibility?

The lines can blur depending on how abstractly you're thinking and sometimes your first guess might not be right.

Thankfully with HTTP handlers I feel like I have a pretty good idea what they should do, no matter what project I've worked on:

1. Accept a HTTP request, parse and validate it.
2. Call some ServiceThing to do ImportantBusinessLogic with the data I got from step 1.
3. Send an appropriate HTTP response depending on what ServiceThing returns.

I'm not saying every HTTP handler ever should have roughly this shape, but 99 times out of 100 that seems to be the case for me.

When you separate these concerns:

- Testing handlers becomes a breeze and is focused a small number of concerns.

-
- Importantly testing ImportantBusinessLogic no longer has to concern itself with HTTP, you can test the business logic cleanly.
 - You can use ImportantBusinessLogic in other contexts without having to modify it.
 - If ImportantBusinessLogic changes what it does, so long as the interface remains the same you don't have to change your handlers.

Go's Handlers

[http.HandlerFunc](#)

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers.

```
type HandlerFunc func(ResponseWriter, *Request)
```

Reader, take a breath and look at the code above. What do you notice?

It is a function that takes some arguments

There's no framework magic, no annotations, no magic beans, nothing.

It's just a function, and we know how to test functions.

It fits in nicely with the commentary above:

- It takes a [http.Request](#) which is just a bundle of data for us to inspect, parse and validate.
- A [http.ResponseWriter](#) interface is used by an HTTP handler to construct an HTTP response.

Super basic example test

```
func Teapot(res http.ResponseWriter, req *http.Request) {
    res.WriteHeader(http.StatusTeapot)
}

func TestTeapotHandler(t *testing.T) {
    req := httptest.NewRequest(http.MethodGet, "/", nil)
    res := httptest.NewRecorder()

    Teapot(res, req)

    if res.Code != http.StatusTeapot {
        t.Errorf("got status %d but wanted %d", res.Code, http.StatusTeapot)
    }
}
```

```
    }  
}
```

To test our function, we call it.

For our test we pass a `httptest.ResponseRecorder` as our `http.ResponseWriter` argument, and our function will use it to write the HTTP response. The recorder will record (or spy on) what was sent, and then we can make our assertions.

Calling a ServiceThing in our handler

A common complaint about TDD tutorials is that they're always "too simple" and not "real world enough". My answer to that is:

Wouldn't it be nice if all your code was simple to read and test like the examples you mention?

This is one of the biggest challenges we face but need to keep striving for. It is possible (although not necessarily easy) to design code, so it can be simple to read and test if we practice and apply good software engineering principles.

Recapping what the handler from earlier does:

1. Write HTTP responses, send headers, status codes, etc.
2. Decode the request's body into a User
3. Connect to a database (and all the details around that)
4. Query the database and applying some business logic depending on the result
5. Generate a password
6. Insert a record

Taking the idea of a more ideal separation of concerns I'd want it to be more like:

1. Decode the request's body into a User
2. Call a `UserService.Register(user)` (this is our ServiceThing)
3. If there's an error act on it (the example always sends a 400 `BadRequest` which I don't think is right), I'll just have a catch-all handler of a 500 Internal Server Error for now. I must stress that returning 500 for all errors makes for a terrible API! Later on we can make the error handling more sophisticated, perhaps with [error types](#).
4. If there's no error, 201 Created with the ID as the response body (again for terseness/laziness)

For the sake of brevity I won't go over the usual TDD process, check all the other chapters for examples.

