Lab 4

In this lab we will build a simulator for the Monty Hall problem. Our emphasis is on three aspects:

- Practicing test-driven development and working with GitHub issues.
- Getting more practice working with ES6 modules and classes.
- Learning how to manage a GUI component while also allowing the majority of the application to be automatically testable.

The Monty-Hall problem

The Monty-Hall problem is as follows:

- You are a contestant at a show, and you are presented with 3 closed doors, you must choose one door.
- You are told that one door has a luxury car behind it, while other two have nothing (but they show you a goat if you open them, as an indication that you lost).
- You get to choose one of the doors. Then the show host opens up one of the remaining doors that contain a goat.
- You are now given the option to STAY with your current door selection, or to SWITCH to the other closed door. You then get to see whether you won or lost.

It can be mathematically proven that it is to your advantage to switch. What we will do in this assignment is create a simulator that will help us see this fact. It will work as follows:

- The user will be presented with three closed doors, then they will select one door, and have another goat-door revealed.
- They will then have the option of switching or staying. Upon their choice, the winning door is revealed and their "scorecard" is updated with the appropriate information.

Updating your project

- Start up GitKraken.
- Make sure that you have committed all files related to your project. When you look at your project in GitKraken, you should be seeing no WIP section and no modified files.
- Right-click at the "upstream" item in the "REMOTE" section on the left side. Choose "Fetch upstream". You should now be seeing that your graph in the middle has two "master branches". One corresponds to "my repository" (the upstream) and the other corresponds to your master branch with the changes

you've made on it for Labs 1 and 2. This should be the "active" branch, with a little checkbox next to it. These two branches should appear to "deviate" from a common start.
- Right-click *my* master branch (upstream). You can do so either in the main window or in the upstream->master section in REMOTE on the left. Choose "Merge upstream/master onto master". You should see the two branches merge into a new "merge" commit.
- There is a good chance you will get "merge conflicts". Ask for help if you are not sure how to handle them.
- You are now ready to work on Lab 3! After you add your changes, save and create a commit in GitKraken, then push your changes.
- You will need to repeat these steps when a new lab is released.

Assignment

In your project you will find a Lab4 folder. It has some startup files, but you will also need to add your own files as you make progress.

Running a local server

In order to test the application, you will need to open the index.html file *as if it was a file served from a server*. In order to achieve that, you will need to be running a local server.

If you are on the lab computers, then open up a terminal window and navigate to the Lab3 folder of your project. In that folder, execute "http-server" from the terminal. You should see a response which will look something like this:

```
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://10.83.0.52:8080
Hit CTRL-C to stop the server
```

Keeping that terminal window open, open up your web-browser and in the location address write:

```
http://127.0.0.1:8080/index.html
```

where the numbers should be whatever your system is reporting.

If you want to do this on your own machines, you need to either know a way to run a local server, or you need to install the http-server package from NPM. The instructions to do that would likely be:

```
npm install -g http-server
```

You may need to install Node and NPM if you don't have those set up yet, and you may also need to run the above command as administrator.

You can stop the server at any time by hitting Ctrl-C on the terminal.

Issues, Milestones, Labels

We will start by setting up GitHub to help us track our progress.

- We will create a **Milestone** called "Lab 4" to keep track of our progress. It will host all the issues related to our project.
- We will create a couple custom **Labels** to help us organize our issues. Most of our issues will be assigned at least one label. This is just so that you familiarize yourself with the use of labels in relation to issues.
- We will then create some issues to describe our main design steps and key big ideas. As we work towards those issues, we will link the commits we make to the issues, so that the set of changes that solve a particular issue is easily identified.

Milestones vs Labels: An issue can have many labels associated to it, but it can only belong to one milestone.

Let's get started!

Creating a milestone

- Go to your project's GitHub page, and to the Issues section.
- Click the Milestones button, and choose "New Milestone" to create a new milestone.
- Give your milestone a title; I named mine Lab 4.
- Set yourself a due date for the milestone; The lab's due date would do.
- You can leave the description blank, or you can write something like "Create a simulator for the Monty Hall problem".
- Click on "Create Milestone". You should now be looking at your new milestone as the only item in a list, and showing it has no issues in it right now. We are going to add issues in a moment.

Creating custom labels

- Switch to the Labels tab. You are seeing the list of labels that are offered by default: bug, duplicate, enhancement, etc. We will create our own set of labels. You do this as follows:
- Click on the New Label button.
- Type in a name for the label. Our first label will be called "scorecard" and it will be used for issues that have to do with the scorecard functionality of our application.
- Write a description if you like. I wrote: "Related to the scorecard functionality".
- Choose a new color by either clicking the random generator button or typing a hex color representation.

- Click on “Create Label”. You should now be seeing your label amongst the list of available labels.

Repeat this process to create a label “doors” that will contain functionality related to having doors, and one called “game” that will contain functionality related to the overall game process (which guides the user to choose a door, then choose to switch or stay etc).

