Command line & package 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`.

```go
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:

```go
//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 godoc.org.

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

```go
//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`)

```go
//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")
```

```go
	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`.

```go
//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.winCall
	}
}
```

Now replace the assertions in both `server_test.go` and `CLI_test.go`.

The test should now read like so

```go
//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

```go
//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
}
```

```go
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)

```go
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 'playe
command-line/v3/cmd/cli/main.go:32:34: implicit assignment of unexported field 'in' i
```

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 `CLI`s) 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` because 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.

```go
//testing.go
package poker

import "testing"

```

```go
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.winCall
	}
}

// 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.

```go
//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.C
./CLI_test.go:15:39: implicit assignment of unexported field 'in' in poker.CLI litera
./CLI_test.go:25:26: implicit assignment of unexported field 'playerStore' in poker.C
./CLI_test.go:25:39: implicit assignment of unexported field 'in' in poker.CLI litera
```

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.

```go
//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(), erro
```

```go
	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 stor
	}

	return store, closeFunc, nil
}
```

Now refactor both of our applications to use this function to create the store.

CLI application code

```go
//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

```go
//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 `main`s.

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.