New design

```
type UserService interface {
    Register(user User) (insertedID string, err error)
}

type UserServer struct {
    service.UserService
}

func NewUserServer(service UserService) *UserServer {
    return &UserServer{service: service}
}

func (u *UserServer) RegisterUser(w http.ResponseWriter, r *http.Request) {
    defer r.Body.Close()

    // request parsing and validation
    var newUser User
    err := json.NewDecoder(r.Body).Decode(&newUser)

    if err != nil {
        http.Error(w, fmt.Sprintf("could not decode user payload: %v", err), http.StatusBadRequest)
        return
    }

    // call a service thing to take care of the hard work
    insertedID, err := u.service.Register(newUser)

    // depending on what we get back, respond accordingly
    if err != nil {
        //todo: handle different kinds of errors differently
        http.Error(w, fmt.Sprintf("problem registering new user: %v", err), http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated)
    fmt.Fprint(w, insertedID)
}
```

Our RegisterUser method matches the shape of http.HandlerFunc so we're good to go. We've attached it as a method on a new type UserServer which contains a dependency on a UserService which is captured as an interface.

Interfaces are a fantastic way to ensure our HTTP concerns are decoupled from any specific implementation; we can just call the method

on the dependency, and we don't have to care how a user gets registered.

If you wish to explore this approach in more detail following TDD read the [Dependency Injection](#) chapter and the [HTTP Server chapter of the "Build an application" section](#).

Now that we've decoupled ourselves from any specific implementation detail around registration writing the code for our handler is straightforward and follows the responsibilities described earlier.

The tests!

This simplicity is reflected in our tests.

```
type MockUserService struct {
    RegisterFunc func(user User) (string, error)
    UsersRegistered []User
}

func (m *MockUserService) Register(user User) (insertedID string, err error) {
    m.UsersRegistered = append(m.UsersRegistered, user)
    return m.RegisterFunc(user)
}

func TestRegisterUser(t *testing.T) {
    t.Run("can register valid users", func(t *testing.T) {
        user := User{Name: "CJ"}
        expectedInsertedID := "whatever"

        service := &MockUserService{
            RegisterFunc: func(user User) (string, error) {
                return expectedInsertedID, nil
            },
        }
        server := NewUserServer(service)

        req := httptest.NewRequest(http.MethodGet, "/", userToJSON(user))
        res := httptest.NewRecorder()

        server.RegisterUser(res, req)

        assertStatus(t, res.Code, http.StatusCreated)

        if res.Body.String() != expectedInsertedID {
            t.Errorf("expected body of %q but got %q", res.Body.String(), expectedInsertedID)
        }
    })
}
```

```
if len(service.UsersRegistered) != 1 {
    t.Fatalf("expected 1 user added but got %d", len(service.UsersRegistered))
}

if !reflect.DeepEqual(service.UsersRegistered[0], user) {
    t.Errorf("the user registered %+v was not what was expected %+v", service.UsersRegistered[0], user)
}

t.Run("returns 400 bad request if body is not valid user JSON", func(t *testing.T) {
    server := NewUserServer(nil)

    req := httptest.NewRequest(http.MethodGet, "/", strings.NewReader("trouble will find me"))
    res := httptest.NewRecorder()

    server.RegisterUser(res, req)

    assertStatus(t, res.Code, http.StatusBadRequest)
})

t.Run("returns a 500 internal server error if the service fails", func(t *testing.T) {
    user := User{Name: "CJ"}

    service := &MockUserService{
        RegisterFunc: func(user User) (string, error) {
            return "", errors.New("couldn't add new user")
        },
    }
    server := NewUserServer(service)

    req := httptest.NewRequest(http.MethodGet, "/", userToJSON(user))
    res := httptest.NewRecorder()

    server.RegisterUser(res, req)

    assertStatus(t, res.Code, http.StatusInternalServerError)
})
}
```

Now our handler isn't coupled to a specific implementation of storage
it is trivial for us to write a `MockUserService` to help us write simple,
fast unit tests to exercise the specific responsibilities it has.

What about the database code? You're cheating!

This is all very deliberate. We don't want HTTP handlers concerned with our business logic, databases, connections, etc.

By doing this we have liberated the handler from messy details, we've also made it easier to test our persistence layer and business logic as it is also no longer coupled to irrelevant HTTP details.

