Project 2.4: Implementation

Team # 18: Long Island Iced SWEs

Project Title: FindASeat

Khushal Kalidindi (kalidind@usc.edu)

Julian Glover (jtglover@usc.edu)

Ramneek Singh (ramneek@usc.edu)

Preface

At this stage of development, we are testing our application to make sure that it functions as intended. This document will detail all the tests we wrote along with what they tested and what bugs we encountered in writing these tests.

Instructions

Order of blackbox tests to run:

NOTE: On MakeAccountTest, EmailAlreadyExistsTest, IDAlreadyExistsTest,
and any time it prompts you out of the app
(like picking a photo) there is a 6 second thread sleep so you can do so!!

To run each test, you must right click on the respective files in “test” and “androidTest” files in the Android Studio and hit run

1. UserDNELoginTest
2. UserExistsLoginTest
2. MakeResTest
3. CantMakeAnotherResTest
4. ModifyTest
5. CancelTest
6. AllErrorsShowUPTest
7. BasicTests (2 tests)
8. EmailAlreadyExistsTest
9. EmailAndPassDontMatchTest
10. IDAlreadyExistsTest
11. InvalidEmailTest
12. InvalidPasswordTest
13. InvalidResTest
14. MakeAccountTest

Order of whitebox tests to run:

1. DatabaseTests (3 tests)
2. ExampleTest3 (13 tests)

Black-box testing

Each of these tests is in the com.example.findaseatfinal2 (androidTest) folder and to run them, you much click on the file of each respective file and hit 'run'

UserDNELoginTest:

This test opens to the main map page, navigates to the profile tab, and tries to sign in. It is unsuccessful because the user doesn't exist in the database.

UserExistsLoginTest:

This test opens to the main map page, navigates to the profile tab, and they try to sign in. It is successful because the user exists in the database, so the test checks their profile tab that their name is there.

MakeResTest:

This test opens to the main map page, navigates to the profile tab, and they sign in. The user is on the map page, and clicks Annenberg. They go to the next day and make a 9:00 am reservation, and then go to their profile page where the current reservation field contains it. The test uncovered that we could make reservations at times that had already been passed, so we fixed that error.

CantMakeAnotherResTest:

The user has an existing reservation now, so when they sign in, click on a marker, and hit create reservation, it tells them they can't since they already have one.

ModifyTest:

The user signs in, goes to their profile tab and sees the reservation they just made. They wish to modify, and click modify and confirm it by saying "yes". They then make a different reservation at Annenberg with 2 time slots at 13:30. Their profile tab shows this change.

CancelTest:

The user wishes to cancel their reservation, so they sign in, go to their profile tab, and hit cancel, then go to their profile tab which shows it is gone.

AllErrorsShowUpTest:

The user goes to log in, and clicks "Sign Up" and then submit, which gives them errors since they have no input in the fields.

MarkerExistsTest (BasicTest 1):

Upon opening the app, all 10 markers show up.

MarkerDNETest (Basic Test 2):

A marker that shouldn't be there indeed doesn't exist.

EmailAlreadyExistsTest:

The user goes to make an account using the same sign up button, but the email they input already exists, so a warning appears.

EmailAndPassDon’tMatch:

The user goes to log in, and puts an email and password that are both in the database, but since they don’t correspond it fails and an error appears.

IDAlreadyExistsTest:

The user goes to sign up like before, but their ID is already in the database, so an error appears.

InvalidEmailTest:

The user attempts to sign up, but the email input is invalid and an error message appears (no @usc.edu).

InvalidPasswordTest:

The user attempts to sign up, but the password input is invalid and an error message appears (must be 8 characters min and a special character).

InvalidResTest:

The user signs in, clicks a building to make a reservation, but doesn’t select a time, so an error message appears.

MakeAccountTest:

The user can successfully sign in and make a new account in the database.

White-box testing

NOTE: Coverage for Email Validation, Password Validation, ID Validation, Name Validation, Affiliation Validation is mentioned after these sections at “Validation Coverage Summary”

emailValidate: Email Validation Test in RegistrationActivity

The objective of this test is to verify the correctness of the email validation logic implemented in the RegistrationActivity class. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.
Test Cases:

Valid Email Address:

- Input: "julian@usc.edu"
- Expected Output: true
- Description: The test checks whether the method isEmailValid correctly validates a valid email address. In this case, the provided email address "julian@usc.edu" is expected to be valid.

Empty Email Address:

- Input: ""
- Expected Output: false
- Description: This test case assesses the behavior of the email validation method when an empty email address is provided. The method should return false since an empty string is not a valid email address.

