SE465
Software Testing, Quality Assurance, and Maintenance
Assignment 1, version 1

Patrick Lam
Release Date: January 16, 2019

Due: 11:59 PM, Wednesday, January 30, 2019
Submit: via git.uwaterloo.ca

Getting set up

We will create a copy of the starter repo for you in your `git.uwaterloo.ca` account. You need to log in to `git.uwaterloo.ca` for that to work.

You will also want to fork a copy ("remix") of the Glitch repo of the webapp at `https://glitch.com/~se465-flashcards`. The "Remix This" button will let you do that.

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.

Submission instructions:

Commit **and push** your modifications back to your fork on `git.uwaterloo.ca`. 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 `git.uwaterloo.ca`.

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 `failure.pdf` or `failure.txt` (containing the description of the failure, the steps to reproduce, and the incorrect and correct outputs) and file `fix.diff`.
4. in directory `q4`, file `backend.pdf` or `backend.txt` (containing your summary of commands and a description of the proposed changes), file `backend-changes.diff` implementing your changes, and finally `shared/selenium/src/test/java/se465/TestBackend.java` implementing your backend as a REST-Assured JUnit suite. Contact me if you want to use a different REST API testing framework.
5. in directory `q5`, file `frontend.pdf` or `frontend.txt` (discussing the feature that makes it difficult to test and the change enabling more controllability), file `frontend-changes.diff` implementing the changes, and `shared/rest-assured/src/test/java/se465/FlashcardsTestSuite.java` with the Selenium test suite.

Question	TA in Charge
1	(mostly machine, supervised by Meet)
2	Jason
3	Meet
4	Michael
5	Parsa

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: "se465-flashcards"

The next 4 questions are about the "se465-flashcards" webapp. I hope that using Glitch will ensure painless setup; when you click the "remix" button on `https://glitch.com/~se465-flashcards`, it will create a copy of the code and set up a virtual machine for you.

This webapp, written in Express + pug, includes both frontend code (in the `public/js` and `views` directories), as well as backend code (in the node.js files in the main directory).

You can get glitch to show you diffs by opening the Logs, and then pressing the Console button. That gets you a console, where you can ask for the `git log` of the project. You can then use `git diff` to print out the diffs that we ask you to submit.

Question 2 (10 points)

In this question, you will perform exploratory testing on "se465-flashcards". The charter will be "Explore the overall functionality of se465-flashcards". (1 point) Summarize in one or two sentences what you perceive as the goal of "se465-flashcards". (5 points) Identify the tasks that "se465-flashcards" 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 "se465-flashcards" webapp. Limit yourself to using the front-end as presented by the application; that is, don't enter custom URLs or send requests to the back-end. (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 (15 points)

Now we'll talk about the back-end. We talked about "controllability" and "observability" for systems. It turns out that the backend API has endpoints that provide controllability but not observability; the frontend directly queries the database to get its results (a poor design!) (1 point) Briefly summarize what each of the commands in `apiRouter.js` can do. (4 points) Propose changes to this API that would make each of the endpoints `/api/update`, `/api/delete`, and `/api/upload` testable. (3 points) Implement these changes and submit a `diff`. (7 points) Write automatic test cases for each of the 3 endpoints. These test cases should ensure that the system is in a known state before they start running. (Your test cases should target your own instance of the webapp.)

REST-assured looks like a good way to use JUnit tests for REST testing: `https://github.com/rest-assured/rest-assured`, although you are free to use a different framework, after checking with me.

Question 5 (15 points)

We'll consider the front-end next. (1 points) What feature of the front-end makes it difficult to test? (2 points) How can we add more controllability to the front-end to make it easier to test? (3 points) Implement your proposed change. (9 points) Create a Selenium test suite that exercises 3 features of the `se465-flashcards` webapp.

In the `shared/selenium` directory in your repo, you will find a `SeleniumExample`. You can run tests from this example using the command:

```
mvn test "-Dtest=se465.SeleniumExample#test*" -Dwebdriver.base.url=http://www.google.com
```

Your test suite should be called `se465.FlashcardsTestSuite` and we should be able to run your tests with this command:

```
mvn test "-Dtest=se465.FlashcardsTestSuite#test*"
```

HINT: You can't get the text from an input element with `getText()`. See instead `http://www.w3schools.com/jsref/prop_text_value.asp`.

Useful reference:

`https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebElement.html`