SUMMER ‘18 INTERNSHIP

Malavika Pande

PROGNOS | 85 BROAD ST, NEW YORK, NY 10004

TABLE OF CONTENTS

Prior to 06/06/2018 2
Learning Go 2
Learning Git 4
06/06/18 5
06/11/18 6
Reviewing the Error Cases 6
06/12/18 7
First round of edits 7
06/12/18 8
Second round of edits 8
06/28/18 Onwards 9
07/02/18 10
The Panic Error 11
Code Coverage 12
Using Print Statements to Debug 14
07/11/18 Onwards: Changing Iteration Technique 15
Some more info on String Literals: 17
Unexpected Errors 17
Concatenating Valid Message: 18
Adjusted Framework 19
Changing the main Validate() error function 19
Creating Separate Branches 22
Conclusion 23

Prior to 06/06/2018

Learning Go

I learned the Go programming language by taking handwritten notes and completing the exercises listed on the interactive go tutorial. This also included reading about **Go doc, Go test, Go vet**, and reading articles listed on the Go Learning Trello card.

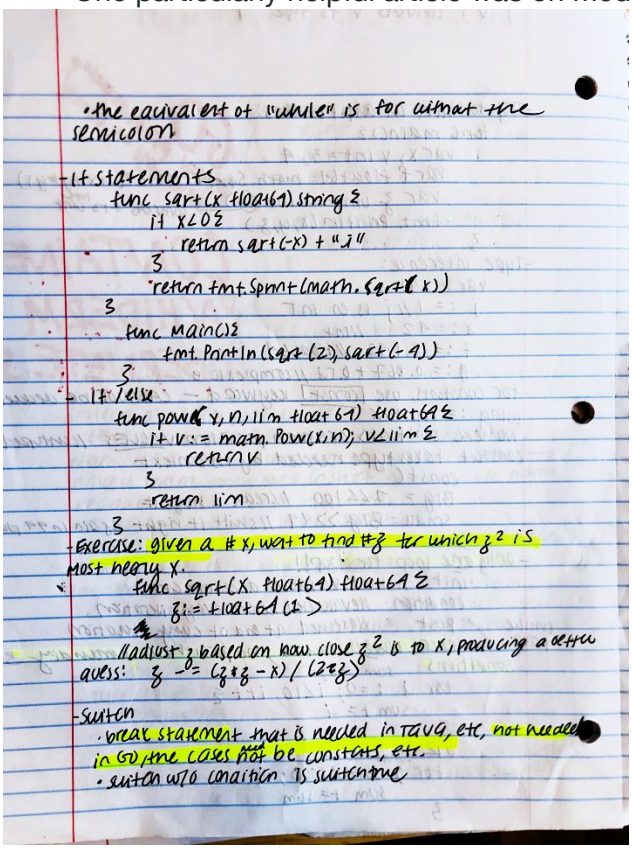
Go doc allows you to navigate from a function's documentation to its implementation with just one click. This convention allows you to generate documentation in various formats (plain text → HTML → UNIX).

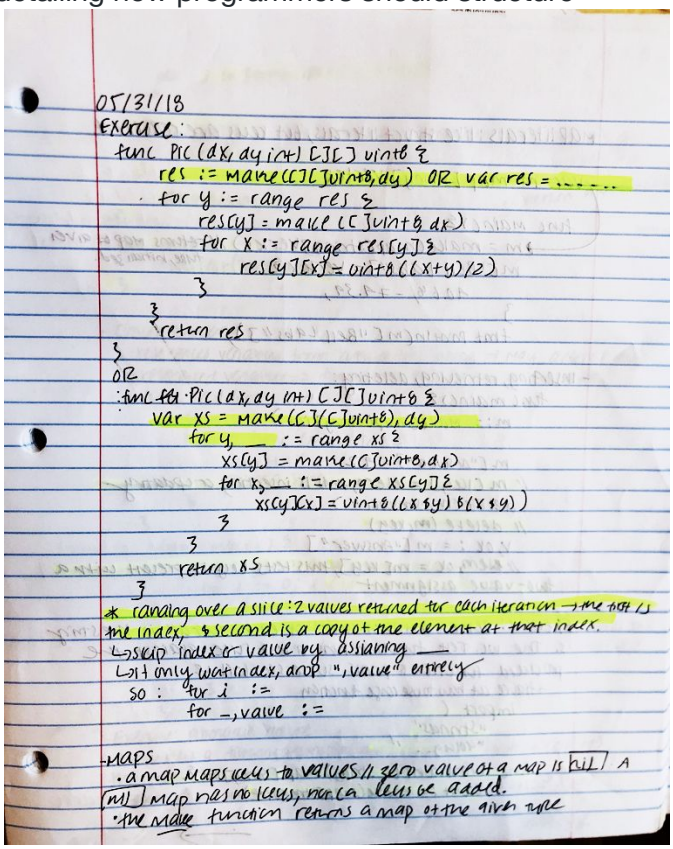
Go test (import "testing") is a package that runs and verifies example code, including a concluding line comment that begins with "output". The expected output is compared with the standard output of the function when run. You can also define subtests and sub-benchmarks and in turn enable hierarchal tests.

Go vet examines the source code and reports suspicious constructs. The exit codes are 2 – erroneous invocation, 1- if a problem is reported, and 0 otherwise.