Invalid Email Address Format:

- Input: "julian@usc,ed"
- Expected Output: false
- Description: The test evaluates the email validation method's handling of an email address with an invalid format. In this case, the email address contains a comma instead of a dot between the domain parts, making it an invalid format. The method should return false.

passwordValidate: Password Validation Test in RegistrationActivity

The objective of this test is to evaluate the password validation logic implemented in the RegistrationActivity class. The test assesses the correctness of the isPasswordValid method by checking various scenarios of password input. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Password:

- Input: "djhdhhdshj&"
- Expected Output: true
- Description: The test checks whether the method isPasswordValid correctly validates a strong and valid password. The provided password "djhdhhdshj&" meets the validation criteria and is expected to be valid.

Invalid Password (Weak):

- Input: "hdshjhdhdhhd"
- Expected Output: false
- Description: This test case evaluates the behavior of the password validation method when an input password is considered weak. The password "hdshjhdhdhhd" does not meet the validation criteria and is expected to be invalid.

Invalid Password (Short):

- Input: "hdhh&"
- Expected Output: false
- Description: The test assesses the handling of a password that is too short to meet the validation criteria. The password "hdhh&" is below the minimum length and is expected to be invalid.

Invalid Password (Empty):

- Input: ""
- Expected Output: false
- Description: This test case checks how the method handles an empty password input. An empty password is considered invalid, and the method is expected to return false.

IDValidate: ID Validation Test in RegistrationActivity

The objective of this test is to verify the correctness of the ID validation logic implemented in the RegistrationActivity class. The test evaluates the behavior of the isIDValid method by testing various scenarios of ID input.The test is located in the ExampleTest3 file

under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid ID:

- Input: "6745674556"
- Expected Output: true
- Description: The test checks whether the method isIDValid correctly validates a valid ID. The provided ID "6745674556" meets the validation criteria and is expected to be valid.

Invalid ID (Empty):

- Input: ""
- Expected Output: false
- Description: This test case assesses the handling of an empty ID input. An empty ID is considered invalid, and the method is expected to return false.

Invalid ID (Short):

- Input: "1727"
- Expected Output: false
- Description: The test evaluates how the method handles an ID that is too short to meet the validation criteria. The ID "1727" is below the minimum length and is expected to be invalid.

Invalid ID (Long):

- Input: "3783273787327837"
- Expected Output: false
- Description: This test case checks the behavior of the method when presented with an ID that exceeds the maximum length. The ID "3783273787327837" is longer than the allowed length and is expected to be invalid.

nameValidate: Name Validation Test in RegistrationActivity

The objective of this test is to validate the correctness of the name validation logic implemented in the RegistrationActivity class. The test assesses the behavior of the isNameValid method by testing two scenarios of name input. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Name:

- Input: "Julian"
- Expected Output: true
- Description: The test checks whether the method isNameValid correctly validates a valid name. The provided name "Julian" meets the validation criteria and is expected to be valid.

Invalid Name (Empty):

- Input: ""
- Expected Output: false
- Description: This test case assesses the handling of an empty name input. An empty name is considered invalid, and the method is expected to return false.

affiliationValidate: Affiliation Validation Test for RegistrationActivity

The objective of this test is to validate the correctness of the affiliation selection logic implemented in the RegistrationActivity class. The test assesses the behavior of the isAffiliationSelected method by testing two scenarios of affiliation input. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Affiliation:

- Input: "Graduate"
- Expected Output: true
- Description: The test checks whether the method isAffiliationSelected correctly validates a valid affiliation selection. The provided affiliation "Graduate" is expected to be valid.

Invalid Affiliation (Default Selection):

- Input: "Select:"
- Expected Output: false
- Description: This test case assesses the handling of the default or invalid affiliation selection "Select:". An affiliation selection of "Select:" is considered invalid, and the method is expected to return false.

loginEmailValidate: Login Email Validation Test for LoginActivity

The objective of this test is to validate the correctness of the email validation logic implemented in the LoginActivity class. The test assesses the behavior of the isEmailValid method by testing three scenarios of email input. The test is located in the ExampleTest3 file under the

com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Email:

- Input: "julian@usc.edu"
- Expected Output: true
- Description: The test checks whether the method isEmailValid correctly validates a valid email address. The provided email "julian@usc.edu" meets the validation criteria and is expected to be valid.

Invalid Email (Empty):

- Input: ""
- Expected Output: false
- Description: This test case assesses the handling of an empty email input. An empty email is considered invalid, and the method is expected to return false.

