Iteration 2 Report

Team: Grumpy Old Men

Members:

- Jason Konikow - jk4057
- Christopher Thomas - cpt2132
- Billy Armfield - wsa2113
- Luigi Pastore Pica - lap2204

Link to github repository: https://github.com/jkon1513/ASE2018

Use Cases

Revise your user stories to reflect what is now fully implemented. Expand each user story into a *use case*. Each use case should include at least title, actor(s), description, basic flow and alternate flows (when applicable). Focus on the basic flow vs. alternate flows, which should specify the step by step actions of the user (or external system using your product) and what your product does in response.. You may optionally include other use case elements such as preconditions, postconditions, etc., but this is not required.

- Login/Authentication
    - Actor
        - Individual user
    - Description
        - In order to access all functions related to the map and user profiles, users must enter their user ID and password.
    - Precondition
        - User has downloaded LionGPS
        - User has existing account
        - User has location services enabled
    - Trigger
        - User opens LionGps
    - Flow
        - User is presented a login view
        - User enters username
        - User enters password
        - User taps “Authenticate/Login” to enter the MapOverlay
        - User is allowed to continue to map on success.
    - Alternate flow: user does not have an account
        - User tapps button to create new account
        - User is presented Registration form

        - User enters username
        - User enters password
        - User confirms account creation
    - Exception flow: user incorrectly enters password
        - User sees an incorrect password error message
        - User is returned to sign in form
    - Exception flow: user enters wrong username
        - User sees incorrect username error message
        - User is returned to sign in form
    - Postcondition
        - The app displays the map overlay
        - The user profile is loaded for use

- Columbia Campus Map Overlay
    - Actor
        - Individual User
    - Description
        - Users can see a map of the Morningside campus
    - Trigger
        - User completes login/authentication
    - Flow
        - LionGPS opens homepage which defaults to the MapOverlay, consisting of a google maps MapFragment centered on the users current position on campus
        - User can scroll, zoom in, and pan out to see the campus around them
        - User is free to use the search feature
    - Postcondition
        - The app shows the map overlay centered on the user

- Search for Building with building name or SSOL building alias
    - Actor
        - Individual user
    - Description
        - Users can search for building listing of any building on campus using a formal name, popular alias, or ssol building code
    - Trigger
        - User enters search string into search bar
    - Flow
        - After login, the app defaults to the MapOverlay, which consists of a search bar and map of the Morningside campus, centered on the users position

        - User tapps search bar which brings up the default system keyboard.
        - User types in name of building on keyboard
        - User tapps magnifying glass on keyboard to search for building data
        - If the building name, alias or code is determined to be valid, the map will show a route to the building
    - Exception flow
        - If the building name is not valid, i.e. it is not a building on campus, is misspelled or etc, the user will see an error message and be free to try another search, or pan around the map
    - Postcondition
        - The app will display a route to the building

- Routing
    - Actor:
        - Individual user
    - Description
        - A path is shown from the user’s current location to the destination building
    - Precondition
        - User has entered a valid search term
    - Flow
        - After going through authentication, the user makes a valid building search and the camera centers on the desired building
        - User taps on button with routing icon on the lower right corner of the screen
        - A path from the user’s current location to the destination building is presented on top of the map
        - The location of the user is tracked and shown on the map in real time
    - Postcondition
        - A route appears on the map overlay from the users GPS location to the selected building

Test Plan

For your test plan, explain the equivalence partitions and boundary conditions necessary to unit-test each of the major subroutines in your system (functions, procedures, methods, etc. excluding getters/setters and helpers), with references to your specific test case(s) addressing each equivalence partition and each boundary condition. Your test suite should include test cases from both valid and invalid equivalence partitions, and just below, at, and just above each boundary condition. State where your test suite resides in your github repository, but you do not need to copy the test code into this document.

