Tests and Test Plan for Illo

# Test Cases

# Use Case Testing

# Written Test 1

**Title: Confirm Permission Changes**

Actors: User

Requirements: Illo is in Permission Management

**Main Scenario:**

1. User selects “Save Changes” Button

2. Illo Saves the permission changes and returns to the Settings screen

**Alternatives:**

1a. User selects the “Cancel Changes” Button

1a1. Illo does not save the permission changes and returns to the Settings Screen

**Test Situations:**

1) User confirms permission changes

2) User cancels the permission changes

**Test Coverage:**

Base: number of main and alternative scenarios: 2

Test situations cover all 2 cases

100% coverage of use case

# Written Test 2

**Title: Start Internal Timer**

Actors: User

Requirements: Illo is on the Timer Screen

**Main Scenario:**

1. User selects the “Play” button.

2. Illo starts the current workout interval.

3. Play Button is Replaced with a Pause button.

**Test Situations:**

1. User Presses the “Play” Button

**Test Coverage:**

Base: number of main and alternative scenarios: 1

100% coverage of use case

# Written Test 3

**Title: Exit Permission Management**

Actors: User

Requirements: Illo is in Permission Management

Main Scenario:

1. User Selects the “Exit” Button.

2. Illo returns to the Setting Screen

**Alternatives:**

1a. User has unsaved changes and Selects the “Exit” Button.

1a1. Illo displays a “Confirm Changes” popup.

**Test Situations:**

1. User exits the menu with no unsaved changes.
2. User exits the menu with unsaved changes.

**Test Coverage:**

Base: number of main and alternative scenarios: 2

Test situations cover all 2 cases

100% coverage of use case

# Written Test 4

**Title: Toggle Permission**

Actors: User

Requirements: Illo is in Permission Management

Main Scenario:

1. User toggles an enabled permission

2. Illo disables the selected permission

**Alternatives:**

1a. User toggles a disabled permission

1a1. Illo enables the selected permission

**Test Situations:**

1. Enabled permission toggle
2. Disabled permission toggle

**Test Coverage:**

Base: number of main and alternative scenarios: 2

Test situations cover all 2 cases

100% coverage of use case

# Written Test 5

**Title: Rename Exercise from Activity Source**

Actors: User  
Requirements: Illo is in the Activity Source Configuration Screen

**Main Scenario:**

1. User selects name of the Activity Source

2. Illo prompts the user to enter text

3. User enters text and confirms

4. Illo renames the Activity Source to the text entered

5. Illo returns to the Activity Source Configuration Screen

**Alternatives:**

1a. User exits the screen

1a1. Illo returns to the Activity Sources screen

3a. User cancels

3a1. Illo does not rename the Activity Source

**Test Situations:**

1. User renames exercise
2. User exits the Activity Source Configuration Screen
3. User Cancels the renaming

**Test Coverage:**

Base: number of main and alternative scenarios: 3

Test situations cover all 3 cases

100% coverage of use case

# Written Test 6

**Title: Set Activity Source**

Actors: User

Requirements: Illo is in the Configuration Screen

**Main Scenario:**

1. User selects the Activity Source field

2. Illo swaps to Activity Sources screen

3. User selects an Activity Source

4. Illo presents list of exercises in selected Activity Source

5. User selects confirm for selected Exercise Set

6. Illo saves the selected Activity Source, and sets the timer’s nextExercise() method to point to thenext() method in the Activity Source

7. Illo closes dialogue and returns to the Illo Configuration Screen

**Alternatives:**

3a. User exits screen

3a1. Illo returns to the Configuration Screen

5a. User selects cancel

5a1. Illo exits the confirmation menu, returning to the Activity Sources screen.

**Test Situations:**

1. User sets the activity source
2. User exits the screen
3. User cancels setting the activity source

**Test Coverage:**

Base: number of main and alternative scenarios: 3

Test situations cover all 3 cases

100% coverage of use case

# Unit Testing

# **ActivitySource.addExercise() Test Class, Method**: ActivitySource, addExercise() **Description**: Tests the ability to add Exercises to an ActivitySource. Also checks the edge case of adding Exercises with names that are already contained by the ActivitySource. The goal is to ensure both that users may be able to add Exercises to their own ActivitySources and that Illo may build the default ActivitySources (“Freeweight Exercises”, “Muscular Stretches”, and “Hand Stretches”) at startup.

