NOTE! THERE ARE MORE THOROUGH INSTRUCTIONS TO RUN TESTS IN THE README FILE IN THE PROJECT.

Coverage Reports

This project contains two coverage reports, one for GUI Testing and another for Unit Testing. GUI utilized Jacoco for coverage while the unit tests utilized the built-in IntelliJ code coverage. We were unable to combine these two reports and so below you will find the results of the separate coverage reports. Note that for some classes both GUI and unit tests were applied and so both tests compensated for parts which the other could not reach, not including the code's own limitations.

Application (GUI) Test Report

Activity Name	Statement Coverage	Method Coverage	Branch Coverage
TutorialActivity	96%	100%	70%
SettingsActivity	33%	50%	0%
AboutActivity	100%	100%	50%
HelpActivity	100%	100%	n/a
MainActivity	98%	100%	81%
StatsActivity	94%	100%	60%
GameActivity	75%	100%	66%
TOTAL*	83%	97%	73%

*note that this includes the class “Element” which is not an activity, which is what the GUI tests were on; as a result, the overall total is lower.

Also, pdf of the main page of activities on the GUI tests have been included in the submission file. The GUI tests take about 25 minutes to run on my machine.

TutorialActivity

We currently have 96% statement coverage, 100% method coverage, and 70% branch coverage. 100% statement or branch coverage isn’t possible, as, on the methods that are called (which is all of them), we are able to test every line and branch that we can, as there are some limitations based on the SDK of the emulator, hard-coded values such as *dots.length* or *switch(position)*, or try-catch blocks that throw an exception by normal use of the program (which is poor programming technique to expect an exception…).

SettingsActivity
We currently have reached 33% statement coverage, 0% branch coverage, and 50% method coverage in SettingsActivity. However, we are unable to get 100% statement or branch (or even method) coverage in this activity, as the methods *bindPreferenceSummaryToValue(Preference preference)*, *isXLargeTablet(Context context),* and *isValidFragment(String fragmentName)* are all uncalled, which makes it impossible for us to test. On the methods that are called, we are able to test every line and branch.

AboutActivity
We currently have reached 100% statement coverage, 100% method coverage, and 50% branch coverage in AboutActivity. However, we are unable to get 100% branch coverage, as, in the *onCreate(Bundle)* in the AboutActivity, there are some checks for if objects are null; however, it is not possible for a user to make these objects null (at least, not through interacting with the GUI).

HelpActivity
We currently have reached 100% statement coverage and 100% method coverage. However, since the file is rather simple, the jacoco report says that the method coverage is “n/a.” By examination of the file, and by observation of the HelpActivity tests, I believe this has been thoroughly tested.

MainActivity
We currently have reached 98% statement coverage, 100% method coverage, and 81% branch coverage in SettingsActivity. However, we are unable to get 100% branch coverage in this activity, as we have already tested every line and branch that we can, as there are some limitations based on the SDK of the emulator, hard-coded values such as *dots.length* or *switch(position)*, try-catch blocks that throw an exception by normal use of the program (which is poor programming technique to expect an exception…), or if-statements that depend on a value that we can’t change from the GUI (like *if (current < layouts.length)* or *if (current >= 0)*).

StatsActivity
We currently have reached 94% statement coverage, 100% method coverage, and 60% branch coverage in SettingsActivity. Notably, this class also has unreachable code (i.e. *return super.onOptionsItemSelected(item);* after return statements for the only two possible cases, or unreachable exceptions).

GameActivity
We currently have reached % statement coverage, % method coverage, and % branch coverage in SettingsActivity. A majority of the misses that we experienced have to do with the fact that, due to the poor structure and implementation of the code, we cannot reliably test the GameBoard in a way that we can check for a game over or a 2048; this is where we would reach out to the developers and ask for the code to be refactored in a way that allows us to use mocks, which is not possible as is, due to the fact that the intents that need to be launched are launched from other intents that depend on outside variables that cannot be mocked.