Examples of basic, helpful Go exercises:

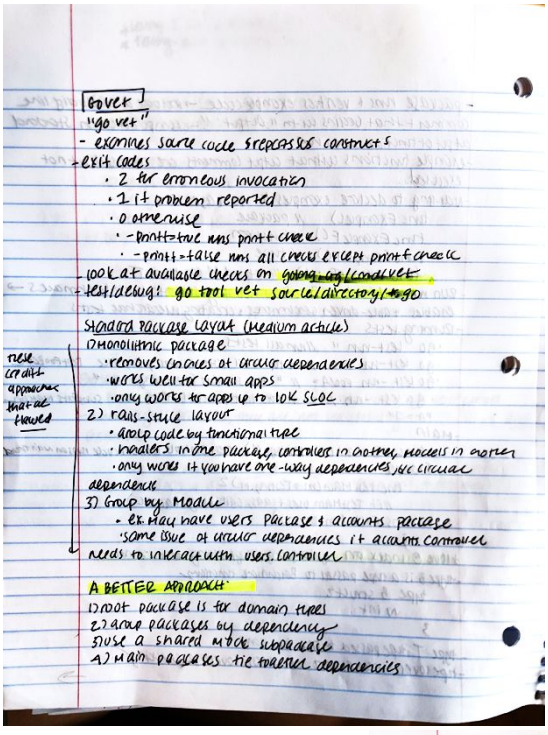
One particularly helpful article was on Medium, detailing how programmers should structure

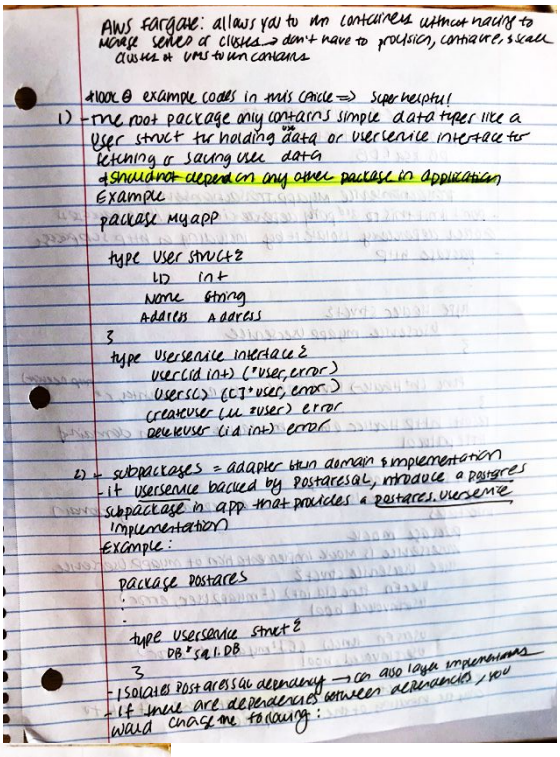


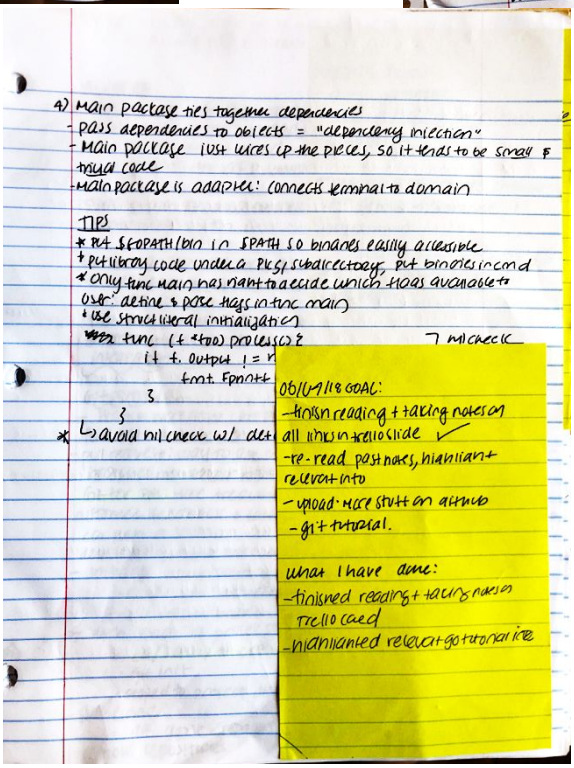


their applications. A better approach, according to the article, is to restrict the root package for

domain types, group packages by dependency, use a shared mock subpackage, and have main packages tie dependencies together.







Learning Git

I had little exposure to GitHub prior to my internship but Git was entirely new to me. I read and took notes on GitHub's tutorial as well and completed an interactive tutorial a Professor at Columbia created.

To get started on the underwriting project, I first had to clone the existing armadillo repository `git clone <repo url>` and create a new branch `git checkout -b validation_cmd2.` To add my changes I run the command `git add internal/roster,` to commit the changes I run `git -m "commit message"` and to push the changes onto GitHub, I run `git push origin validation_cmd2.`

Other important commands include:

- `git status` – notifies user whether a file is tracked or not
- `git config --global user.name "your full name"` – configures git environment
- `git grep` – searches specified patterns in all files in the repository
- `git fetch, git rebase origin/master` -updates changes made in master branch to local branch

06/06/18

I was provided information on what I had to do to create a validation framework. I downloaded visual studio code, installed the go extension, and attempted to install go metalinter but had troubles doing so.

The original structure of the Roster App was as follows: cmd (prepare new roster for evaluation) => output a JSON file => cmd (catalog members in the roster appending risk score) => output a JSON file => cmd. In UNIX, pipes redirected into standard inputs and standard output. So, the process I was working on was new roster | **validate** | catalog | evaluate > o.json. The validate command essentially checks if the roster is valid or not.

Initial brainstorming:

- How do I read the CSV file?
- How can I use the existing function to read the CSV file?
- How do I extract individual rowers?
- How do I read the headers?
- How do I iterate across the columns?
- Will I have to check for duplicates?
- Should I use a 2-D array?

06/11/18

Reviewing the Error Cases

I spoke to Denise earlier about the error cases and she was helpful on explaining each one individually. Some important things to note were that #11 required existing Prognos lab registry, this error case required looking at 2 cached files. #15 and #16 implies that there are “service hours” and customers want system up to 24 hours/day.

So, I completed the functions to check for all error cases except the ones I did not know as much information about.

