OS Exec

You can find all the code here

keith6014 asks on reddit

> I am executing a command using os/exec.Command() which generated XML data. The command will be executed in a function called GetData().

> In order to test GetData(), I have some testdata which I created.

> In my _test.go I have a TestGetData which calls GetData() but that will use os.exec, instead I would like for it to use my testdata.

> What is a good way to achieve this? When calling GetData should I have a "test" flag mode so it will read a file ie GetData(mode string)?

A few things

- When something is difficult to test, it's often due to the separation of concerns not being quite right
- Don't add "test modes" into your code, instead use Dependency Injection so that you can model your dependencies and separate concerns.

I have taken the liberty of guessing what the code might look like

```
1  type Payload struct {
2          Message string `xml:"message"`
3  }
4
5  func GetData() string {
6          cmd := exec.Command("cat", "msg.xml")
7
8          out, _ := cmd.StdoutPipe()
9          var payload Payload
10         decoder := xml.NewDecoder(out)
```

```
11
12          // these 3 can return errors but I'm ignoring for brevity
13          cmd.Start()
14          decoder.Decode(&payload)
15          cmd.Wait()
16
17          return strings.ToUpper(payload.Message)
18  }
```

- It uses `exec.Command` which allows you to execute an external command to the process
- We capture the output in `cmd.StdoutPipe` which returns us a `io.ReadCloser` (this will become important)
- The rest of the code is more or less copy and pasted from the excellent documentation.
    - We capture any output from stdout into an `io.ReadCloser` and then we `Start` the command and then wait for all the data to be read by calling `Wait`. In between those two calls we decode the data into our `Payload` struct.

Here is what is contained inside `msg.xml`

```
1 <payload>
2     <message>Happy New Year!</message>
3 </payload>
```

I wrote a simple test to show it in action

```
1 func TestGetData(t *testing.T) {
2          got := GetData()
3          want := "HAPPY NEW YEAR!"
4
5          if got != want {
6                  t.Errorf("got %q, want %q", got, want)
7          }
8 }
```

Testable code

Testable code is decoupled and single purpose. To me it feels like there are two main concerns for this code

1. Retrieving the raw XML data
2. Decoding the XML data and applying our business logic (in this case `strings.ToUpper` on the `<message>`)

The first part is just copying the example from the standard lib.

The second part is where we have our business logic and by looking at the code we can see where the "seam" in our logic starts; it's where we get our `io.ReadCloser`. We can use this existing abstraction to separate concerns and make our code testable.

The problem with GetData is the business logic is coupled with the means of getting the XML. To make our design better we need to decouple them

Our `TestGetData` can act as our integration test between our two concerns so we'll keep hold of that to make sure it keeps working.

Here is what the newly separated code looks like

```
type Payload struct {
        Message string `xml:"message"`
}

func GetData(data io.Reader) string {
        var payload Payload
        xml.NewDecoder(data).Decode(&payload)
        return strings.ToUpper(payload.Message)
}

func getXMLFromCommand() io.Reader {
        cmd := exec.Command("cat", "msg.xml")
        out, _ := cmd.StdoutPipe()

        cmd.Start()
        data, _ := ioutil.ReadAll(out)
        cmd.Wait()

        return bytes.NewReader(data)
}
```

```
22 func TestGetDataIntegration(t *testing.T) {

23         got := GetData(getXMLFromCommand())
24         want := "HAPPY NEW YEAR!"
25
26         if got != want {
27                 t.Errorf("got %q, want %q", got, want)
28         }
29 }
```

Now that `GetData` takes its input from just an `io.Reader` we have made it testable and it is no longer concerned how the data is retrieved; people can re-use the function with anything that returns an `io.Reader` (which is extremely common). For example we could start fetching the XML from a URL instead of the command line.

```
1  func TestGetData(t *testing.T) {
2          input := strings.NewReader(`
3  <payload>
4      <message>Cats are the best animal</message>
5  </payload>`)
6
7          got := GetData(input)
8          want := "CATS ARE THE BEST ANIMAL"
9
10         if got != want {
11                 t.Errorf("got %q, want %q", got, want)
12         }
13 }
```

Here is an example of a unit test for `GetData`.

By separating the concerns and using existing abstractions within Go testing our important business logic is a breeze.