Invalid Email (Short):

- Input: "j@usc.ed"
- Expected Output: false
- Description: The test evaluates how the method handles an email address that is too short to meet the validation criteria. The email "j@usc.ed" is below the minimum length and is expected to be invalid.

Validation Coverage Summary: 10/22, since these checks in total, (including Sign In and Affiliation Tests) check if else for every non-database verification text fields, with 2 each, since they test for if and else. They do not check for database related checks, which are done through BlackBox tests. Or the image upload check, which is also done through BlackBox Tests

loginPasswordValidate: Login Password Validation Test for LoginActivity

The objective of this test is to validate the correctness of the password validation logic implemented in the LoginActivity class. The test assesses the behavior of the isPasswordValid method by testing four scenarios of password input. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Password:

- Input: "djhdhhdshj&"

- Expected Output: true
- Description: The test checks whether the method isPasswordValid correctly validates a strong and valid password. The provided password "djhdhhdshj&" meets the validation criteria and is expected to be valid.

Invalid Password (Weak):

- Input: "hdshjdhhdhhd"
- Expected Output: false
- Description: This test case evaluates the behavior of the password validation method when an input password is considered weak. The password "hdshjdhhdhhd" does not meet the validation criteria and is expected to be invalid.

Invalid Password (Short):

- Input: "hdhh&"
- Expected Output: false
- Description: The test assesses the handling of a password that is too short to meet the validation criteria. The password "hdhh&" is below the minimum length and is expected to be invalid.

Invalid Password (Empty):

- Input: ""
- Expected Output: false
- Description: This test case checks how the method handles an empty password input. An empty password is considered invalid, and the method is expected to return false.

Coverage: 8/14, since these checks cover the validation of non-database related checks for valid email and password, (4 for email and 4 for password) but do not cover database related checks, which are covered through BlackBox

CheckBuildingSpotsAvail: Building Spots Availability Spot Setting Test for ReservationActivity

The objective of this test is to verify the correct application of tags to views and text views in the ReservationActivity class. The test assesses the behavior of the setTagsForViews method by checking the tags assigned to different views representing building spots.The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Check Unavailable Spot:

- Spot Index: 0
- Expected Tag: -1
- Description: The test checks whether the method correctly assigns a tag of -1 to an unavailable building spot at index 0.

Check Available Spot (Middle Range):

- Spot Index: 20
- Expected Tag: 2
- Description: The test verifies that an available building spot at index 20 is correctly assigned a tag of 2.

Check Available Spot (Upper Range):

- Spot Index: 39
- Expected Tag: 2
- Description: This test case confirms that an available building spot at index 39 is correctly assigned a tag of 2.

Check Available Spot (Lower Range):

- Spot Index: 17
- Expected Tag: 2
- Description: The test ensures that an available building spot at index 17 is correctly assigned a tag of 2.

Check Unavailable Spot (Lower Range):

- Spot Index: 16
- Expected Tag: -1
- Description: This test case checks whether an unavailable building spot at index 16 is correctly assigned a tag of -1.

Check Unavailable Spot (Upper Range):

- Spot Index: 40
- Expected Tag: -1
- Description: The test confirms that an unavailable building spot at index 40 is correctly assigned a tag of -1.

Check Unavailable Spot (Last Spot):

- Spot Index: 47
- Expected Tag: -1
- Description: This test case verifies that the last building spot at index 47, which is unavailable, is correctly assigned a tag of -1.

CheckAllBuildingSlotAvail: Checking Availability Tags for All Building Slots in ReservationActivity

The objective of this test is to ensure that the setTagsForViews method in the ReservationActivity class correctly assigns tags to views and text views representing building slots. The test specifically checks if the tags are set appropriately to represent the availability of

building slots within the defined range. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Check Available Slots:

- Slots Index Range: 17 to 39 (inclusive)
- Expected Tag: 2
- Description: The test verifies that the tags for views representing available building slots within the specified range are correctly set to 2.

Check Unavailable Slots (Before and After Available Range):

- Slots Index Range: 0 to 16 and 40 to 47
- Expected Tag: -1
- Description: This test checks that the tags for views representing unavailable building slots, both before and after the available range, are correctly set to -1.

Coverage: Branch 2/2, since these tests cover for all possible values that can be inputted into setTagsForViews and verifies that for any possible List of text views which are structured like how they are used in the code

Reset Square

The `resetSquaresTest` method is a comprehensive unit test for the `resetSquares` function within the `ReservationActivity` class. The objective of the test is to validate that the `resetSquares` method accurately updates the background colors of a collection of view objects, reflecting their respective states. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Detailed Breakdown of the Test:

1. Test Setup:
   - An instance of `ReservationActivity`, referred to as `RA`, is instantiated. This provides access to the `resetSquares` method to be tested.
   - A list of 48 `View` objects is initialized, representing slots in the reservation system. Each view is initially tagged with a value of `2`, presumably indicating a default state.

