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.

```
1 type CLI struct {
2         playerStore PlayerStore
3         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.

                                if len(blindAlerter.alerts) <= i {
                                        t.Fatalf("alert %d was not scheduled %v", i,
                                }

                                alert := blindAlerter.alerts[i]

                                amountGot := alert.amount
                                if amountGot != c.expectedAmount {
                                        t.Errorf("got amount %d, want %d", amountGot,
                                }

                                gotScheduledTime := alert.scheduledAt
                                if gotScheduledTime != c.expectedScheduleTime {
                                        t.Errorf("got scheduled time of %v, want %v",
                                }
                        })
                }
        })
```

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
                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_10m
                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.Sprint(want), func(t *testing.T) {

                        if len(blindAlerter.alerts) <= i {
                                t.Fatalf("alert %d was not scheduled %v", i, blindAle
                        }

                        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 `BlindAlerter.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) *CL
```

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) *CL
        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.Sprint(want), func(t *testing.T) {

                        if len(blindAlerter.alerts) <= i {
                                t.Fatalf("alert %d was not scheduled %v", i, blindAle
                        }

                        got := blindAlerter.alerts[i]
                        assertScheduledAlert(t, got, want)
                })
        }
```

```
34 })
```

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

```
1 === RUN   TestCLI
2 --- FAIL: TestCLI (0.00s)
3 === RUN   TestCLI/it_prompts_the_user_to_enter_the_number_of_players
4     --- FAIL: TestCLI/it_prompts_the_user_to_enter_the_number_of_players (0.00s)
5 === RUN   TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chips_at_0s
6         --- PASS: TestCLI/it_prompts_the_user_to_enter_the_number_of_players/100_chip
7 === RUN   TestCLI/it_prompts_the_user_to_enter_the_number_of_players/200_chips_at_12m
```

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!

```
1 func (cli *CLI) PlayPoker() {
2         fmt.Fprint(cli.out, PlayerPrompt)
3 
4         numberOfPlayers, _ := strconv.Atoi(cli.readLine())
5 
```

```
        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 {
```

```
25          in  *bufio.Scanner
            out  io.Writer

26          game *Game
27  }
28
29  func NewCLI(store PlayerStore, in io.Reader, out io.Writer, alerter BlindAlerter) *CL
30          return &CLI{
31                  in:  bufio.NewScanner(in),
32                  out: out,
33                  game: &Game{
34                          alerter: alerter,
35                          store:   store,
36                  },
37          }
38  }
39
40  const PlayerPrompt = "Please enter the number of players: "
41
42  func (cli *CLI) PlayPoker() {
43          fmt.Fprint(cli.out, PlayerPrompt)
44
45          numberOfPlayersInput := cli.readLine()
46          numberOfPlayers, _ := strconv.Atoi(strings.Trim(numberOfPlayersInput, "\n"))
47
48          cli.game.Start(numberOfPlayers)
49
50          winnerInput := cli.readLine()
51          winner := extractWinner(winnerInput)
52
53          cli.game.Finish(winner)
54  }
55
56  func extractWinner(userInput string) string {
57          return strings.Replace(userInput, " wins\n", "", 1)
58  }
59
60  func (cli *CLI) readLine() string {
61          cli.in.Scan()
62          return cli.in.Text()
63  }
```

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`

```
1 func NewCLI(in io.Reader, out io.Writer, game *Game) *CLI {
2         return &CLI{
3                 in:   bufio.NewScanner(in),
4                 out:  out,
5                 game: game,
6         }
7 }
```

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* `Game`s 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
                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.Starte
                }
        })
```

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.

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
        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
    --- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does
        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
    --- FAIL: TestCLI/it_prints_an_error_when_a_non_numeric_value_is_entered_and_does
        CLI_test.go:70: got 'Please enter the number of players: ', want 'Please ente
```

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 ag
```

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, "")
        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", fur
                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 *testir
                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 st
                game := &GameSpy{}

                stdout := &bytes.Buffer{}
                in := userSends("pies")

                cli := poker.NewCLI(in, stdout, game)
                cli.PlayPoker()

                assertGameNotStarted(t, game)
```

```
40          })      assertMessagesSentToUser(t, stdout, poker.PlayerPrompt, poker.BadPlay
41  }
```

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.

```
1 type BlindAlerter interface {
2         ScheduleAlertAt(duration time.Duration, amount int)
3 }
4
```

```
// 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])
}
```