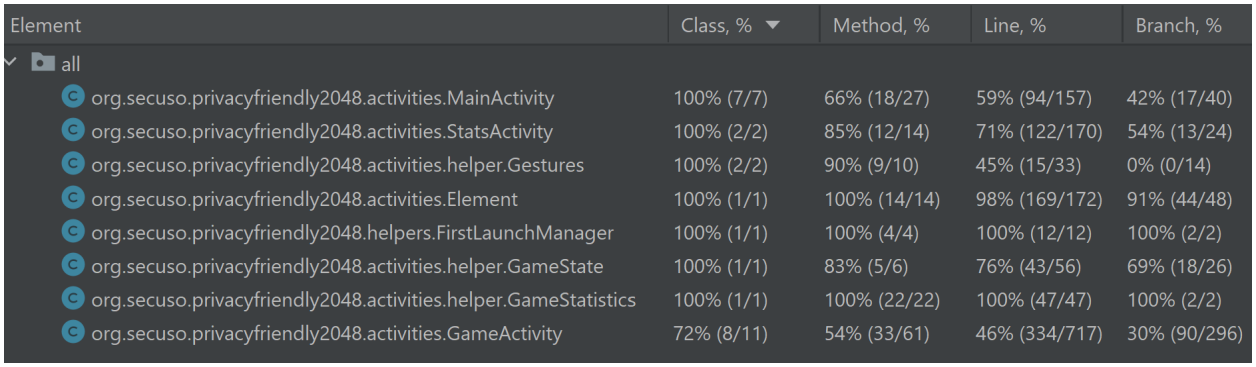
New Failures

The GUI tests did not initially find any faults. However, with our additional tests, we see that, after resetting the game statistics via the StatsActivity, the Highest Number does not get updated to zero, even though all of the other numbers do. Additionally, resetting the game statistics in one tab (for example, 4X4), resets the game statistics in all tabs. I am considering this a failure as there is no documentation, or warning, that this is the expected behavior, and I personally believed that resetting the game statistics in one tab would not affect the others. I also notice that the id for moves, redos, and undos on the statistics page for 6x6 board and 7x7 board are not properly implemented with their id name; so, the tests fail, not because the value is right, but because the id is not right, even though the code in the StatsActivity implies it should use those ids. It is worth noting, a large majority of the code that we missed in our coverage is due to unreachable code, either by methods not being called or differences in the SDK.

Unit Test Coverage Report

Class Name	Method Coverage	Line Coverage	Branch Coverage
GameStatistics	100%	100%	100%
GameState	83%	76%	69%
Element	100%	98%	91%
FirstLaunchManager	100%	100%	100%
Gestures	90%	45%	0%
GameActivity	54%	46%	30%
StatsActivity	85%	72%	54%
MainActivity	66%	59%	42%
TOTAL	**74.5%**	**61.3%**	**41.2%**

IntelliJ Coverage Report



Element	Class, %	Method, %	Line, %	Branch, %
all				
org.secuso.privacyfriendly2048.activities.MainActivity	100% (7/7)	66% (18/27)	59% (94/157)	42% (17/40)
org.secuso.privacyfriendly2048.activities.StatsActivity	100% (2/2)	85% (12/14)	71% (122/170)	54% (13/24)
org.secuso.privacyfriendly2048.activities.helper.Gestures	100% (2/2)	90% (9/10)	45% (15/33)	0% (0/14)
org.secuso.privacyfriendly2048.activities.Element	100% (1/1)	100% (14/14)	98% (169/172)	91% (44/48)
org.secuso.privacyfriendly2048.helpers.FirstLaunchManager	100% (1/1)	100% (4/4)	100% (12/12)	100% (2/2)
org.secuso.privacyfriendly2048.activities.helper.GameState	100% (1/1)	83% (5/6)	76% (43/56)	69% (18/26)
org.secuso.privacyfriendly2048.activities.helper.GameStatistics	100% (1/1)	100% (22/22)	100% (47/47)	100% (2/2)
org.secuso.privacyfriendly2048.activities.GameActivity	72% (8/11)	54% (33/61)	46% (334/717)	30% (90/296)

Running Unit Tests with Coverage

To run all unit tests right click on the unit test test folder and select “Run with coverage”. This should generate the coverage report using IntelliJ. If no report is generated, modify the run configurations by adding the project folder as a package and click on the modify tab and select use tracing in order to generate branch coverage. Unit tests were only written for the classes above as the remaining classes and activities either had no way to be unit tested or were classes which were never called or used.

Summary

For the unit tests, we reached 100% branch coverage on GameStatistics and FirstLaunchManager. For many of the classes we were unable to achieve that goal with unit testing alone for many reasons. The main reason being that this app is heavily coupled with UI and so unit Tests were only performed on parts of the code which could be tested via unit tests. The other reason being the code implemented bad coding practices which are not ideal for unit testing, such as using global variables and functions returning void. Thus there are many parameters which we were unable to manipulate in order to reach branch coverage for the unit tests. Due to the nature of this app, the GUI coverage report should compensate for many parts of the code which unit test coverage was unable to reach. The activities especially are more UI based thus explaining their low coverage. For the GameActivity and StatsActivity specifically there were some parts that were unreachable due to to the type of exception in the catch clause that we were unable to recreate. For instance, expecting an InvalidClassException or SecurityException when reading a file or deleting a file.

GameState
For the GameState objects this goal was not reached because the class contains a constructor which is never used and we refrained from testing code that are never called.

Element
This goal was not reached because the element class contains an if statement in the SetColors function which is out of our control. We cannot control the type of object the background of Element is.

Gestures
The onFling function is a complex swipe which we struggled to recreate through unit testing and thus this was left for GUI testing. This function also takes up the majority of the code in the class, explaining the low coverage.

GameActivity
This activity contained many global variables most of which were private and many functions which returned void. So there were limits as to what could be tested via unit testing as well the variables which could be manipulated to achieve branch coverage. Activities are heavily coupled with UI and so many parts of it could only be tested through GUI.

StatsActivity and **MainActivity**
Heavily coupled with UI and so unit tests only performed on code which can be unit tested

New Failures

One of the constructors for GameState, GameState(Element[][] e, Element[][] e2), contained two parameters. Note that this constructor has no documentation and provides no documentations on how the inputs should be or what behavior to expect. Therefore I received ArrayIndexOutOfBoundException on some of the attempted inputs.