The unit tests for the business logic of LionGPS's Android activities, namely MapActivity and Login and MapOverlay, are located at:

Test location: ~/ASE2018/LionGps/app/src/test/java/ase/liongps/test/

Log/report location: https://github.com/jkon1513/ASE2018/tree/master/logs

DatabaseInteractorTests.java:

- Equivalence Partitions:
  - For our unit tests for the .getUser("Username") methods, valid inputs are a a string representing a User's username. No other inputs are valid. The User must have been previously added - a name of a user for one that has not been added is invalid input
  - For the getSearches() test, which checks if a LinkedList<String> of searches has been successfully retrieved, there are no equivalence partitions as it is a void function. Whether or not it is successfully retrieved depends on a User having been loaded via firebase, and having passed our authentication system, both of which are scenarios beyond the scope of Unit Testing.
- Boundary Conditions:
  - A string's validity is contingent on whether or not it has been provided by our authentication system, and if it represents a Username (if a string representing a password is passed to the .getUser method, that is invalid).

Constants.java:

- Equivalence Partitions:
  - There are no inputs - this unit tests checks that no other code has mutated constants located in Constants.java.
- Boundary Conditions:
  - Not applicable.

LoginPresenterTest.java:

- Equivalence Partitions:
    - There are no inputs - this test checks that LoginPresenter executes .onDestroy() and that there are no side effects when this method is called.
- Boundary Conditions:
    - Not applicable.

MapOverlayPresenterTest.java:

- Equivalence Partitions:
    - .getRecentSearches() is a method without required arguments and the unit test associated with it checks that a) a null is returned as no user currently exists to be referenced in our code and that b) this method has no side effects (this is ascertained via the Mockito.verifyNoMoreInteractions() method).
    - .initMap() is a void method and the unit test checks that there are no side effects incurred when the test is called. Since it is void there are no inputs to provide.
    - .initUser(“Username”) requires a valid string input, corresponding to the Username of a User Object.
    - .getLocationData(“Location or Location Alias”) requires a string input that belongs to our list of valid locations on campus or our list of aliases of those locations (e.g. “Hamilton” or “Ham” or “HAM”).
- Boundary Conditions:
    - Not applicable.
    - Not applicable.
    - A string’s validity is contingent on whether or not it has been provided by our authentication system, and if it represents a Username (if a string representing a password is passed to the .getUser method, that is invalid).
    - A string’s validity is contingent on whether or not it belongs to our app’s list of valid locations on campus and their aliases. Any string or input beyond that list/set is an invalid input.

SearchInteractorTests.java

- Equivalence Partitions:
    - .getBuilding(name) is tested; a valid input is comprised of a string representing a User that has been added to our system. If the user does not exist, null is returned, violating our junit check.
    - .getShortHand(), .getLat(), .getLng() do not require inputs.
    - .getValidBuildings() does not require inputs.
    - .getValidBuildings.().contains(name) tests the java ArrayList data structure .contains() method, which requires that an object corresponding to the type of the ArrayList (in this case String) is passed to it. The string should belongs to our list of valid locations on campus or our list of aliases of those locations (e.g. “Hamilton” or “Ham” or “HAM”).

    - Our final batch of tests for the Search Interactor checks our .isValidSearch() method. This method returns a boolean whose state returns true when the input string is a member of our list of valid strings, and false when the input string is not a member. The first batch of tests ensures that false is returned when invalid strings are provided, which include alpha-numeric characters, non alpha-numeric ASCII codes, empty strings, strings with spaces, and special tab and newline characters. The second batch of tests ensures that case is ignored when strings are provided, so a string with mixed cases (“HaMilton HaLL”) would constitute a valid input. Strings with trailing or leading whitespaces are also valid.
- Boundary Condition
    - A string that corresponds to anything that is not a user, or to a user that has not been added to our app yet.
    - Not applicable
    - Not applicable
    - Must be the right type and must be a string that belongs to our app’s list of valid locations on campus and their aliases. Any string or input beyond that list/set is an invalid input.
    - Any string that matches our String list of locations and their aliases is valid, regardless of the case of the characters or whether or not they have leading or trailing whitespaces.