06/12/18

First round of edits

Evan's corrections:

1) Add "valid" flag
2) Add validation messages to struct as well
3) Validations inserted into processing pipeline between load/catalog

The focus should be on ValidateMembers for now

The following day, the above edits and a draft of roster_validate_test.txt was made.

```
20  20  func (r *Roster) Validate() error {
21  21          rows := readCSV("AK1.psv")
22  22          r.Valid = true
23  23
24  24          //Validation No2
25  25          if !r.validateMembers(rows) {
26  26                  r.Valid = false
27      -           //append(r.ValidationMessage, "Risk Predictor Score cannot be provided because
28      -           //number of Eligible Member is less than the minimum of 10.")
    27  +           //append(r.ValidationMessage, Risk Predictor Score cannot be provided because
    28  +           //number of Eligible Member is less than the minimum of 10.)
29  29          }
30  30          return nil
31  31  }
    ❄   @@ -58,7 +58,6 @@ func readCSV(fileName string) [][]string {
58  58  // Fix this
59  59  func (r *Roster) validateMembers(rows [][]string) bool {
60  60          r.Valid = true
61      -       var tokenfour [][]string
62  61          for i := range rows {
63  62                  tokenfour := rows[i][8]
64  63                  //Checks if number of elements in T4 column < 10
```

**r.Valid added*

```
❄   @@ -24,8 +25,16 @@ func (r *Roster) Validate() error {
25          //Validation No2
26          if !r.validateMembers(rows) {
27                  r.Valid = false
        -           //append(r.ValidationMessage, Risk Predictor Score cannot be provided because
        -           //number of Eligible Member is less than the minimum of 10.)
28      +           r.ValidMsg = "Risk Predictor score cannot be provided because number of Eligible Members is less than the minimum of 10"
29      +           //append(r.ValidationMessage, "Risk Predictor Score cannot be provided because
30      +           //number of Eligible Member is less than the minimum of 10.")
31      +           return fmt.Errorf(r.ValidMsg)
32      +   }
33      +
34      +   if !r.checkID(rows) {
35      +           r.Valid = false
36      +           //figure out syntax for this
37      +           //append(r.ValidationMessage, )
38          }
39          return nil
40  }
```

**r.ValidMsg added*

06/12/18

Second round of edits

After rebasing the master branch, essentially updating the changes that were made, I continued making edits to the existing code. Additionally—OPAL level validation tests no longer had to implemented.

Evan's corrections:

1) readCSV() not needed → use Roster's Raw `PSVReader()`
2) Don't need to read in rows here → Roster r will always have method available to get and loop through rows efficiently so you can use it closer where it is needed
3) In individual functions, don't set r.Valid as it has a leaky side effect. Consider making the "helper functions" functions rather than a method on the struct. For these type of functions, just return true/false and do bookkeeping for r.Valid in Validate method.

For context, I had originally created my own `readCSV()` functions that would help read and extract the rows (of type [][] String)

```
func readCSV(fileName string) [][]string {

    csvFile, err := os.Open(fileName)

    if err != nil {
        log.Fatalf("Cannot open '%s' : %s\n", fileName, err.Error())
    }
    //Input stream separated into rows and columns
    r := csv.NewReader(csvFile)
    r.Comma = '|'
    r.LazyQuotes = true

    //Read all the rows at once
    rows, err := r.ReadAll()

    if err != nil {
        log.Fatalln("Cannot read data:", err.Error())
    }
    //defer csvFile.Close()
    return rows
}
```

06/28/18 Onwards

In previous commits, altered Roster File itself with a function that returns r.RawRows. This does not work as well because it is difficult to return r.RawRows as type [][]String. So, this version is more or less the original version that does the following:

```
func (r *Roster) Validate() error {
r.Valid = true
```

```
//readAll returns remaining records from r, each record is a slice of fields
//Use r.RawPSVReader to pull out rows from roster
reader := r.RawPSVReader()

rows, err := reader.ReadAll()

if err != nil {
        log.Fatal(err)
}

// S.No1
if !checkID(rows){
        r.Valid = false
        r.ValidMsg = "Roster ID Number should not be blank"
        return fmt.Errorf(r.ValidMsg)
}
```

The reader uses r.RawPSVReader() from roster_go, and using reader.ReadAll() returns the rows in the file. The rest of the functions are dependent on how the rows are extracted. They take in these rows and iterate across specific columns corresponding to each field (i.e RosterID, etc.).

Issues:

- Not passing go test -v -coverprofile c.out ./...
- Using r.RawRows would require possibly changing the overall structure of roster_validate // how r.RawRows would be implemented is in the comments of roster.go in this commit as well as previous commits

Now, I need to address why go test -v -coverprofile c.out ./... is not passing and what is needed to merge these commits into master. Most issues have been related to how the csv file is read and how the rows/columns within this file are identified. Creating the actual helper functions (checkID, checkIndustry, etc.)—that correspond to each S.No respectively—was relatively quick.

07/02/18

Git Practice

I picked up another Trello card, that was simple, and just required me to change score.json → output.json in job_upload.go. A simple task like this also helped me exercise some key git

skills. I first had to make sure my remote workspace was entirely updated, so I switched branches to master using `git checkout master` and then used `git pull` to update it. Running `git checkout -b rename` created a new branch where I would then change the code. Then, I ran `git add internal/job/job_upload` to add modified and new (unstaged) files to the staging area, and then finally ran `git commit -m "changed score → output"` to commit the changes. Finally, I ran `git push origin rename` and created a pull request so my commits could be merged into master.

Key things that I learned:

- Make sure to delete unnecessary comments in the file before you commit them
- Remember to manually create the pull request on github after `git push origin <branch name>`
- To update your own branch with changes made to the master branch, switch to the master branch using `git checkout master` and then run `git checkout master`

The Panic Error

The reason why the program was failing to build and compile was primarily because of *2 reasons*. One was that in my validate_test file, the literal string with the sample roster was not only incorrect because it already contained risk scores, it was taking into account spaces because of how I had indented the string literal. What seemed like a minor error was the cause of several, long debugging sessions that were to of no avail.