You will notice that the labels we are creating are fundamentally different than the ones provided by GitHub: those focused on the type of issue (bug, enhancement etc); ours focused on the functionality affected. There are no rules for what labels to use; in general do whatever helps you organize the issues.

Creating our first issue We’ll create an overall planning issue that contains our plan of action.

- Click at the Issues tab near the top. Then click on New Issue on the right.
- The title of our issue will be “Monty Hall Simulator planning”.
- In the “Write” section below, copy and paste the following:

```
We will build a simulator for the Monty Hall problem. This issue will serve to keep trac

- [ ] Make sure we can run tests.
- [ ] Create a simple Score class to hold counts for the outcomes of each round of the g
- [ ] Plan the overall index.html structure and create containers for the various sectio
- [ ] Create basic UI for the scorecard, both HTML and suitable CSS.
- [ ] Create and hook up a controller that relates the Score class to the scorecard UI.
- [ ] Create simple door class that contains two pieces of information about a door: Wh
- [ ] Create a door UI class that represents a visual representation of a door on the we
- [ ] Create a controller that connects the door class to the door UI.
- [ ] Create a game class that drives the overall simulation. It should have the ability
- [ ] Create the UI elements for the overall game, including buttons for resetting and s
- [ ] Create the main game controller that coordinates the simulation.
```

- Click on Submit to create this new issue. You should see a list of checkboxes. We’ll use those boxes to track our overall progress.
- Use the Milestone option on the right to assign this issue to the Lab 4 milestone.

Working on the assignment

Now we start our work on the assignment.

Verifying out testing environment The first order of business would be to verify that our testing environment is all set up. You should have already started an http-server from the terminal, running on the location where your files are checked out. With that you should be able to open up the two files index.html and tests.html and you should not see any worrisome errors in the console for the two pages.

If it all seems to work out and you see one passing test and no errors on the console on either page, go ahead and click that first checkbox in our master issue that said "make sure we can run tests".

Creating the Score class We will start by creating the Score class. We will do it in a test-driven approach.

First, we will create an issue that describes in more detail how the class will work. Go ahead and create such an issue, titled "Create Score class". Add the following to the comment area for the issue:

```
The scorecard class maintains the score for the simulation. It keeps track of four instance v

- `reset()` resets all the counts to 0.
- `addResult(userAction, result)` that takes as input the result of a play and updates the co
```

Make sure to add the **scorecard** label to the issue, and to add it to your **milestone**. After you create the issue, take note of its number (#XXX). You will need it in a few moments.

Now that we have an issue, we can start work on our Score class. We start by creating a test class:

- Create a file test/score.spec.js. Use the test/example.spec.js as a start point.
- Add a script tag for this new file in the tests.html file, below the one for observable.
- Use a better string label for the describe part, to refer to "Score instances" or something like that. In that first it part, your first test, set its string label to say something to the effect of "start with all counts set to zero".
- In the body of the it part, start with something like: let score = new Score();. Now look at your tests page and you should be getting errors that Score is not defined.
- Let's work on defining it. We will need to import it, go before the describe in your test file, and add import Score from '../js/score.js';. Run your tests again, and you should now be receiving an error in the console, because js/score.js doesn't exist.
- Create an empty js/score.js file, and run your tests again. You should be now seeing an error in your test about the file "not having a default export".
- Add an export default class Score {} part to the score.js file. Your tests should now be compiling. Of course we still need to fix our test for initial scores.
- Back inside the it in the score.spec.js test file, we now want to add tests for the four variables. Add the line expect(score.switchWins).to.equal(0); to the tests, and add 3 more similar lines for the other 3 variables that are mentioned in your issue. Run your tests and they should fail because undefined is not equal to 0.
- To make the tests pass, back in the js/score.js file add a constructor inside the Score class, and in that constructor set this.switchWins=0; and similarly for the other three variables. Watch your tests pass.

- Great! Now let's make a commit. Open up GitKraken, and review the changes you made to the three files. If they look OK, go ahead and stage them and make a commit with summary saying "Add basic Score card. Ref #XXX". Here #XXX is the number of the GitHub issue that you created earlier about the score class, so put that number in there instead (for example in my case it looks like so: Ref #2). The ref word is important there. You have to use either ref or close, if you wanted the issue to automatically close at the same time.
- Push your changes, and now in GitHub when you look at your issue you should see a reference in it to the commit you just made.
- You want to do this for each commit you make from this point on, link it as above to a corresponding issue.