All we need to do is now implement our UserService using whatever database we want to use

```
type MongoUserService struct {
}

func NewMongoUserService() *MongoUserService {
    //todo: pass in DB URL as argument to this function
    //todo: connect to db, create a connection pool
    return &MongoUserService{}
}

func (m MongoUserService) Register(user User) (insertedID string, err error) {
    // use m.mongoConnection to perform queries
    panic("implement me")
}
```

We can test this separately and once we're happy in main we can snap these two units together for our working application.

```
func main() {
    mongoService := NewMongoUserService()
    server := NewUserServer(mongoService)
    http.ListenAndServe(":8000", http.HandlerFunc(server.RegisterUser))
}
```

A more robust and extensible design with little effort

These principles not only make our lives easier in the short-term they make the system easier to extend in the future.

It wouldn't be surprising that further iterations of this system we'd want to email the user a confirmation of registration.

With the old design we'd have to change the handler and the surrounding tests. This is often how parts of code become unmaintainable, more and more functionality creeps in because it's already designed that way; for the "HTTP handler" to handle... everything!

By separating concerns using an interface we don't have to edit the handler at all because it's not concerned with the business logic

around registration.

Wrapping up

Testing Go's HTTP handlers is not challenging, but designing good software can be!

People make the mistake of thinking HTTP handlers are special and throw out good software engineering practices when writing them which then makes testing them challenging.

Reiterating again; **Go's http handlers are just functions.** If you write them like you would other functions, with clear responsibilities, and a good separation of concerns you will have no trouble testing them, and your codebase will be healthier for it.

TDD Anti-patterns

From time to time it's necessary to review your TDD techniques and remind yourself of behaviours to avoid.

The TDD process is conceptually simple to follow, but as you do it you'll find it challenging your design skills. **Don't mistake this for TDD being hard, it's design that's hard!**

This chapter lists a number of TDD and testing anti-patterns, and how to remedy them.

Not doing TDD at all

Of course, it is possible to write great software without TDD but, a lot of problems I've seen with the design of code and the quality of tests would be very difficult to arrive at if a disciplined approach to TDD had been used.

One of the strengths of TDD is that it gives you a formal process to break down problems, understand what you're trying to achieve (red), get it done (green), then have a good think about how to make it right (blue/refactor).

Without this, the process is often ad-hoc and loose, which can make engineering more difficult than it could be.

Misunderstanding the constraints of the refactoring step

I have been in a number of workshops, mobbing or pairing sessions where someone has made a test pass and is in the refactoring stage. After some thought, they think it would be good to abstract away some code into a new struct; a budding pedant yells:

You're not allowed to do this! You should write a test for this first, we're doing TDD!

This seems to be a common misunderstanding. **You can do whatever you like to the code when the tests are green**, the only thing you're not allowed to do is **add or change behaviour**.

The point of these tests are to give you the freedom to refactor, find the right abstractions and make the code easier to change and understand.

Having tests that won't fail (or, evergreen tests)

It's astonishing how often this comes up. You start debugging or changing some tests and realise: there are no scenarios where this test can fail. Or at least, it won't fail in the way the test is supposed to be protecting against.

This is next to impossible with TDD if you're following **the first step**,

Write a test, see it fail

This is almost always done when developers write tests after code is written, and/or chasing test coverage rather than creating a useful test suite.

Useless assertions

Ever worked on a system, and you've broken a test, then you see this?

false was not equal to true

I know that false is not equal to true. This is not a helpful message; it doesn't tell me what I've broken. This is a symptom of not following the TDD process and not reading the failure error message.