I instead copied and pasted a sample roster AK1.psv and formatted like so:

```
20      r := Roster{Raw:`Roster ID Number|Employer Group Name|SIC / NAICS Code|Employer State Code|Employer Zip
21    AK1 - 44|||AK|99701||||0a3a028b6762abc8bee3c38d59a8189322cf3a500042|F|1978|996|false||
22    AK1 - 44|||AK|99701||||0c41a9248064ffe3cb747b8c361b24c1a022631c0042|F|1964|996|true||
23    AK1 - 44|||AK|99701||||1e12adeb7b6b4df5151e6b33bac97ca47ad710910042|F|1963|995|false||
24    AK1 - 44|||AK|99701||||1f00be004b571a109a2230c4511de4a64d32b2d90042|F|1954|995|false||
25    AK1 - 44|||AK|99701||||2a1acbc268243bf5679f26752d19a2a6efcde14b0042XXX|M|1970|995|false||
26    AK1 - 44|||AK|99701||||32409beb6ae045b1bc55a61523b0cb97adbbebf80042XXX|M|1960|996|true||
27    AK1 - 44|||AK|99701||||4d21aa8e32b458e78fea9a93fda7e29007b712c20042XXX|F|1988|900|false||
28    AK1 - 44|||AK|99701||||5abfbc7bc5fdc4cb0b174cd3eb4b00cfb3372a9a0042XXX|M|1981|996|true||
29    AK1 - 44|||AK|99701||||686d991d7ecb77427dd581099d0672229363a8e20042XXX|F|1984|997|true||
30    AK1 - 44|||AK|99701||||889685e23c0116f68747126c13554b1c3c4dd7f30042|F|1958|995|false||
31    `}
```

Another reason why the program was failing to compile was because of the following line:

```
if err != nil {
        log.Fatal(err)
}
```

Returning log.Fatal(err) was what was causing the Panic. Instead, I had to return err. After fixing these two errors, the program was finally able to compile. When running `go test ./...` the, program finally compiled, and what occurred was the following:

```
Malavikas-MacBook-Pro-2:armadillo malavikapande$ go test ./...
ok      github.com/medivo/armadillo/cmd/armadillo       (cached)
ok      github.com/medivo/armadillo/cmd/armadillo_lambda        (cached)
ok      github.com/medivo/armadillo/internal/job        (cached)
ok      github.com/medivo/armadillo/internal/roster     0.063s
Malavikas-MacBook-Pro-2:armadillo malavikapande$ go test ./...
ok      github.com/medivo/armadillo/cmd/armadillo       (cached)
ok      github.com/medivo/armadillo/cmd/armadillo_lambda        (cached)
ok      github.com/medivo/armadillo/internal/job        (cached)
ok      github.com/medivo/armadillo/internal/roster     0.063s
Malavikas-MacBook-Pro-2:armadillo malavikapande$ go test ./...
# github.com/medivo/armadillo/internal/roster
internal/roster/roster_validate_test.go:49:19: undefined: expectedValidMsg
internal/roster/roster_validate_test.go:50:34: undefined: expectedValidMsg
ok      github.com/medivo/armadillo/cmd/armadillo       (cached)
ok      github.com/medivo/armadillo/cmd/armadillo_lambda        (cached)
ok      github.com/medivo/armadillo/internal/job        (cached)
FAIL    github.com/medivo/armadillo/internal/roster [build failed]
Malavikas-MacBook-Pro-2:armadillo malavikapande$ go test ./...
ok      github.com/medivo/armadillo/cmd/armadillo       (cached)
ok      github.com/medivo/armadillo/cmd/armadillo_lambda        (cached)
ok      github.com/medivo/armadillo/internal/job        (cached)
ok      github.com/medivo/armadillo/internal/roster     0.052s
Malavikas-MacBook-Pro-2:armadillo malavikapande$
```

*All four were ‘ok’

Code Coverage

Now that the code was finally able to compile, the next steps were to:

1) Use the test coverage tools to identify other test cases you might be able to create and create them.

2) Push everything up to remote

3) Clean up the code, remove any needed comments, etc.

4) Use git rebase to squash the all the commits into a single commit

5) Open a pull request for review.

Code coverage is a technique for understanding how good your tests are. It isn't' the end all be all of good testing but is a decent indicator.

When running `go test -cover ./...`

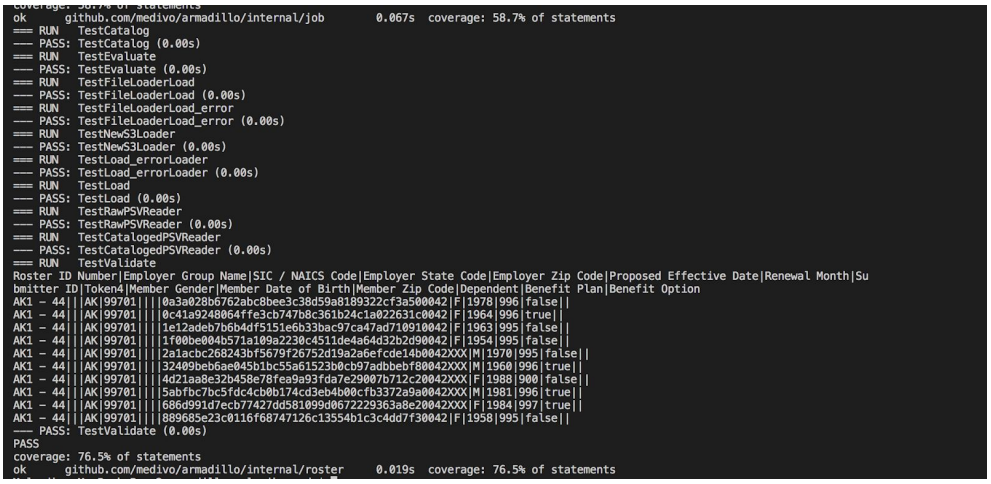
The program outputs the following:

```
ok      github.com/medivo/armadillo/cmd/armadillo       (cached)        coverage: 60.0% of statements
ok      github.com/medivo/armadillo/cmd/armadillo_lambda        (cached)        coverage: 10.3% of statements
ok      github.com/medivo/armadillo/internal/job        (cached)        coverage: 58.7% of statements
ok      github.com/medivo/armadillo/internal/roster     (cached)        coverage: 76.5% of statements
```

To get a more detailed and visual code coverage report Evan suggested running `go test -v -coverprofile c.out ./...` and `go tool cover -html=c.out.` Then, I can select roster_validate.go in the html viewer and see what test coverage I have.