Refactoring OK we are not done with the class yet. Let's look back at our issue comment. It looks like we need to add the two functions reset and addResult. It would be nice if we could write reset first, but since reset simply sets the values to 0, and they are already 0, there is no meaningful test we can write for it. We will therefore start from addResult. But before we do so, let's take a closer look at our test code. We are about to add some more it statements, and each of them will need to work with a Score instance. It would be nice if we didn't have to write it every single time. Here's how we will do that:

- Within your describe, but before the first it, we will declare the score variable: `let score;`.
- Below it, but before the it, add a call to beforeEach:
  ```
  beforeEach(() => { score = new Score(); });
  ```
- Remove the `let score = ...` statement from within the it.
- Save and run your tests to make sure they still run OK.

What we did is called *refactoring*: We changed the structure of the code to prepare it for future change, but without changing the behavior. We will be doing a number of these refactorings. In fact, we'll do one more right now. This is a popular refactoring, called *extract function*:

Look at the body of our test. We test that the four stored values match four specific numbers, namely 0. I imagine we'll need to do a lot more similar checks in future tests. Instead of repeating these four lines, we'll turn them into a function:

- Copy the four lines.
- Within the describe, but immediately following the it, create a new function called checkValuesAre(), and paste those four lines to its body.
- Replace the body of the it with a called to checkValuesAre.
- Run your tests to make sure they pass.

Hm OK this is nice, but it's not going to do us much good unless the values are always going to be 0. What we need to do next is turn these values into function parameters, and the method that achieves that is called *extract parameter*.

- Add a parameter switchWins to the definition of checkValuesAre.
- Replace the 0 in the first expect in the body of checkValuesAre with switchWins.
- In your call to checkValuesAre from within the it, put the value 0 there. That is where the parameter switchWins will take its value from.
- Run your tests to make sure they still pass.

Now repeat for the other three zeros, to add parameters switchLosses, stayWins and stayLosses. Run your tests after each parameter to make sure you did not mess up anywhere along the way.

Go ahead and create a commit from all these changes. You can put "test refactoring" on its summary. Make sure to reference your issue number!

More tests Now we are finally ready to add more tests! We start with the test for addResult.

- Within the describe but after the first it, start a new it that states that the score class increments the appropriate value when addResult is called. In the body, add a line score.addResult(Score.ACTION_SWITCH, Score.RESULT_WIN); which is supposed to icrement the corresponding score. Run your tests, and they should fail at this point, because scores don't have an addResult method yet.

- Go to the Score class, and add an addResult method, that takes the parameters action and result. It does not need to do anything yet so you don't have to write anything in its body. But run your tests and make sure they all pass.

- Now we need to actually test something. Back in the it test we are writing, add a checkValuesAre(1, 0, 0, 0) line, we expect that to have been incremented. Run your tests to see them fail.

- Now we need to make it pass. Go back to your Score class code and the empty body of addResult. There are many ways we could try to make this work, but we'll simply do nested if statements:

```
if (action == Score.ACTION_SWITCH) {
    if (result == Score.RESULT_WIN) {
        this.switchWins += 1;
    }
}
```

  Add this to the body of addResult and watch your tests pass.

- Now we continue the tests to account for the other cases. Add a score.addResult(Score.ACTION_S line to your test, and a corresponding checkValuesAre(1,1,0,0); line (note that they accumulate). Now run your tests, and notice that they fail but not for the reason we expected: It says that 2 is not equal to 1, not that 0 is not equal to 1.

- We finally need to deal with the fact that we never defined Score.RESULT_WIN etc. So they are all undefined, and therefore all equal to each other. So right now everything we do will increment the switchWins. Let's remedy that. Do the bottom of the js/score.js file, after the class definition ends, add four lines like: Score.ACTION_SWITCH = "switch"; and similarly for the other three constants. Then run your tests again, and they should now be failing for the correct reason.
- The reason is of course that the inner if condition above needs a corresponding else clause that increments the loss count. Add it and then watch your tests pass.
- Add the appropriate pair of lines to test the STAY action and WIN result, and watch your tests fail. Then add an else clause to the outer if statement to increment the STAY cases.
- Finally, add the pair of lines that tests the STAY action and LOSS result, and the corresponding code, and make sure your tests pass.

Great! Time for another commit now, saying that you added the addResult method. Don't forget to reference your issue number!

Now we just need to add the reset function and its tests.

- Add a third it test that talks about the score instances setting all counters to 0 when reset is called. In that test, use some addResult calls to increment the counters, then do score.reset(); then a line about checkValuesAre(0, 0, 0, 0); to verify that reset did in fact reset the values.
- You should see your test fail because reset doesn't exist. Add an empty reset() method to the Score class to see your tests now fail for the right reason, namely 0 not equal to 1.
- Now we need to implement reset. We already did this work in the constructor. So just copy the 4 lines from the constructor and put them in the reset method. Then replace the body of the constructor with a call to this.reset().

Now commit this, and close the issue! You can either do it by including close #... in the commit message, or from the GitHub interface.

Now that we have closed this issue, go back to your master checklist issue and check off the second item.

This concludes part a of the lab. Due to its size, the lab is broken into pieces. Continue to part b[1]

[1] ./4b.html