CircuitSolver Test Plan

Introduction

CircuitSolver is a mobile app targeted for engineering students that allows one to analyze the voltage drop and current across components in a hand-drawn electrical circuit. It aims to eliminate the inconvenience (and time) of using in-app circuit builders and printable components kits when trying to analyze circuits. Our app consists of various components and uses multiple technologies and frameworks (OpenCV, TensorFlow, NgSPICE engine, Android Studio). Hence, our app is particularly vulnerable to issues of performance and integration complexity. Our testing plan addresses this vulnerability and contains strategies to mitigate their effects. Furthermore, key to our product's success is a smooth user experience, since it's goal is to save time and effort in solving circuits. Thus our testing strategy also incorporates feedback from users in our target group.

Verification Strategy

This product is designed for engineering students with the purpose of simplifying the analysis of hand-drawn circuits. Thus, our verification strategy stresses importance on user experience feedback from peer engineering students. Correctness of logic, which is also vital to the product's success, will be tested thoroughly by unit and integration tests to ensure a robust system. Key points of our verification strategy are included below.

- **External feedback on UI mockups/prototypes/iterations**: As we prototype our UI, we will seek feedback from at least 5 other engineering students and note any common issues they report. Each new iteration of the UI design, we will receive feedback from at least 1 other engineering student.
- **Internal review of UX**: Our own team members are part of the target audience, so our feedback is meaningful for this product as well. All team members will test the UX regularly through the emulators (at least 3 times a week), and at least 2 people with will be testing on their Android devices (at least once a week).
- **Automated UI tests**: Once enough of our UI is built, we will write automated UI tests for happy path scenarios (and exception/error scenarios if time) which should be run at least 3 times a week. This would ensure new code changes do not break our minimum viable product.
- **Unit and integration tests**: The above points are focused on UX, but this point is the foundation for ensuring our circuit analysis logic itself is correct. See functional testing strategy for details.

Non-functional Testing and Results

Portability testing: Test our app on both “virtual” devices on the emulator and physical devices with different resolutions/screen sizes. Make sure everything is still usable. Tests defined below will be used across different devices using a fast emulator such as genymotion.
Performance testing: Test our app with circuits of varying complexities. Ensure that image recognition and circuit analysis can be performed in reasonable time (5 seconds max). In Donald Knuth's paper "Structured Programming With GoTo Statements", he wrote: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%." Since we are currently at first trying to get our app working we have not looked into tracking tools to measure system time and plan on using simple approaches such as *System.nanotime()* to measure execution time of particular identified bottlenecks (i.e. such as a triple for loop we currently have in our OpenCV service). Results will be documented in the associated issue on Github (see organization under Functional Testing Strategy)

Functional Testing Strategy

Given time constraints and how our code will likely change drastically in the coming weeks as we realize prior designs aren’t as feasible as we initially thought (as happened in the JPacman undo assignment) we seek a testing strategy with a low learning curve, that is easy to set up, maintain and is focused more on testing features rather than code design and expectation. Functional testing will involve unit testing (testing components in isolation - doesn’t require an emulator) coupled with a mocking framework to make mock classes to test them and all dependencies. However, this will only be necessary if we’re using some sort of dependency inversion to abstract the implementation details of our services. We believe we are doing this with our usage of OpenCV, but are not sure, so we may end up not needing a mocking framework. Secondly we will require instrumentation testing (driving the activity lifecycle manually - requires an emulator) to test the UI itself through simulated user interaction. With this in mind we have the following strategies

Framework choice and reason

- For Unit testing options available include RoboElectric and JUnit. We will use JUnit as it is more familiar, has better integration with the popular mocking framework Mockito, providing more realistic testing (runs on the emulator rather than a JVM) and is considered easier to set up. It is regarded as being harder to maintain in the long-term, but given our time constraints this is not an issue. Finally, it is typical practice to move to JUnit later in production for more rigorous testing. We don't have the time to move between frameworks. [Source]

- For a mocking framework, options include Mockito and EasyMock. Here we choose Mockito as it has far more support from the StackOverflow community and works well with JUnit. Moreover, EasyMock has been noted to focus more on code design and tests will break if code design is violated even when the feature is still working. [Source]
- Finally, for Instrumentation testing options available include Espresso and UIAutomator. Here we pick Espresso because it is considered best for small scale apps, easier to set up and works well with Mockito. UIAutomator on the other hand has a steeper learning curve and is focused more on external OS integration and testing across multiple applications at the same time. None of these features are required by us. [Source, Source].