When running `go test -v -coverprofile c.out ./...`

The program outputs the following:



76.5% of statements are run

When running `go tool cover -html=c.out`

The program outputs the following:

file:///var/folders/89/gt_2szt15yx2bgw2b0j_zj780000gn/T/cover275409030/coverage.html#file10

I need to create a test file that tests a roster file *without* a roster ID. This will help me see if the errors are outputting. To do this, I need to use *subtests*. On Friday I met with Evan and that

was extremely helpful in understanding why I was returning the same error no matter how much I changed my code, and this was because of an error in my tester file. And then I checked the coverage and it was about 76%, so now I have to identify all the edge cases and create tests that address them.

Questions:

For a “flawed roster”, would I get rid of of the pipe as well?

Correct Roster:

AK1 - 44|||AK|99701||||0a3a028b6762abc8bee3c38d59a8189322cf3a500042|F|1978|996|false||

Incorrect Roster (just removing the ID, not the pipe):

|||AK|99701||||0a3a028b6762abc8bee3c38d59a8189322cf3a500042|F|1978|996|false||

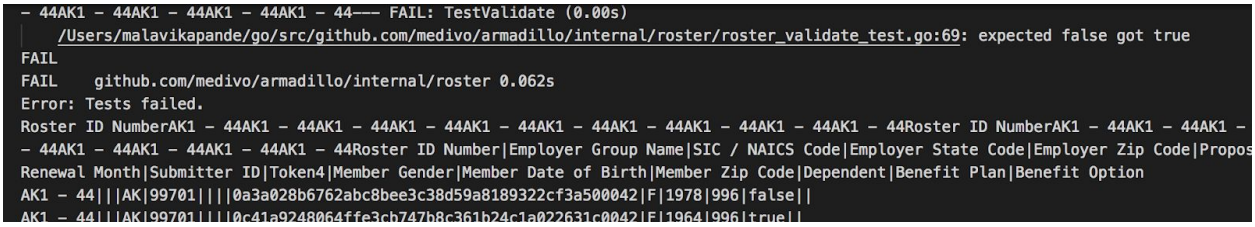
I realized that I actually should not remove the pipe, because then a “wrong number of fields” error will occur

When the test is run for the roster with the missing Roster ID, it is returning true when it should be returning false.

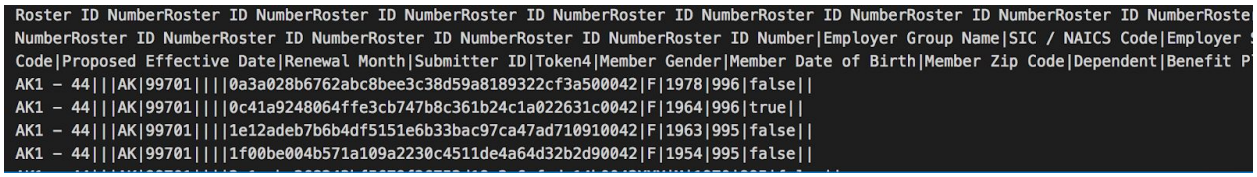
07/10: I created tests for more of the error cases that I use in my validation process, checking if the expectedValid boolean value is correct. Now I need to make sure that the expectedValidMsg is correct and I’m going to restructure the framework and use subtests now that I have a better understanding of how they work. I also need to make sure I’m inputting an incorrect roster correctly because initially my tests weren’t passing.

Using Print Statements to Debug

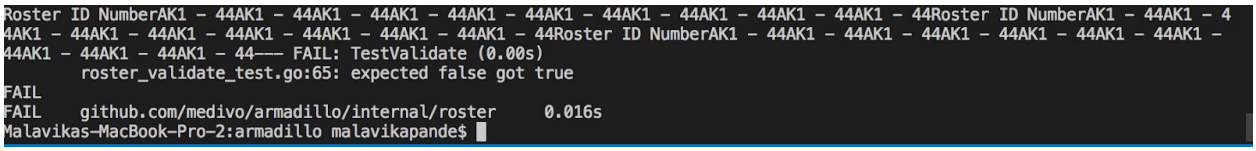
To figure out if `func checkID()` is working correctly, I decided to print rosterID. This would show me whether my way of extracting the rosterID column was correct.



Luckily, my way of extracting the column worked, but I realized that the function was only printing the first row in the rosterID column. I tried adding i++, but this didn't work. But now, my program is only printing "Roster ID…" etc repetitively. When I put the print statement after the second for loop, it prints "Roster ID" about 14 times, the same number of remaining headers in the row. This means I have to change the range.



It's printing "821111151161011143282111111511610111" when I include a print statement "fmt.Print(element)" after the line: `for_, element := range rosterID{.` This makes sense, because I did not print string(element). When I print string(element) instead, nothing prints out. Stuck, I found https://www.tutorialspoint.com/go/go_multi_dimensional_arrays.htm to be helpful. I realized my first for loop was actually incorrect, so I changed it to `for _ row := range rows{.` Printing the rosterID with `fmt.Print(row[0])` prints the Roster IDs, but it keeps looping over. Now it's printing all the rows in the column, which is a good start.



07/11/18 Onwards: Changing Iteration Technique

Adding `for _ row := range rows{,` allows me to iterate throughout *all* the rows in a particular column. Now, I can check an error case for each row in a columns.

07/11: I spent the majority of yesterday debugging after finding a problem in my code when I ran several tests, checking to see if I was getting the expected output. Luckily, I believe I fixed the error but now I have to make that same change on every function I've written.

