GHC 2018 Pairing Stations

Summary

The GHC 2018 Pairing Laptops will have multiple pairing exercises available for use during the conference:

Console applications (all found in this repo)
Available in Java, Ruby, Clojure, .NET Core, and JS

- Morse Code Translator
- Queens Attack

Front-End (Browser) Applications

- Morse Code Translator (Vue.js)
    a. Review the README for an overview of how this app works and tips on pairing
    b. Check out Sara's video walkthrough!
- Test Invaders (Javascript)

Infrastructure-As-Code

- Container testing with Docker (Docker and Ruby)
    - Review the README for an overview of how this app works and tips on pairing
    - Check out Rosemary's video walkthrough!

How it Works

We have 4 pairing stations in the booth. This is a great way to interact with people, and they have a lot of fun with it. Make sure you are in the booth during the times assigned to you. If you are scheduled during a conference talk you want to see, make sure you trade times with someone else we in advance; be sure to keep track of the booth schedule!

Some attendees are nervous about pairing. Reassure them that it's not an interview, and it's really more for them to see what it's like to work with us than us evaluating them. If they are hesitant, you can start by doing the typing so they can get comfortable watching you do it. Then ask them if they'd like to type.

There are exercises in Javascript, Ruby, and Java. Try to let the participant choose the language that is comfortable for them. The booth laptops will have the code cloned, IntelliJ and VSCode installed, any any libraries needed for the exercises.

We have a lot of people come through, so these exercises are meant to be fairly short. Test Invaders can go longer if someone is really interested. Be mindful of if there are other people waiting to get a chance to pair. The goal is to give them a taste of how we work, not to keep them in the booth for hours writing a complex application or trying to figure out an algorithm.

Talk to them about what they think about this way of working. Have they ever done TDD before? Pairing? What was good? This is how we work every day. Discuss the TDD cycle, refactoring, etc. Remember to do these

things. Remember to keep them on track - only try to fix one test at a time! They will want to jump ahead and do multiple at once.

Most of them have tests written already, but ignored. Remove the ignore annotations one by one as you make them pass. You can add more tests too. Make small local commits - talk to them about git if they haven't used it Show them IDE shortcuts for refactoring. Get the attendee's information before they leave if they are willing to share it.

Pairing Exercises

Queens Attack

- Language: Java, Ruby, Clojure, and JS
- What it does:
    - Related to a chess board
    - Give the position of the black and white queen pieces and determine if attack is allowed
    - Attack is allowed if they are in the same row (rank), column (file), or diagonal path. They do not have to be next to each other.
    - Validate position (no negatives or out of bounds)
    - Extract position into a new class if you have time
- Staging:
    - In the Ruby directory, 'bundle install' in each app
        - Run 'rspec spec' in each ruby directory to see the tests
    - In the Javascript directory 'npm install' in each app
        - Run 'gulp jest' in each javascript directory to see the tests
    - In the java directory
        - Run 'gradle test' to see the tests for each app
    - In the Clojure directory
        - Run 'lein test' to see the tests for each app
- Hints:
    - There are answers in the Ruby directory if you're curious.

Morse Code Translator

- Language: Java, Ruby, Clojure, .NET Core, and JS
- What it does:
    - Simple translations between English and Morse Code
    - Split letters and words on single or triple spaces
    - Handle validations of input
    - Can translate both ways
- Staging:
    - In the Ruby directory, 'bundle install' in each app
        - Run 'rspec spec' in each ruby directory to see the tests
    - In the Javascript directory 'npm install' in each app

        - Run ‘gulp jest’ in each javascript directory to see the tests
    - In the java directory
        - Run ‘gradle test’ to see the tests for each app
    - In the Clojure directory
        - Run ‘lein test’ to see the tests for each app
    - In the .NET Core directory
        - Run ‘dotnet test --no-build -v n’
- Hints:
    - There are answers in the Ruby directory if you’re curious.

Front-End Morse Code Translator

- Language: Vue.js
- What it does:
    - Simple translations between English and Morse Code
    - Split letters and words on single or triple spaces
    - Handle validations of input
    - Can translate both ways
- Staging:
    - Run `yarn start`
    - Open browser: http://localhost:8080/
- To run tests: `yarn unit`
- Before attendee comes to the booth:
    - Checkout the **pairing-station** branch. All but 2 of the tests are already broken.

Test Invaders

- Language: JS
- What it does:
    - Space invaders game
    - It's fun and the graphics draws people into the booth.
- Gotchas:
    - There are some ruby specs and js specs - don't worry about the ruby ones
    - Be careful with the jasmine tests - if you have certain syntax errors, those test will just stop running and you won’t know they are broken. Or they will run and claim that they pass. There should be 63 specs. (I think!)
- Staging:
    - Run `shotgun` and go to 127.0.0.1:9393/ on the browser. This will pop up with the Test Invaders game.
- To run tests: `rake jasmine:ci`
- Hints:
    - It starts with one failing test on the invader
        - When invader collides with tank bullet, it should die
        - Then fails when invader collides with invader bullet, it should not die
    - Look for the other TODO comment in the tank spec
        - The tank should die if hit by a bullet
        - You just make this pass without a conditional at first

        - Then you'll notice the tank will die when colliding with its own bullet too
        - Write another test when you see the problem.
    - Come up with more changes on your own if you have time…
        - Maybe make the invaders shoot more slowly
        - The tank could have multiple lives so it doesn't die if hit right away
        - The tank shouldn't be able to shoot bullets if it's dead
        - The game should end when the tank dies or the invaders all die
        - Track points

Infrastructure-as-Code

- Language: Dockerfile, Bash
- What it does:
    - Creates a Docker container with a simple purple webpage
    - Starts broken - you'll need to edit the makefile AND Dockerfile to fix the tests.
- Staging:
    - Run `make build-and-run` to pull up the wrong webpage.
    - Open `README` with image of correct webpage
- Before attendee comes to the booth:
    - `git checkout -- .` to return to previous (broken) state
- To run tests: `make test`
- Hints:
    - There are answers in "answers-no-peeking" directory.

Setting up the Pairing Station Laptops

To setup a pairing station laptop, from a 'clean/wiped' ThoughtWorks Macbook, follow the steps on the README of laptop-install.

From 'wiped' MacBook, on WiFi:

1. Set a non-blank password to allow sudo access
2. Remove extraneous things from Dock
3. Install Chrome https://www.google.com/chrome/
4. Install JetBrains IntelliJ IDE Community https://www.jetbrains.com/idea/download/#section=mac
5. Install VSCode https://code.visualstudio.com/download
6. Open VSCode and use Ctrl + Shift + P. Search for "Shell Command: Install 'code' command in PATH" so you can install extensions programmatically.
7. Follow the remainder of the README to install.
8. Get the pair programming repos and run the tests to verify they work.