@Test  
**public** **void** testAddExercise(){  
 testAS.addExercise(**new** Exercise("Exercise"+TEST\_SIZE, **null**, **null**));  
 assertTrue("Add Exercise "+TEST\_SIZE, testAS.exerciseBank.containsKey("Exercise"+TEST\_SIZE));  
  
 *// repeat names* testAS.addExercise(**new** Exercise("Exercise"+TEST\_SIZE, **null**, **null**));  
 assertTrue("Add Exercise "+TEST\_SIZE+"\_1", testAS.exerciseBank.containsKey("Exercise"+TEST\_SIZE+"\_1"));  
  
 testAS.addExercise(**new** Exercise("Exercise"+TEST\_SIZE, **null**, **null**));  
 assertTrue("Add Exercise "+TEST\_SIZE+"\_2", testAS.exerciseBank.containsKey("Exercise"+TEST\_SIZE+"\_2"));  
}

# ActivitySource.removeExercise() Test

@Test  
**public** **void** testRemoveExercise(){  
 testAS.removeExercise("Exercise0");  
 assertFalse("Remove Exercise0", testAS.exerciseBank.containsKey("Exercise0"));  
  
 **int** currentSize = testAS.size();  
 testAS.removeExercise("Exercise0");  
 assertEquals(currentSize, testAS.size()); *// size shouldnt change.*  
 *// reset* testAS.addExercise(**new** Exercise("Exercise0", **null**, **null**));  
 testAS.reorderExercise("Exercise0", 0);  
}

# **Class, Method**: ActivitySource, removeExercise() **Description**: Tests the ability to remove Exercises from an ActivitySource. Also checks the edge case of removing Exercises that are not present in the ActivitySource. The goal is to ensure that users are (eventually) able to edit their ActivitySources by removing undesired Exercises.

|  |  |
| --- | --- |
| **Test Inputs** | **Expected Outcome** |
| Remove an Exercise from an AS that is contained by the AS | Exercise is removed from the AS |
| Remove an Exercise an AS that is not contained by the AS | No Exercise is removed from the AS (checked by the size). |

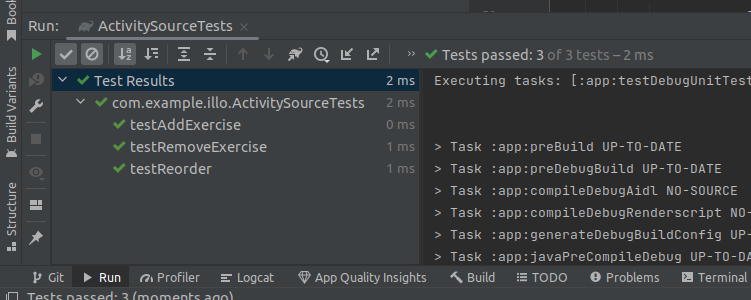
# ActivitySource.reorderExercise() Test

@Test  
**public** **void** testReorder(){  
 ArrayList<Exercise> al = **new** ArrayList<>();  
 **for**(String k : testAS.exerciseBank.keySet()){  
 al.add(testAS.exerciseBank.get(k));  
 }  
  
 Exercise popped = al.remove(10);  
 al.add(0, popped);  
 testAS.reorderExercise("Exercise10", 0);  
 *// check order* **int** al\_index = 0;  
 **for**(String k : testAS.exerciseBank.keySet()){  
 assertEquals(al.get(al\_index), testAS.exerciseBank.get(k));  
 al\_index++;  
 }  
  
 *// check move from beginning of AS to middle* popped = al.remove(0);  
 al.add(10, popped);  
 testAS.reorderExercise("Exercise10", 10);  
 al\_index = 0;  
 **for**(String k : testAS.exerciseBank.keySet()){  
 assertEquals(al.get(al\_index), testAS.exerciseBank.get(k));  
 al\_index++;  
 }  
  
 *// check move to index outside of list (append to end)* popped = al.remove(15);  
 al.add(popped);  
 testAS.reorderExercise("Exercise15", TEST\_SIZE+20);  
 al\_index = 0;  
 **for**(String k : testAS.exerciseBank.keySet()){  
 assertEquals(al.get(al\_index), testAS.exerciseBank.get(k));  
 al\_index++;  
 }  
  