07/12: I touched base with Evan after stand-up, and he agreed that the new for loop was a good change. However, there has to be a rosterIDIndex, and a function to get the rosterIdIndex in the roster.go file. This is because within the csv file, *we do not actually know beforehand what column the rosterID, or any field, will be in*. He suggested to do the following:

```
func checkID(rows [][]string) bool {

    for _, row := range rows {

        if isEmpty(row[rosterIdIndex]) {

          return false

       }

    }

    return true

}
```

The isEmpty is more of pseudocode at this point, but it is relevant because checking to see if an element = "" is ineffective. We want to check if the field is empty, not if it contains a space. This is because the ID itself could very well contain a space.

The function `GetRawIndexFor()` looks like the following:

```go
// GetRawIndexFor returns the index into the raw PSV for the given column header name
func (r *Roster) GetRawIndexFor(name string) (int, error) {
    psv, err := r.RawPSVReader().ReadAll()
    if err != nil {
        return 0, err
    }

    if len(psv) == 0 {
        return 0, fmt.Errorf("no roster loaded when getting index of '%v'", name)
    }

    for i, field := range psv[headerIndex] {
        if field == name {
            return i, nil
        }
    }

    return 0, fmt.Errorf("no index for name '%v' found", name)
}
```

07/13: touched base with Evan yesterday and he wrote a helper function that we thought would be necessary in figuring out which columns correspond to what field in the file. I changed the implementation of all my functions yesterday and now I'm just writing more tests and thinking of more possible edge cases.

I had to make sure my individual helper functions had a pointer to Roster so I could use `r.GetRawIndexFor`. I was also struggling with an error that required me to declare a field's index, in this case the rosterID's, as `rosterIDIndex, err := r.GetRawIndexFor("Roster ID Number")`. Then, I also had to deal with the fact that `err` was not being used, so I just wrote: `_ = err.` *This isn't optimal*, but it was the only way to get my program to compile.

When I inputted a flawed roster, one with a missing ID, it *correctly outputted false*. I checked for an empty field by first eliminating white space using strings.Replace (I should have thought of this before because I've used regex and string functions similar to this numerous item) and then using the length function, checking to see if it is equal to zero. Even though everything is printing correctly, my test is still failing so I think this either means I've written the test file wrong or I'm misinterpreting how it works.

One thing I've realized is that, there needs to be a way to output more than one ValidMsg.

07/16: After testing more cases and all my functions, I realized I needed to find a way to deal with a roster that would have multiple errors, and not just output an error for the first one it catches. This was why one of my tests wasn't passing. That aside my initial and most basic tests are all passing.

07/17: Yesterday I spent a good portion changing the framework of how my tests cool so I could isolate each test better. I also realized some tests weren't passing because of the way I had structured them. Hoping to wrap this up soon.

07/18: My update is similar to yesterday's, and I'm continuing to work with subtests and making sure all my tests are passing. I also updated my running Google Doc.

****

Some more info on String Literals:

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals. A raw string literal would be something like `` `foo` ``; its value includes the uninterpreted characters between the quotes and backslashes are disregarded. An example of an interpreted string literal is "bar", in which any character may appear except newline and an unescaped double quote.

Unexpected Errors

After looking a lot into sub-testing and changing the structure of my roster_validate_test.go, I reverted back to my original framework in order to correctly identify and deal with the error.

```
--- FAIL: TestValidate (0.00s)
        roster_validate_test.go:39: unexpected err: SIC / NAICS should not be blank.
        roster_validate_test.go:67: unexpected err: Roster ID Number should not be blank.
FAIL
```

What is difficult for me to process is that, the unexpected err is printing exactly what I want it to print, but as an “err”. This might be because of how I’m returning fmt.Errorf(r.ValidMsg) in the actual roster file. (In addition to this issue, I had a bug in my laptop so I had to get it replaced, and there were some issues transferring files when I used migration assistant). When I was stuck with this error, I spent time reviewing more about different Go packages and read several example codes concerning subtests and error codes. I even completed some practice exercises so I could think more clearly.

07/19: I’ve spent a lot of time dealing with an unexpected error and that’s the only issue that I’m having currently. I reverted back to a more simplistic form of testing and used a lot of print statements to figure out why the unexpected error was printing, and that’s currently the only issue I’m having. Everything else is outputting the expecting values it should. I’m hoping to push everything up today. I’m just having technical difficulties pushing what I have to github because I had to transfer information onto a different computer (because the one I was using wasn’t working).

07/23: I’m still dealing with an unexpected error and I had to create a new go environment workspace. I’ve just spent the past few days cleaning up my program and I still need to clean up my test file a bit more. Because I’m working on a new laptop and had issues (there was no administrator account on my computer so I had to reboot my entire system), I spent a lot of time working with terminal so I could provide my directory permission to make appropriate changes and test my program. In spite of the setback, I was able to learn and manipulate commands I had never known before. Now that this was fixed, I finally concatenated my strings effectively using bytes.Buffer. This seemed like Go’s version of Java’s Stringbuilder.

07/24: Yesterday I was finally able to give the directory I was working on permission to make changes and test my program, once that was fixed I just added a way to concatenate my strings effectively using Buffer.

Concatenating Valid Message:

I created a variable “buffer” that is able to concatenate the error message strings and returns all the error messages—this ensures that the program isn’t only outputting the first, single, error it catches. In my first test file I have different test functions that test different rosters. The expected variables are all outputting correctly. What is bizarre is that ValidMsg is viewed as a non-nil err, and that is what is returned when you check for unexpected errors.

I also realized that a lot of errors build on each other—for example, you can’t check if an ID is unique if it does not exist. This is why in those helper functions I first call the function that checks if the field exists or not.

I spent a lot of time re-doing how I handled test-cases after speaking with Evan, which was super helpful— and also created more helper functions to handle edge cases. I also incorporated the setUniqueFields function so I wouldn’t have to write as many lines of code, and in turn make things more efficient. I had time so once I got the validate Members function working (I need to push it up), I also structured all my functions like that. I worked on changing my other functions so they would work the way that the `ValidateMemberCount()` would work and also adjusted and changed my entire framework so it could incorporate other methods/functions in other branches.

Adjusted and Finalized Framework

Main Validate() error function

Working with functions with multiple return types is common in Go, especially because of the tedious and meticulous way it deals with errors.