Going back to the drawing board,

Write a test, see it fail (and don't be ashamed of the error message)

Asserting on irrelevant detail

An example of this is making an assertion on a complex object, when in practice all you care about in the test is the value of one of the fields.

```
// not this, now your test is tightly coupled to the whole object
if !cmp.Equal(complexObject, want) {
    t.Error("got %+v, want %+v", complexObject, want)
}

// be specific, and loosen the coupling
got := complexObject.fieldYouCareAboutForThisTest
if got != want {
    t.Error("got %q, want %q", got, want)
}
```

Additional assertions not only make your test more difficult to read by creating 'noise' in your documentation, but also needlessly couples the test with data it doesn't care about. This means if you happen to change the fields for your object, or the way they behave you may get unexpected compilation problems or failures with your tests.

This is an example of not following the red stage strictly enough.

- Letting an existing design influence how you write your test **rather than thinking of the desired behaviour**
- Not giving enough consideration to the failing test's error message

Lots of assertions within a single scenario for unit tests

Many assertions can make tests difficult to read and challenging to debug when they fail.

They often creep in gradually, especially if test setup is complicated because you're reluctant to replicate the same horrible setup to assert on something else. Instead of this you should fix the problems in your design which are making it difficult to assert on new things.

A helpful rule of thumb is to aim to make one assertion per test. In Go, take advantage of subtests to clearly delineate between assertions on the occasions where you need to. This is also a handy technique to separate assertions on behaviour vs implementation detail.

For other tests where setup or execution time may be a constraint (e.g an acceptance test driving a web browser), you need to weigh

up the pros and cons of slightly trickier to debug tests against test execution time.

Not listening to your tests

Dave Farley in his video "When TDD goes wrong" points out,

TDD gives you the fastest feedback possible on your design

From my own experience, a lot of developers are trying to practice TDD but frequently ignore the signals coming back to them from the TDD process. So they're still stuck with fragile, annoying systems, with a poor test suite.

Simply put, if testing your code is difficult, then using your code is difficult too. Treat your tests as the first user of your code and then you'll see if your code is pleasant to work with or not.

I've emphasised this a lot in the book, and I'll say it again **listen to your tests**.

Excessive setup, too many test doubles, etc.

Ever looked at a test with 20, 50, 100, 200 lines of setup code before anything interesting in the test happens? Do you then have to change the code and revisit the mess and wish you had a different career?

What are the signals here? Listen, complicated tests == complicated code. Why is your code complicated? Does it have to be?

- When you have lots of test doubles in your tests, that means the code you're testing has lots of dependencies - which means your design needs work.
- If your test is reliant on setting up various interactions with mocks, that means your code is making lots of interactions with its dependencies. Ask yourself whether these interactions could be simpler.

Leaky interfaces If you have declared an interface that has many methods, that points to a leaky abstraction. Think about how you could define that collaboration with a more consolidated set of methods, ideally one.

Interface pollution As a Go proverb says, the bigger the interface, the weaker the abstraction. If you expose a huge interface to the users of your package, you force them to create in their tests a stub/mock that matches the entire API, providing an implementation

also for methods they do not use (sometimes, they just panic to make clear that they should not be used). This situation is an anti-pattern known as [interface pollution](#) and this is the reason why the standard library offers you just tiny little interfaces.

Instead, you should expose from your package a bare struct with all relevant methods exported, leaving to the clients of your API the freedom to declare their own interfaces abstracting over the subset of the methods they need: e.g [go-redis](#) exposes a struct (`redis.Client`) to the API clients.

Generally speaking, you should expose an interface to the clients only when:

- the interface consists of a small and coherent set of functions.
- the interface and its implementation need to be decoupled (e.g. because users can choose among multiple implementations or they need to mock an external dependency).