 *// check move from end of list to middle* popped = al.remove(al.size()-1);  
 al.add(15, popped);  
 testAS.reorderExercise("Exercise15", 15);  
 al\_index = 0;  
 **for**(String k : testAS.exerciseBank.keySet()){  
 assertEquals(al.get(al\_index), testAS.exerciseBank.get(k));  
 al\_index++;  
 }  
  
 *// check move to same index* testAS.reorderExercise("Exercise15", 15);  
 al\_index = 0;  
 **for**(String k : testAS.exerciseBank.keySet()){  
 assertEquals(al.get(al\_index), testAS.exerciseBank.get(k));  
 al\_index++;  
 }  
}

# **Class, Method**: ActivitySource, reorderExercise() **Description**: Tests the ability to shift an Exercise from one position in an AS to a new one. Other elements should shift around them. That objects at an equal to or greater index all shift up by one to make room for the moved Exercise. An ArrayList mirrors the desired behavior well – so that is what we test against. We also handle edge cases – indices outside of the “legal” range. Our goal is to allow great customization of ActivitySources, particularly Workouts. Workouts have a consistent order every usage, so being able to reorder the Exercises is critical to the UX. Users may also want to organize their other ActivitySources in particular ways, for organizational purposes.

|  |  |
| --- | --- |
| **Test Inputs** | **Expected Outcome** |
| Change an Exercise’s order | Exercise’s order is shifted in the AS. That is, if the AS is a Workout, this is the position in the order that the specified Exercise will appear. The other elements shift order around the moved Exercise. |
| Move an Exercise from the beginning/end of the AS to the middle | Edge case. Should behave as specified above |
| Move an Exercise to a negative index | Exercise gets moved to the first index of the AS |
| Move an Exercise to an index greater than the number of Exercises present. | Exercise gets moved to the last index of the AS |

The above three tests are all contained in **ActivitySourceTests** in the test directory of the Android Studio project. Running these tests with Android Studio’s “Run” tab gives the following output when passed (check the bottom left).



# Workout.incrementAtIndex() Test

# **Class, Method**: Workout, incrementAtIndex() **Description**: The design of Workouts is bases on the idea of order. Ensuring that that order is maintained properly is paramount. The goal of this test is to ensure that Workouts are always at the correct index – “atIndex”. Particularly, that when atIndex comes to exceed the size of the Workout, it wraps back to the beginning of the Workout here. incrementAtIndex() has parameter overrides. It can take no arguments and increment by 1, or take an int argument and increment by that many. The version with no arguments calls the single-argument version. They are tested together here.

@Test  
**public** **void** testIncrementAtIndex(){  
 **int** s = testWO.size();  
  
 **int** current = testWO.getAtIndex();  
 testWO.incrementAtIndex(-1);  
 assertEquals(current, testWO.getAtIndex());  
  
 **for**(**int** i = 0; i<10; i++){  
 testWO.incrementAtIndex();  
 }  
 assertEquals((current+10)%s, testWO.getAtIndex());  
  
 **int** big\_change = (**int**) Math.floor(TEST\_SIZE\*111.25);  
 testWO.incrementAtIndex(big\_change);  
 assertEquals((current+10+big\_change)%s, testWO.getAtIndex());  
  
 testWO.setAtIndex(0);  
 testWO.incrementAtIndex(s);  
 assertEquals(0, testWO.getAtIndex());  
}

|  |  |
| --- | --- |
| **Test Inputs** | **Expected Outcome** |
| Increment atIndex by 10 | At index should now be 10 more than it was before. Or, if this exceeds the size of the Workout, it should be original atIndex plus the change mod the size of the Workout. |
| Increment by a negative number | Edge case. atIndex does not change. |
| Increment by a number many times larger than the size of the | atIndex should wrap as expected. atIndex should become the original atIndex + the big change mod the size of the Workout. |
| Increment by the size of the Workout. | atIndex should stay the same. |