Look at the example below:

```
func getStockPriceChangeWithError(prevPrice, currentPrice float64) (float64, float64, error)
    if prevPrice == 0 {
        err := errors.New("Previous price cannot be zero")
        return 0, 0, err
    }
    change := currentPrice - prevPrice
    percentChange := (change / prevPrice) * 100
    return change, percentChange, nil
}
```

```
func main() {
    prevStockPrice := 0.0
    currentStockPrice := 100000.0

    change, percentChange, err := getStockPriceChangeWithError(prevStockPrice, currentStockP

    if err != nil {
        fmt.Println("Sorry! There was an error: ", err)
    } else {
        if change < 0 {
            fmt.Printf("The Stock Price decreased by $%.2f which is %.2f%% of the prev price
        } else {
            fmt.Printf("The Stock Price increased by $%.2f which is %.2f%% of the prev price
        }
    }
}
```

The function `func getStockPriceChangeWithError(prevPrice, currentPrice float64) (float64, float 64, error)` returns 3 types, including an error type (see the last line of the function, `return change, percentChange, nil`).The main function calls `getStockPriceChangeWithError(prevPrice, currentPrice float64)` in third line: `change, percentChange, err := getStockPriceChangeWithError(prevStockPrice, currentStockPrice).` The program won't compile unless the function is called this way because it returns multiple values. In a similar manner, my "main" function, the overarching `Validate() error` must take into account that all the "helper" validation functions, of type

Roster, return a boolean and an error. My solution to this was a series of if statements that essentially checked if the individual validation functions were false, and returned nil. The “ok” signifies whether the function returns true or false.

```
if ok, err := r.validateMemberCount(); err != nil {

        return err

    } else if !ok {

        r.Valid = false

        r.ValidationMessages = append(r.ValidationMessages, "Risk Predictor score
cannot be provided because number of Eligible Members is less than the minimum")

    }
```

Individual validation functions

The validation functions themselves were also restructured. Essentially, there were *two* things I had to check for in almost every field: whether the field existed, and whether the field was unique. Checking whether a field exists requires using the `len()` function, which, as the name implies, checks the length of the string. Taking account of whitespace, I also had to trim it using `strings.Replace()`. The following implementation is used:

Here is an example of validateIndustry(), which checks whether an industry code exists:

```
func (r *Roster) validateIndustry() (bool, error) {

    rows, err := r.RawPSV()

    if err != nil {

        return false, err

    }

    industryIndex, err := r.GetRawIndexFor("SIC / NAICS Code")

    if err != nil {

        return false, err

    }

    for _, row := range rows {
```

```
        trimmedVal := strings.Replace(row[industryIndex], " ", "", -1)

        if len(trimmedVal) == 0 {

            return false, nil

        }

    }

    return true, nil

}
```

Checking to see whether a field is unique requires using a *map*—a data structure I've had much experience with in my classes at Columbia. Hash table implementations are important because they allow fast look-ups, adds, and deletes. Map types are reference types, and the make function is what initializes the hash map data structure. Maps are thus instrumental in detecting duplicate elements. Because the error use case specifies that a field must I've initialized a variable called `const uniqueField`, where the Field is either the State code, zip code, etc. `const uniqueField` is equal to the number of unique eligible members there are. Currently, I've set the number to one: this ensures there is only *one* unique field. The following implementation is used:

Here is an example of validateIndustryUnique(), which checks whether an industry code is unique:

```
func (r *Roster) validateIndustryUnique() (bool,error) {

    rows, err := r.RawPSV()

    if err != nil {

        return false, err

    }

    industryIndex, err := r.GetRawIndexFor("SIC / NAICS Code")

    if err != nil {

        return false, err

    }

    const uniqueIndustry = 1

    industryCodes := make(map[string]struct{})

    for _, row := range rows[1:]{

        industryCodes[row[industryIndex]] = struct{}{}

    }

    if len(industryCodes) > uniqueIndustry{
```

```
        return false, nil
    }
    return true, nil
}
```

Integration Tests vs. Unit Tests

I realized that the way I had been testing my validations had been wrong. The reason I was initially receiving an unexpected error was because I was trying to test too many things at the same time. I was, oddly enough, trying to combine integration and unit tests, or something along those lines. A *unit test* is a software testing method that tests individual units of source code. In the case of the roster application, a unit test would solely test individual validation functions with test cases. An *integration test* essentially combines all individual software modules and tests them as a group. This always occurs *after* unit testing. An integration test would test the overarching `Validate() error` function which calls on each and every validation function.

Here is a partial example of a unit test for validateMemberCount(), which checks whether the number of eligible members is less than 10.

```
func TestValidateMemberCount(t *testing.T) {
    cases := []struct {
        rawPSV   string
        expected bool
    }{
        {"Token4|A|B\nt1|X|Y\nt2|X|Y\n", false},
        {"Token4\n1\n2\n3\n4\n5\n6\n7\n8\n9\n0", true},
        {"Token4\n1\n2\n3\n4\n5\n6\n7\n8\n9\n0\nA", true},
        {"Token4\n1\n2\n3\n4\n5\n6\n7\n8\n1\n1\n1", false},
    }

    for _, c := range cases {
        r := Roster{Raw: c.rawPSV}

        actual, err := r.validateMemberCount()
        assert.NoError(t, err)
        assert.Equal(t, c.expected, actual, c.rawPSV)
    }
}
```

This particular style of testing is known as *subtesting*, which allows you to test *multiple* cases in shorter lines of code.