Think about the types of test doubles you use

- Mocks are sometimes helpful, but they're extremely powerful and therefore easy to misuse. Try giving yourself the constraint of using stubs instead.
- Verifying implementation detail with spies is sometimes helpful, but try to avoid it. Remember your implementation detail is usually not important, and you don't want your tests coupled to them if possible. Look to couple your tests to **useful behaviour rather than incidental details**.
- [Read my posts on naming test doubles](#) if the taxonomy of test doubles is a little unclear

Consolidate dependencies Here is some code for a `http.HandlerFunc` to handle new user registrations for a website.

```
type User struct {
    // Some user fields
}

type UserStore interface {
    CheckEmailExists(email string) (bool, error)
    StoreUser(newUser User) error
}

type Emailer interface {
    SendEmail(to User, body string, subject string) error
}
```

```
func NewRegistrationHandler(userStore UserStore, emailer Emailer) http.HandlerFunc {
    return func(writer http.ResponseWriter, request *http.Request) {
        // extract out the user from the request body (handle error)
        // check user exists (handle duplicates, errors)
        // store user (handle errors)
        // compose and send confirmation email (handle error)
        // if we got this far, return 2xx response
    }
}
```

At first pass it's reasonable to say the design isn't so bad. It only has 2 dependencies!

Re-evaluate the design by considering the handler's responsibilities:

- Parse the request body into a User □
- Use UserStore to check if the user exists □
- Use UserStore to store the user □
- Compose an email □
- Use Emailer to send the email □
- Return an appropriate http response, depending on success, errors, etc □

To exercise this code, you're going to have to write many tests with varying degrees of test double setups, spies, etc

- What if the requirements expand? Translations for the emails? Sending an SMS confirmation too? Does it make sense to you that you have to change a HTTP handler to accommodate this change?
- Does it feel right that the important rule of "we should send an email" resides within a HTTP handler?
 - Why do you have to go through the ceremony of creating HTTP requests and reading responses to verify that rule?

Listen to your tests. Writing tests for this code in a TDD fashion should quickly make you feel uncomfortable (or at least, make the lazy developer in you be annoyed). If it feels painful, stop and think.

What if the design was like this instead?

```
type UserService interface {
    Register(newUser User) error
}
```

```
func NewRegistrationHandler(userService UserService) http.HandlerFunc {
    return func(writer http.ResponseWriter, request *http.Request) {
        // parse user
        // register user
    }
}
```

```
// check error, send response
}
}

• Simple to test the handler □
• Changes to the rules around registration are isolated away from
HTTP, so they are also simpler to test □
```

Violating encapsulation

Encapsulation is very important. There's a reason we don't make everything in a package exported (or public). We want coherent APIs with a small surface area to avoid tight coupling.

People will sometimes be tempted to make a function or method public in order to test something. By doing this you make your design worse and send confusing messages to maintainers and users of your code.

A result of this can be developers trying to debug a test and then eventually realising the function being tested is only called from tests. Which is obviously **a terrible outcome, and a waste of time**.

In Go, consider your default position for writing tests as from the perspective of a consumer of your package. You can make this a compile-time constraint by having your tests live in a test package e.g package gocoin_test. If you do this, you'll only have access to the exported members of the package so it won't be possible to couple yourself to implementation detail.

Complicated table tests

Table tests are a great way of exercising a number of different scenarios when the test setup is the same, and you only wish to vary the inputs.

But they can be messy to read and understand when you try to shoe-horn other kinds of tests under the name of having one, glorious table.

```
cases := []struct {
    X      int
    Y      int
    Z      int
    err    error
    IsFullMoon  bool
    IsLeapYear  bool
    AtWarWithEurasia bool
}{}
```

Don't be afraid to break out of your table and write new tests
rather than adding new fields and booleans to the table struct.

A thing to bear in mind when writing software is,

[Simple is not easy](#)

"Just" adding a field to a table might be easy, but it can make things far from simple.

Summary

Most problems with unit tests can normally be traced to:

- Developers not following the TDD process
- Poor design

So, learn about good software design!

The good news is TDD can help you improve your design skills because as stated in the beginning:

TDD's main purpose is to provide feedback on your design.
For the millionth time, listen to your tests, they are reflecting your design back at you.

Be honest about the quality of your tests by listening to the feedback they give you, and you'll become a better developer for it.