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 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

Define `Countdown`

```
1 func Countdown() {}
```

Try again

```
1 ./countdown_test.go:11:11: too many arguments in call to Countdown
2     have (*bytes.Buffer)
3     want ()
```

The compiler is telling you what your function signature could be, so update it.

```
1 func Countdown(out *bytes.Buffer) {}
```

`countdown_test.go:17: got '' want '3'`

Perfect!

Write enough code to make it pass

```
1 func Countdown(out *bytes.Buffer) {
2         fmt.Fprint(out, "3")
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 put 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!" aftward.

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-- {
                time.Sleep(1 * time.Second)
                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 4 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 4s 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 `Sleep`ing 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.

```
1 type Sleeper interface {
2         Sleep()
3 }
```

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 4 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 != 4 {
                t.Errorf("not enough calls to sleeper, want 4 got %d", spySleeper.Cal
        }
}
```

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-- {
                time.Sleep(1 * time.Second)
                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 in dependency. Let's fix that.

```
func Countdown(out io.Writer, sleeper Sleeper) {
        for i := countdownStart; i > 0; i-- {
                sleeper.Sleep()
                fmt.Fprintln(out, i)
        }

        sleeper.Sleep()
        fmt.Fprint(out, finalWord)
}
```

The test should pass and no longer take 4 seconds.

Still some problems

There's still another important property we haven't tested.

`Countdown` should sleep before each print, e.g:

- `Sleep`
- `Print N`
- `Sleep`
- `Print N-1`
- `Sleep`
- `Print Go!`

- etc

Our latest change only asserts that it has slept 4 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)
        }

        sleeper.Sleep()
        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{
                sleep,
                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 in 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{
                        sleep,
                        write,
                        sleep,
                        write,
                        sleep,
                        write,
                        sleep,
                        write,
                }

                if !reflect.DeepEqual(want, spySleepPrinter.Calls) {
                        t.Errorf("wanted calls %v got %v", want, spySleepPrinter.Call
                }
        })
}
```

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, spyT
        }
}
```

There should be nothing new in this test and it is setup very similar to the previous mock tests.

Try and run the test

```
sleeper.Sleep undefined (type ConfigurableSleeper has no field or method Sleep, but
```

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}
```

```
4 }       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
- practise 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. The "proper" term for mocks though are "test doubles"

> 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.