ECE453/CS447/SE465
Software Testing, Quality Assurance, and Maintenance
Assignment/Lab 1, version 1

Patrick Lam
Release Date: January 16, 2017

Due: 11:59 PM, Monday, January 30, 2017
Submit: via ecgit

Getting set up

After setting up your ssh key at `http://ecgit.uwaterloo.ca`,
fork the provided git repository at `git@ecgit.uwaterloo.ca:se465/1171/a1`:

```
ssh git@ecgit.uwaterloo.ca fork se465/1171/a1 se465/1171/USERNAME/a1
```

and then clone the provided files:

```
git clone git@ecgit.uwaterloo.ca:se465/1171/USERNAME/a1
```

(You can also download the provided files from the course repository (`assignments/a1` in `https://github.com/patricklam/stqam-2017`), but don't do that if you're in the course—it'll make submitting harder.)

An account on `ecelinux.uwaterloo.ca` is available to you. Several of the resources required for this assignment are already installed on these servers. Probably the Vagrant image is easiest to work with. If you are attempting to connect to a server from off campus, remember you will need to connect to the University's VPN first: `https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn/about-virtual-private-network-vpn`

I expect each of you to do the assignment independently. I will follow UW's Policy 71 for all cases of plagiarism.

How to run everything

Please see `a1-technotes.pdf` for more details about how the provided software works.

Submission instructions:

Commit **and push** your modifications back to your fork on `ecgit`. It's git, so you can submit multiple times. After submission, **please re-clone your submissions to make sure you have uploaded all necessary files**.

Submission summary

Here's what you need to submit in your fork of the repo. Be sure to commit and **push** your changes back to `ecgit`.

1. your modified `FormattedCommandAliasTest.java` file in path `shared/bukkit/src/test/java/org/bukkit/command`.
2. in directory `q2`, either file `exploratory.pdf` or `exploratory.txt`, respectively in PDF or text format. I've included `exploratory.tex` which you can LaTeXinto `exploratory.pdf`. (No Microsoft Word files, please).
3. in directory `q3`, file `fix.diff` (generate with `git diff`) and file `failure.pdf`.
4. in directory `q4`, file `no-dead-end.diff` and either file `graph.png` or `graph.pdf`.
5. in directory `q5`, your scripts/test harness with fake data, along with the code you use to generate it. (I used Python), along with file `scalability.pdf` summarizing your timings.

Question	TA in Charge
1	(mostly machine)
2	Jun
3	Song
4	Parsa
5	Xinye

Question 1 (10 points)

For this question, you will write JUnit tests for the `FormattedCommandAlias` class of the Bukkit Minecraft server API to achieve 100% statement coverage.

The provided Vagrant image includes a slightly modified version of Bukkit in the `~/shared/bukkit` directory (bukkit diff also available in course github at `assignments/a1/bukkit-test-instrumentation.diff`. I added a skeleton test class for `FormattedCommandAlias`, called `FormattedCommandAliasTest`.

You can run the tests in the VM with the command `mvn test`. To generate the Jacoco coverage report, use `mvn package`. You'll find the resulting reports in `~/shared/bukkit/target/site/jacoco/org.bukkit.command`.

Your task. Add JUnit unit tests to `FormattedCommandAliasTest` that achieve 100% statement coverage for the `org.bukkit.command.FormattedCommandAlias` class and that verify the results of the computation.

Marking scheme: We will mark your modified `org.bukkit.command.TestFormattedCommandAlias` class. 5 points for coverage (full marks for 100% statement coverage, nonlinearly scaled down for less), 5 points for your tests having passing assertions that verify the output. You will get 0 points if your code doesn't compile.

Questions 2–5: "average"

The next 4 questions are about the "average" Dart webapp, which I've also included in the Vagrant image. I developed this webapp to help me evaluate transfer requests. It is heavily based on the "Tour of Heroes" Angular Dart demo, with additional help from the "Nest o' Pirates" Dart server example. The vagrant image includes the webapp. The `a1-technotes.pdf` document describes how to start the webapp. Once started, you can navigate to your own copy of the webapp at `http://localhost:8088/average` on your computer.

Question 2 (10 points)

In this question, you will perform exploratory testing on "average". The charter will be "Explore the overall functionality of average". (1 point) Summarize in one or two sentences what you perceive as the goal of "average". (5 points) Identify the tasks that "average" should be able to do and classify them as primary or contributing. (You probably want to do this in parallel with your exploratory testing). (1 points) Identify areas of potential instability. (3 points) Produce exploratory testing notes summarizing your findings (one or two paragraphs); in Question 3 you will report a bug, so no need to do that here.

Question 3 (10 points)

(3 points) Identify a failure (bug) in the "average" webapp. (2 points) Write down a sequence of steps to reproduce the bug. (5 points) Implement a fix for the bug and explain the fix; show the incorrect and correct outputs. (Screenshots are probably your best bet).

(Think about the program's input space and poke at corners of the input space.)

Question 4 (10 points)

(8 points) Draw a graph summarizing the navigation structure of "average". Nodes are pages with distinct URLs. Edges are links between these pages as realized by hyperlinks in the app. Submit your graph. (2 points) Identify the dead end (node which has no successors) in the navigation graph and submit a patch which corrects it.

Question 5 (10 points)

The file `shared/average/fake-data-small.sh` contains some fake student and course data. You can feed it to your instance of webapp by sourcing it: from a prompt (either inside or outside the VM), type: "`. ./fake-data-small.sh`"

Your task is to evaluate the scalability of "average". (6 points) To do so, programmatically generate test cases of various sizes.

The scalability parameter to investigate is the number of students. students.

- Each fake student should give rise to one REST API call to `/student` and one to `/student_enrolment`, generating the student and the enrolment in 1A. I recommend randomly drawing the student's program from a small set that you manually create. You may also want to create fake names using a Faker library e.g. `https://github.com/stympy/faker/blob/master/lib/locales/en.yml`.
- Create a fixed set of courses. Mine are randomly generated but yours don't have to be.
- For each student, generate fake course marks. You should generate the course marks randomly about a mean with a given standard deviation.

Incidentally, SE 2021 courses had the following data:

	mean	σ
MATH 115	85.1	12.2
MATH 117	83.0	12.6
ECE 140	88.9	11.5
ECE 105	82.2	13.0
CS 137	82.5	10.6

(4 points) Empirically determine the scalability of the app; the easier way is to manually time the JavaScript execution time (Chrome's developer tools include a Timeline), and the harder (but better) way is to develop a test harness. Include the data points that you collect. Let's say that a running time over 1 second is unacceptable.

Bonus (0 points)

Figure out why the browser console displays a `NullError` exception when loading the "Courses" page and propose a fix. I suspect this is also manifests as a user-visible failure which you could use for Question 3.