Strategy

- Unit tests will be built for for each method in classes that performs reasonable work (not getters and setters). Since we were able to reach a 90+% threshold for code coverage (for class, method and line coverage) for non-ui elements in our assignment with the undo functionality working fine we will aim for this coverage threshold. Aiming for as close to 100% is ideal, however code coverage approaches 100% asymptotically. Consequently, the last 10% is likely more effort than it is worth, offering small returns for the effort expended.
- Automated UI testing will be performed for each of our use-cases specified in our Requirements Documentation. This will be done using Espresso and the mocking frameworks Mockito. Testing our use cases in this way would further provide integration testing to ensure all backend components (OpenCV/TensorFlow/circuit-logic) behave across each other as intended.
- Unit and Instrumentation tests should run on dev machines before each merge, ensuring iterative changes do not change feature requirements.
- Every significant feature, bug fix, refactoring, etc is given an issue using our GitHub repo's issue tracking. Each issue is assigned to at least one person and attached to at least one of the following labels:
    - OpenCV
    - TensorFlow
    - Circuit-Logic
    - UI
- Issues will furthermore be given labels based on which item they are tied to on our product backlog and what that item's priority is (very high, high, medium, low priority). Github issue tracking is well known by all of us meaning no learning curve is required and we became even more familiar with it through the JPacman assignment.

Test Cases and Results

We'll make use of the same user stories and scenario's template as specified in the JPacman assignment. Each of these will describe a particular test to be performed. Each of these will eventually become its own Github issue as need be with some already there

(note at the time of this submission we have not finished this section. We are working on submitting an updated version soon)

```
Title (one line describing the story)

Narrative:
As a    [role]
I want  [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 2: ...
```

Our Use Cases will guide the user stories which in turn will guide our test plan (see table below)

USER STORY 1:

```
Title: Edit a previous saved circuit in order to change some components or
change numerical values:

Narrative:
As a user
I want to edit components or numerical values of previously saved circuits
So that I can re-analyze the circuit following changes

Scenario 1: Title
Given the user has a View Circuit page open on CircuitSolver
 And the user previously used the app to analyze that circuit
When  the user taps "edit"
Then  the application displays the Edit Circuit screen
 And all components

Scenario 1: Title
```

```
Given the Edit Circuit screen is open on CircuitSolver
 And the user previously used the app to analyze a circuit
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...
```

USER STORY 2:

```
Title: View component values of a previous analyzed circuit

Narrative:
As a    [role]
I want  [feature]
So that [benefit]

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...
```

```
Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...
```

USER STORY 3:

```
Title: Solve a circuit from an uploaded or taken circuit picture

Narrative:
As a    [role]
I want  [feature]
So that [benefit]

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...

Scenario 1: Title
Given [context]
 And [some more context]...
When  [event]
Then  [outcome]
 And [another outcome]...
```

The most important part of the test plan is the actual list of tests to be executed. Your entries in this list should be sufficiently detailed that someone else on the project could construct and execute the test based on the test description. Your test plan serves not only to describe what should be tested, but it also documents the results of the tests. Particularly for failed tests, it should hold sufficient information so that the developer tasked with fixing the problem can recreate it. Note: If you are using a separate problem/bug tracking tool, you can merely include a reference to the problem number. Here is a sample table. It might be useful to have separate tables with each table focusing on different parts of the program and/or different levels of testing.

See https://github.com/Frikster/CircuitSolverApp/issues for our issue tracking.

TEST CASES BASED ON EACH SCENARIO
SCENARIO BASED ON EACH USE CASE
Test # x.y = From use case x, scenario y

Test #	**Requirement Purpose**	**Action/Input**	**Expected Result**	**Notes**
	Test that a circuit component can be edited	A list of mock circuits saved,		