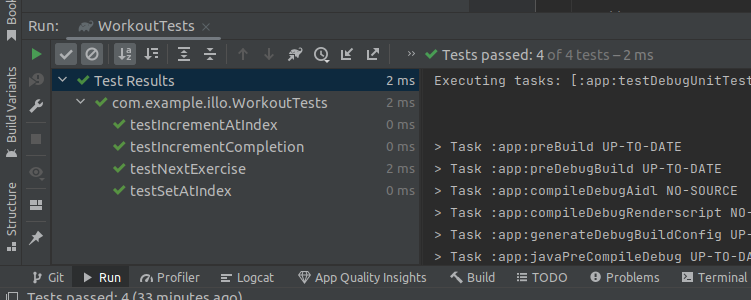
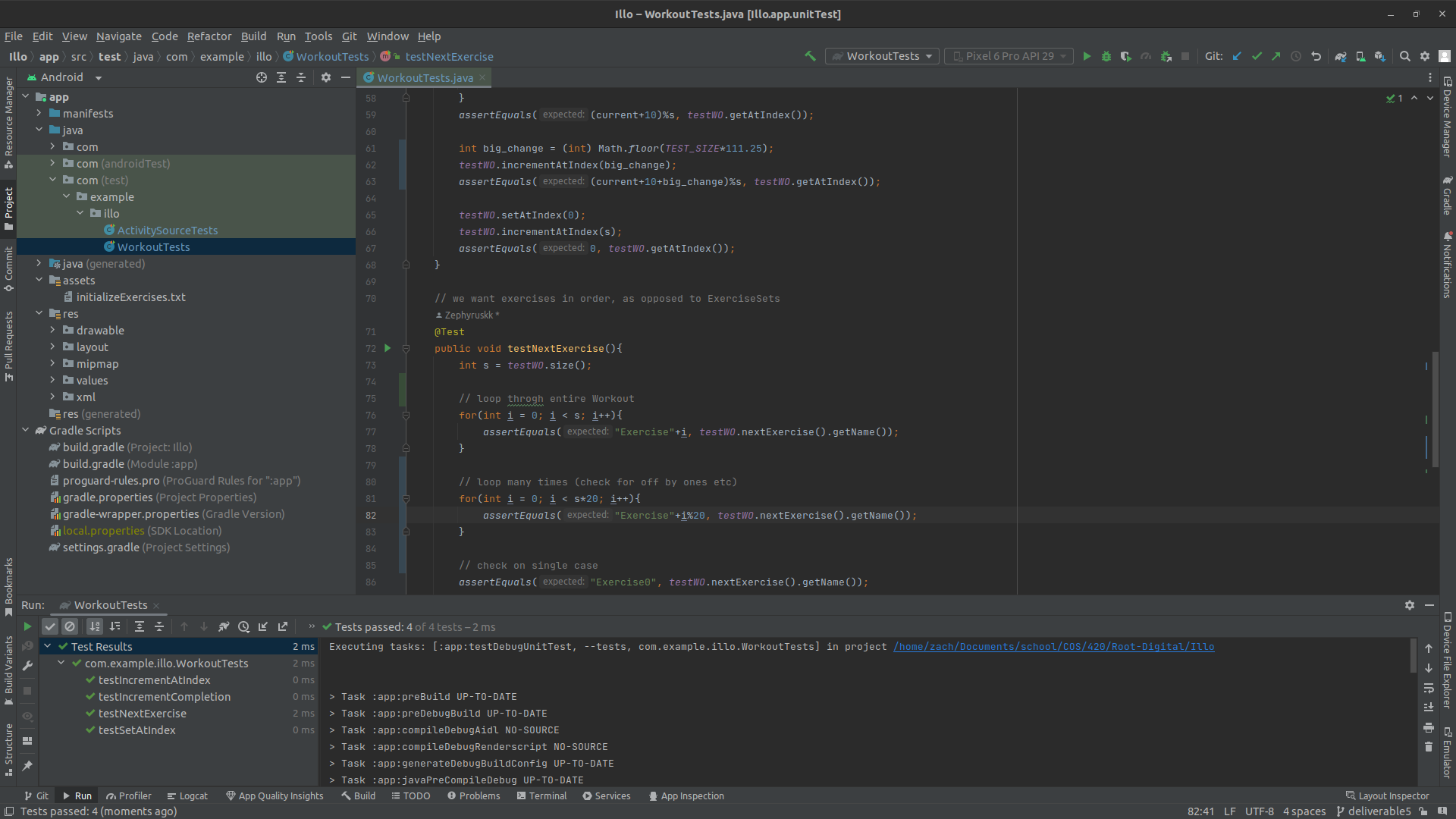
# Workout.nextExercise() Test

# **Class, Method**: Workout, nextExercise() **Description**: The design of Workouts is bases on the idea of order. Ensuring that that order is maintained properly is paramount. No matter how many times a Workout is looped through, it should produce the same order. That is, each Exercise should always correspond to a particular index or number. For this test, the order is inferred via the construction of the test Workout. Check the @BeforeClass instantiation in WorkoutTests.java to see more.