2. Modifying Specific Views:
   - The test alters the tags of two specific views within the list: the 31st view (index 30) is set to `0` and the 36th view (index 35) to `1`. These tag values likely represent unique states that need to be handled differently by the `resetSquares` method.

3. Invoking the Tested Method:
  - `resetSquares` is called on the `RA` instance, passing in the list of prepared views. The method is expected to evaluate each view's tag and update its background color accordingly.

4. Assertions:
  - The background colors of three views from the list are checked:
    - The view at index 20 is asserted to have a background color of `Color.GREEN`, indicating it retains or is reset to the default state.
    - The view at index 30 is checked for a background color of `Color.RED`, suggesting a change due to its unique tag.
    - The view at index 35 is verified to have a background color of `Color.GREEN`, implying that it has been reset to the default state, despite its tag alteration.

5. Test Objective:
  - The primary aim of this test is to confirm that the `resetSquares` function correctly interprets and responds to the tag values of the views, updating their background colors to reflect their current states. This includes maintaining the default state and handling special conditions appropriately.

Reset Squares Coverage: Branch: 2/2 since these tests cover for all possible values that can be inputted into resetSquares and verifies that for any possible List of text views which are structured like how they are used in the code

tagTest: Checking Tags for Building Slots in ReservationActivity

The objective of this test is to verify the behavior of the validChecker method in the ReservationActivity class. The test checks whether the method correctly determines the validity of a selected building slot based on its tag and the number of slots required. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Slot with Sufficient Slots:
- Selected Slot: Index 20
- Number of Slots Required: 3
- Expected Result: Valid (True)
- Description: The test verifies that the validChecker method correctly determines that the selected slot at index 20 is valid for a reservation requiring 3 slots.

Invalid Slot with Insufficient Slots:
- Selected Slot: Index 29
- Number of Slots Required: 4

- Expected Result: Invalid (False)
- Description: The test ensures that the validChecker method correctly identifies that the selected slot at index 29 is invalid for a reservation requiring 4 slots.

Invalid Slot with Unavailable Tag:

- Selected Slot: Index 42
- Number of Slots Required: 2
- Expected Result: Invalid (False)
- Description: This test checks that the validChecker method correctly identifies that the selected slot at index 42 is invalid due to its tag being -1 (unavailable).

tagTest Coverage: Branch 1/1 since this function determines if a tag can be selected or not based on if it is available, it only has one possible branch that it can go in. The branch is true or false. If the value is anything less than or equal to 0, then the function returns false and that is the one branch it covers.

timeHasPassedTest: Checking Time Validations in ProfileActivity

Objective: The objective of this test is to ensure that the validateTime method in the ProfileActivity class correctly validates reservation times based on the current date, slot index, and reservation date. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Cases:

Valid Reservation Time:

- Slot Index: 40
- Reservation Date: 12/01/2023
- Expected Result: Valid (True)
- Description: The test checks that the validateTime method correctly validates a reservation time for a future date and slot index.

Invalid Reservation Time (Past Date):

- Slot Index: 40
- Reservation Date: 10/30/2023
- Expected Result: Invalid (False)
- Description: This test ensures that the validateTime method correctly identifies an invalid reservation time for a past date.

Invalid Reservation Time (Before Slot Time):

- Slot Index: 40
- Reservation Date: 11/01/2023
- Expected Result: Invalid (False)

- Description: The test verifies that the validateTime method correctly identifies an invalid reservation time for a date before the current slot time.

Invalid Reservation Time (Same Date and Slot):

- Slot Index: 28
- Reservation Date: 11/11/2023
- Expected Result: Invalid (False)
- Description: This test ensures that the validateTime method correctly identifies an invalid reservation time for the same date and slot.

Valid Reservation Time (Same Date and Later Slot):

- Slot Index: 34
- Reservation Date: 11/11/2023
- Expected Result: Valid (True)
- Description: The test checks that the validateTime method correctly validates a reservation time for the same date and a later slot.

timeHasPassedTest Coverage: 5/5, since the test validates that for all checks for, month day or hour, the reservation can only be shown as current for valid times

validResTime: Checking Valid Reservation Times in ReservationActivity

Objective: The objective of this test is to verify the behavior of the validTime method in the ReservationActivity class. The test checks whether the method correctly validates reservation times based on the current date, slot index, and the selected slot's tag. The test is located in the ExampleTest3 file under the com.example.findaseatfinal2 (test) folder and to run the test, you must right click on the file and run that file.