Here is a partial example of an integration test:

```
func TestValidate(t *testing.T) {
    cases := []struct {
        rawPSVFile       string
        expectedValid    bool
        expectedMessages []string
    }{
        {"testdata/valid_roster.psv", true, []string{}},
        {"testdata/invalid_roster_validateMemberCount.psv", false, []string{"Risk Predictor
score cannot be provided because number of Eligible Members is less than the minimum"}},
        {"testdata/invalid_roster_validateRoleIndicator.psv", false, []string{"Eligible Member
Role Indicator should not be blank"}},
        {"testdata/invalid_roster_validateIndustryUnique.psv", false, []string{"We cannot
provide a Risk Predictor Score at this time because it appears that there is more than one
SIC/NAICS in the file"}},
    }

    for _, c := range cases {
        r := Roster{}
        r.Load(NewFileLoader(c.rawPSVFile))

        err := r.Validate()
        assert.NoError(t, err)
        assert.Equal(t, c.expectedValid, r.Valid)
        assert.Equal(t, c.expectedMessages, r.ValidationMessages)
    }
}
```

As you can see, this test ensures that the program is outputting the expected validMsg string and expected valid boolean value. The invalid rawPSVFile are different for each validation. This will be further explained in the next section.

Creating Separate Branches

Evan suggested I create branches corresponding to each validation. This isolates the validation functions so they can be tested individually and be merged into master. This required me to refer to my existing pull requests and commits, which contain the validation functions themselves as well as the test cases. I first had to switch to the master branch, then create separate branches using `git checkout -b <name of branch>.` The master branch already had `validateMemberCount()` and `validateRoleIndicator()` in place, so each additional branch would contain a validation method in addition to those. In order to isolate each test, for each branch I created, I had to create invalid test rosters. For example, the invalid roster csv file for `validateIndustryCode()` has missing industry codes, but *everything else is valid*. Though somewhat simple, creating the separate branches was somewhat tedious and required meticulous attention, especially when creating the test rosters.

****

07/30: I added more tests for each individual function and am in the process of creating different branches and reaching 100 % code coverage for each. I've just pushed up the first one.

07/31: I finished creating and pushing up each validation in a separate branch, adding tests to get the code coverage close to 100 percent, and creating fake invalid rosters for each. Now all of that work needs to reviewed.

Refactoring Validate() Error

Instead of a series of if else statements, there is a shorter way to call all the individual validation functions. One option is to create a slice of functions, loop through each one and check if valid or not, adding the appropriate error message where necessary. The other option if to create an array of function result structs such as { Valid string, Error error, Message string }. I played around with the first option, but found difficulty implementing it because all functions are bound to a Roster type r. So, what would result would be a slice of *methods*. I found the following example:

Function Collections

Of course, we can use functions in all the same places that we can use regular data types. We can, for example, create a slice of functions, chose a function at random, and then execute it. We define the type "binFunc", which is a binary function; it takes two integers, and returns one integer. This isn't strictly necessary for this example, but typing `binFunc` over and over again is much more convenient than typing `func(int, int) int` everywhere you see it.

```
type binFunc func(int, int) int

func main() {
    // seed your random number generator.
    rand.Seed(time.Now().Unix())

    // create a slice of functions
    fns := []binFunc{
        func(x, y int) int { return x + y },
        func(x, y int) int { return x - y },
        func(x, y int) int { return x * y },
        func(x, y int) int { return x / y },
        func(x, y int) int { return x % y },
    }

    // pick one of those functions at random
    fn := fns[rand.Intn(len(fns))]

    // and execute it
    x, y := 12, 5
    fmt.Println(fn(x, y))
}
```

I attempted to re-create the `Validate() error` framework like so, but my work was to of no avail because I completely disregarded that all functions have pointers to Roster (`func *r Roster).` I believe that in order to work around this you have to define some sort of interface that will return the Roster type, but I am not 100 percent sure. This is something I will definitely think more about in the future.

Conclusion

I can confidently say that the past few months have taught me more about the practical use of computer science and programming than my courses at Columbia have. Computer Science in the academic and professional world are indeed different: in academia rarely are you working with others collaboratively on a project, whereas professionally you almost always work in teams, and have to build off of someone else's work. That being said, Evan and the Sponges team have really instilled in me the idea that code must be *readable*. The code you write should be so clear that the code itself is documentation. You want the code to be maintainable. You want to write it so somebody else can pick up where you have left off. In school we are taught to abbreviate and shorten variables; And, sometimes the most complex frameworks and method are falsely glamorized but this can be somewhat detrimental when you are working with other people because they may not know what exactly the variable you are writing symbolizes, or what the function even does.. When I paired up with Evan he suggested I change my `validRoleIndic()` function to `validRoleIndicator()` for this reason.

Another fundamental rule of guidance—one that CS classes also emphasize—is YAGNI: you ain't gonna need it! The less code you write, the better. You want to write code in a way so that a portion of the code you are writing can easily be taken out. This was especially important throughout the Underwriting Project. For instance, if a PM decides that the member gender must be indicated by 'female' and 'male' instead of 'f' and 'm', making this change should be fairly easy.

Apart from programming skills, my internship overall has ingrained in me a refreshing and fierce sense of independence: the ability to work, to write, and to debug, without someone guiding me each and every step of the way . At school when I was faced with a bug I would often turn to TA's, friends, and professors to help me. On a team everyone is working towards the same goal and has to meet certain—and to be honest, difficult—deadlines, so you have to learn to deal with certain complexities and bugs on your own. My best friends in these past months were print statements and breakpoints, which helped me locate where an error was and why it was being caused. Go and Github were additionally both new to me, so I frequently took notes and completed tutorials to familiarize myself with both. Especially in the beginning, I found myself taking notes at every stand-up and every meeting to understand what each member of the team was working on and the overall goals of the product.

Although at many times I was frustrated with errors that I did not know how to handle, and doubted my abilities as a programmer and a member of a team, I felt motivated after every stand-up to continue to push through. Hearing the amount of work, planning, and coordination that goes into creating something remarkable like the Roster application made me grateful to even be apart of it. In fact, whenever I paired up with Evan, the feedback I received was extremely specific, detailed, yet simple. His way of explaining things, his patience, and the way he is so easily able to work with every member on the team are truly exemplary qualities. I can say the same about the rest of the Sponge team and the Product Managers I have worked with. One thing I have noticed is the *flexibility and adaptability* required in undertaking something as complex as the Underwriting Project. One of the most valuable things I have

learned this Summer is working with different people and different teams—whose ideas may conflict with your own— to achieve the same goal.