UserTest.java:

- Equivalence Partitions:
    - .getName(), .getSearches() and getPw() are all functions that do not require inputs.
    - .updateHistory() requires a string input that corresponds to a search that is being inputted by the user.
- Boundary Conditions:
    - Not applicable.
    - A string’s validity is contingent on whether or not it has come from our app’s search logic - a string representing a username, a password, or any thing that is not a valid location on campus would comprise an invalid input. Our app prevents lookup of any string that isn’t a valid location on campus elsewhere.

Branch Coverage

Then describe how you measured the branch coverage achieved by your test suite. This should include adding a coverage tool to your post-commit CI process, with its reports added to a folder in your github repository. Remember that coverage should strive to achieve 100%, but probably won't; state what coverage you did achieve. Add test cases for loop initiation, continuation and termination even if your coverage tool does not handle loops (if your code has any loops).

Business logic coverage: 77%

As Gail has mentioned in lecture, achieving a high test coverage in an android project is extremely challenging. We took a unique approach to testing our app and actually implemented a system architecture known as MVP (Model - View - Presenter) with the purpose of making our code more modular, and easier to unit test. MVP seeks a separation between business logic, android framework logic, and UI logic. The business logic is what we are most concerned with testing thoroughly, as it is what makes our app function the way it should.

Our approach to testing is as follows:

Our “Model” for each package holds a number of classes called “interactors” that handle business logic for a single given function of our app. For example the “LoginInteractor’ handles logic for logging in a user, while the “RegistrationInteractor” handles logic for registering a new user. We have seeked to achieve a 75% coverage rate with these classes and have mostly succeeded. Why only 75%? Well because the interactors at times need to make network requests, and it is poor form to make network requests from a unit test, therefore we are using Mockito to mock those functions to return sample data that we can then test our business logic against rather than opening a net connection, or reading file i/o. This has been great practice, but unfortunately doing something like that is not counted when coverage is calculated and gives the appearance of a lower coverage. Therefore we have set a goal of 75% coverage

The view and presenter portions of our app we have been less adamant about testing. The presenter is simply an intermediary between the view and the model, hardly ever returns any output, and can only be tested using a mockito.verify call. The view is restricted to simply displaying things on the screen, and utilizes pure android framework calls that we did not write, and have been tested thoroughly by Google developers. In addition to this fact, the proper way of testing the presenter and view is through a different form of testing called “instrumented testing” that requires a mobile phone emulator. Getting that to run in docker during a travis build has proven to be impossible, and is what inspired us to implement the MVP architecture in the first place. The whole point of MVP is to create a more modular structure with all testable logic in one place, i.e. the model. That is what we have built with our app.

Coverage Reports unfortunately, do not take any of this into account. When you generate a coverage report in android studio it scans all code in the entire project. This includes the built in android framework, Gradle, any application library dependencies (in our case google firebase, authentication, maps, and directions) and anything even beyond. Literally thousands of lines of code that we don't even have the access to write unit tests for are factored in to our coverage report, and just drags down our percentage. For this reason we have chosen not to generate these reports and instead view the coverage data of the individual interactors containing our business logic. The coverage is as follows:

DatabaseInteractor: 70%
SearchInteractor: 86%
LoginInteractor: 100%
GeoLocation interactor: 65%
BuildingDataInteractor: 70%
UtilityTests: 80%

Perhaps if we had chosen to create a pure java application we would have made our lives easier, and could boast better coverage reports. The fact is though that we decided to challenge ourselves to learn a new technology, utilized a system architecture weeks before it was even covered in class, and delved deep into testing theory and design to try to come to a close compromise to what Gail was hoping for out of these projects. We ask that you keep this in mind when assessing our project.