**public** **void** testNextExercise(){  
 **int** s = testWO.size();  
  
 *// loop throgh entire Workout* **for**(**int** i = 0; i < s; i++){  
 assertEquals("Exercise"+i, testWO.nextExercise().getName());  
 }  
  
 *// loop many times (check for off by ones etc)* **for**(**int** i = 0; i < s\*20; i++){  
 assertEquals("Exercise"+i%20, testWO.nextExercise().getName());  
 }  
  
 *// check on single case* assertEquals("Exercise0", testWO.nextExercise().getName());  
  
 *// reset* testWO.setAtIndex(0);  
}

|  |  |
| --- | --- |
| **Test Inputs** | **Expected Outcome** |
| Traverse the Workout step by step | Each Exercise should be in the order it appears in AS |
| Traverse the Workout many times | Still, order is maintained. When a Workout has been traversed, it returns to the zero’th index. |
| Traverse the Workout just once, from position i. | The Exercise at i should be returned. |

The above three tests are all contained in **WorkoutTests** in the test directory of the Android Studio project. Running these tests with Android Studio’s “Run” tab gives the following output when passed, shown on the next page. A full screen screenshot is also included for added context.



# Alarm Activates

|  |  |
| --- | --- |
| Test Inputs | Expected Outcome |
| Timer Running | No Alarm |
| Timer Hits 0 and resets | Alarm runs once then stops |
| Timer is 0 | Alarm runs once then stops |

This test will make sure that the sound alarm will not interfere with the user’s manual input of timer duration.

To create this test:

1. separate test timer with varying lengths, (100sec, 10sec, 0sec)

2. Assertions that the alarm plays when the timer hits 0,

3. Assertion the alarm does not continue to run after once.

This test will test the onFinish() Spinner override method in the mainActivity class of the app.

# UI color Swapping Test

|  |  |
| --- | --- |
| **Test Inputs** | **Expected Outcome** |
| Default | All windows are white bg, blue border |
| Green | All windows are light blue bg, green border |
| Red | All windows are light red bg, red border |

This test will make sure that all UI windows are affected by changing the color options.  
1. Each window will need to be checked for correct color  
2. Each window will need to be checked that they stay the selected color when app is closed   
This test will run using the window swapping functions (yet to be written) in the mainActivity class of the app as well as the color saving controller.

# Username Input Test

|  |  |
| --- | --- |
| Valid | Invalid |
| 1 < Length < 32 | Length = 0, Length> 32 |
| Includes Characters | No Special Characters (Except Space) |
| Can Include Numbers | No Emojis |
|  | Crass Words |

This test will make sure that user input of username will be viable with motivational text. This test will also do a rudimentary check for crass/inappropriate words.

1. Input will need to be checked that it is saved to the app

2. Input must have a length of at least 1 and at most 32.

3. Input must not use special characters

4. Input must not use Emojis

5. Input will be checked for inappropriate words and reject if found.

This test will test the username validation method (yet to be written) in the User class.

# Streak Test

|  |  |
| --- | --- |
| Test Inputs | Expected Outcome |
| 24 hours pass | Reset Streak |
| An Activity Source is completed | Streak +1 |
| An ActivitySouce that has already been completed in the past 24 hours is completed again. | No change |

This test will check if the user use streak is accurate to usage daily.

1. Check last time the app was opened, if 24 hours streak should be 0

2. Check if last time streak was increased, if 24 hours streak should be 0

3. Check if last time streak was increased, if less than 24 hours + set complete Streak +1

4. Check if Streak has been increased in 24 hours, + ActivitySource completed, Streak + 0

This test will test methods that control the streak variable (yet to be written) in the User class.

# Feedback Test

|  |  |
| --- | --- |
| Test Inputs | Expected Outcome |
| Empty Report | Will not Accept |
| Rating | Save Ratings to rating Database |
| Rating + Text | Save Ratings to rating Database & Text to Text Database |

This test will make sure that user Feedback will be saved accurately in an external Database.

1. If the report is empty, the system should not allow submission

2. If the report is just a rating, check if it is sent to the correct rating Database

3. If the report is a rating and text, check if it is sent to the correct rating and text databases.

This test will test the Feedback Controller class (yet to be written).