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

```go
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

```go
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

```go
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

```go
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

```go
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

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

Now let's refactor the tests to use `Rectangle` instead of plain `float64`s.

```go
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.

```go
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

```go
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.

```go
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:

```go
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.Errorf` we are calling the method `Errorf` 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.

```go
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 meth
./shapes_test.go:29:16: circle.Area undefined (type Circle has no field or method Are
```

> 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

```go
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.

```go
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

```go
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).

```go
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 `Rectangle`s and `Circle`s 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.

```go
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

```go
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.

```go
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.

```go
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
    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

```go
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

```go
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

```go
{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

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

	areaTests := []struct {
		name    string
		shape   Shape
		hasArea float64
	}{
		{name: "Rectangle", shape: Rectangle{Width: 12, Height: 6}, hasArea:
		{name: "Circle", shape: Circle{Radius: 10}, hasArea: 314.1592653589
		{name: "Triangle", shape: Triangle{Base: 12, Height: 6}, hasArea: 36.
	}

	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.has
			}
		})

	}

}
```

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.