Test Method:

Test Cases:

Valid Reservation Time with Available Slot:

- Slot Index: 32
- Tag of Selected Slot: 0
- Expected Result: Valid (True)
- Description: The test verifies that the validTime method correctly validates a reservation time for an available slot with tag 0.

Valid Reservation Time with Available Slot:

- Slot Index: 28
- Tag of Selected Slot: 1
- Expected Result: Valid (True)

- Description: This test ensures that the validTime method correctly validates a reservation time for an available slot with tag 1.

Invalid Reservation Time with Unavailable Slot:

- Slot Index: 28
- Tag of Selected Slot: 0
- Expected Result: Invalid (False)
- Description: The test checks that the validTime method correctly identifies an invalid reservation time for an unavailable slot with tag 0

timeHasPassedTest Coverage: 5/5, since the test validates that for all checks for, month day or hour, the reservation can only be made at valid times

Database Tests

The following tests are for functions which verify that expected changes to the database were actually made. These tests are used in code and in the blackbox tests which have the user make changes that affect the database. It uses a combination of Robotium and JUnit, in order to initilize database before running tests for the functions. To run these tests, you must right click and run file DatabaseTests in com.exmaple.findaseatfinal2 (androidTest)

The tests utilize an `IsPassed` object for asserting outcomes and include exception handling for interruptions during waits. Setup (`setUp`) and teardown (`tearDown`) methods are implemented for initializing the Robotium `Solo` instance and post-test cleanups.

1. `testUserAddedVerificationTest`:

This test assesses user-related functionalities with `FirebaseVerifier`. It conducts three checks:
- Verifies that a specific user (ID "4219746589") is correctly added (expected to pass).
- Ensures proper handling of a non-existing user (ID "0") (expected to fail).
- Checks for failure upon altering the test user's name to an incorrect value, despite using the correct ID (expected to fail).

Each scenario includes a 2-second wait to account for asynchronous operations.
Coverage: Branch, 3/4 = 75%, this is because these tests are used to verify it corresponding BlackBox test for User Registration adds a user correctly, so by verifying these functions work, it indirectly validates that the if statements regarding checking that the ID’s are added correctly to database without duplication and that the user is correctly created after those checks are executed, it only doesn’t account for user name duplication.

2. `testResAddedVerificationTest`:

Focused on reservation functionalities, this test:
- Verifies the correct addition of a reservation (ID "1") with specific attributes (expected to pass).
- Checks the response to a non-existing reservation (ID "-1") (expected to fail).
- Verifies the failure when modifying the `indoor` attribute of the reservation incorrectly, while using the correct ID (expected to fail).

It follows a similar 2-second wait pattern after each operation.

Coverage: Branch, 4/4 = 100%, this is because these tests are used to verify it corresponding BlackBox test for Reservation Addition adds a user correctly, so by verifying these functions work, it indirectly validates that the if statements regarding checking that the ID's are added correctly to database without duplication and that the reservation is correctly created after those checks are executed

3. `testResCancelledVerificationTest`:

This test examines reservation cancellation:
- Verifies the non-cancellation of a reservation (ID "1") (expected to pass).
- Confirms the cancellation status of a different reservation (ID "12") (expected to pass).

Each check is followed by a 2-second delay.

Coverage: Branch, 4/4 = 100%, this is because these tests are used to verify it corresponding BlackBox test for Reservation Cancellation cancels a user correctly, so by verifying these functions work, it indirectly validates that the if statements regarding checking that the ID's which are canceled are canceled correctly in the database without loss of information.

Summary of Tools Used

JUnit:

- Employed primarily for white-box testing, JUnit enabled the creation of assertions to validate the output of our functions.
- It also supplemented our black-box tests, providing a means to assert outcomes for Robotium and UI Automator tests.

Robotium:

- Used for developing black-box tests which simulated user interactions directly with the app, without requiring knowledge of the underlying codebase. This included tests for adding users, as well as adding, cancelling, and modifying reservations.

- Additionally, Robotium facilitated the initialization of Firebase for specific database validation. These white-box tests were crucial in verifying that the black-box tests resulted in the intended modifications to the database.

UI Automator:

- Utilized for black-box testing of the Map UI, UI Automator was instrumental in verifying that pop-ups for buildings appeared correctly and behaved as expected, ensuring they were fully interactive and functional.