Testing Document

EECS 2311 Z – *Software Development Project*

Group 15: Aya Abu Allan, Gianluca Corvinelli, Mark Savin

Table of Contents

Section 1 - General Information

Section 1.1: *Introduction*

Test cases were developed for the **SceneAA**, **ScenarioAA**, and **SceneAAWriter** classes. As a disclaimer, we have created test cases for getter and setter methods, but we will not go into detail about how these tests were derived unless these getters and setters do more than just get and set (i.e., they are complex methods). The purpose of unit testing is to test the behaviour of the code in a meaningful way, and getters and setters are merely a means to an end. We do go into detail about how tests were derived for complex methods. An example of a complex method would be a method that can clear the pins of a braille cell. It is important to test these kinds of methods because we want to ensure that they are behaving as expected. One last thing worth mentioning is that in writing test cases we were able to find bugs in our code which leads to improvements in the design and functionality of our code.

Section 2 - Test Cases

Section 2.1: *Test Cases Used*

SceneAATest

- public void test01_getAudioNameOption1()
- public void test02_ getInteractionAudioNameAtIndex()
- public void test03_audioOption1Exists()
- public void test04_getSceneName()
- public void test05_getButtons()
- public void test06_getNOB()
- public void test07_setPinsScene() throws InterruptedException
- public void test08_setPinsScene()
- public void test09_getPinsAsBoolean ()
- public void test10_clearPins()
- public void test11_setInterationTextInput()
- public void test12_setInteractionPreset()
- public void test13_setQuestion()

ScenarioAATest

- public void test01_getNOB()
- public void test02_getNOC()
- public void test03_getScenarioName()
- public void test04_getCurrentScene()
- public void test05_getScenario()
- public void test06_setNumberOfButtons()
- public void test07_setNumberOfCells()
- public void test08_setCurrentSceneIndex()
- public void test09_findSceneIndex()
- public void test10_newCurrentScene()

SceneAAWriterTest

- public void test01_writeQuestion()
- public void test02_writePins()
- public void test03_writeAudioOption1()
- public void test04_writeInteractions()
- public void test05_writeInteractions()
- public void test06_writeInteractions()
- public void test07_writeInteractions()
- public void test08_writeInteractions()
- public void test09_writeInteractions()

- public void test10_writeInteractions()
- public void test11_getSceneAsTxt()
- public void test12_getSceneAsTxt()
- public void test13_getSceneAsTxt()

Section 2.2 - How Test Cases Were Derived and Implemented

Section 2.2.1: *SceneAATest*

test01, test03, test04, test05, test06, test11, test12, and test13 are test cases for the getter and setter methods of the **SceneAA** class and will not be the focus of our discussion as mentioned in *Section 1.1 - Introduction*.

public void test02_getInteractionAudioNameAtIndex():

To explain what this test does, it is important to understand what an interaction button is. An interaction button is a means of dynamically communicating with students. The teacher may decide to create a scene with four buttons, each of those buttons will have some type of interaction. Perhaps the teacher has decided that they would like to ask the students what letter is being displayed by the braille cell, and so each button would contain some type of interaction. There are five types of interactions: "*No Interaction*", "*Play Correct Audio Clip*", "*Play Wrong Audio Clip*", "*Repeat Question Text",* and "*Skip to Next Scene*". If for example, the teacher wanted button three to be the button selected to provide the correct answer they would select the "*Play Correct Audio Clip"* interaction when the student presses that button to signify they got the correct answer. Likewise, if the student selected the wrong answer then you would select the "*Play Wrong Audio Clip"* interaction to signify to the student that they answered incorrectly. The number of interaction buttons is directly correlated to the number of buttons that the scene object creates. If there are four buttons in a scene, then there will be four interactions, one interaction for each button.

This test case tests for two things. First, it tests that a scene can only have as many buttons as the scene object has created and that the scene must have at least one button. The second test, tests to verify that each button has an interaction.

We start the test by creating a **SceneAA** object with five buttons and one braille cell. This part of the test handles boundary conditions. What happens when we have too many or too few buttons? We used a try catch block and wanted to get button six. We chose button six knowing this should fail because the scene object only has five buttons. If the test did not fail, we wrote a command, "*fail("Should have thrown IndexOutOfBoundsException")"* indicating that something went wrong and the test should have failed. If, however, the test did fail we would reach the catch block which would then assert an, "*IndexOutOfBoundsException"* confirming that our test did indeed fail as we had expected. If an "*IndexOutOfBoundsException"* was not thrown our other catch block would assert that some other Exception or Error was thrown. Either way, if we reached a catch block we knew we were handling our boundary condition. We also handled the case when there was less than one button. To test this is the same as testing to see whether we can get button six. It is the same process, but we are testing a different boundary condition.

For the second part of this test, we created a new **SceneAA** object with one button and one braille cell. We ignore the number of braille cells this scene has because we are only concerned with the number of buttons and their interactions. We created a constant string variable called, "*AUDIO_NAME"* and stored in it the string, *"Audio1"*. We then set button one

to have the name, *"Audio1"* and then asserted that the name stored at button one was indeed *"Audio1"*. This is to prove that each button can have an interaction associated with it.

public void test07_setPinsScene() throws InterruptedException:

This test verifies that the pins in our current scene are being set correctly. This test works by creating a new **SceneAA** object called "*pinScene"* and designating the first braille cell to display the letter "a" so the pin state of this braille cell is "10000000". To test if this is true we initialized a variable called *"brailleCell"* to be of type "*BrailleCell"* and referred it to the first braille cell in the scene by writing, *"pinScene.getBrailleCells().get(0)"*. The *get(0)* allows us to refer to the first braille cell in the scene. We then had to assert each pin state of the cell to be true or false, true meaning "1" and false meaning "0". To have the braille cell output "10000000" we asserted the first pin of "*brailleCell"* to be true and the remaining pin states to be false, which results in a string of "10000000". Thus, we are confident that the braille cell is displaying the letter "a".

public void test08_setPinsScene():

This test is similar to the previous test, test07. The difference is that this test uses a list of Booleans to set the pin states of the braille cell. We initially created an empty list of Booleans and then added true or false to the list until we had eight pin states, because a braille cell requires eight pins to display a character. We used the command, "*listAsBoolean.add(true)"* so the first value in the list is "1" and the remaining values were set to "0" by writing the command, "*listAsBoolean.add(false).*" The braille cell would then have a pin state of "10000000" indicating that the letter "a" is displayed.

public void test09_getPinsAsBoolean() throws InterruptedException:

This test verifies that we can get the pin state of a braille cell. We created a **SceneAA** object and we also created a *BrailleCell* object and name it "*bCell1*". We set "*bCell1*" to display the letter "a" by writing the command, "*bCell1.displayCharacter('a')*". We created a List<Boolean> and called it "*listAsBoolean*". We then wrote "*listAsBoolean.add(true)*" followed by "*listAsBoolean.add(false)*" repeated seven times which would create a pin state of "10000000". Next, we asserted that "*listAsBoolean*" was equal to the character displayed by the "*bCell1*" object. This test passed because the pin state for the character "a" is "10000000" which is precisely the state of our "*listAsBoolean*" object.

public void test10_clearPins():

This test verifies that the pin states of a braille cell can be cleared (all pin states are set to zero, "00000000"). We begin writing the test by creating a new **SceneAA** object with three braille cells and one button. This test ignores the number of buttons in the scene because we are only concerned with clearing the pins of each braille cell. We chose to get the first braille cell and arbitrarily set its pin state to "10101010" by using the following command, "*scene.getBrailleCells().get(0).setPins("10101010")*". The "0" in *get(0)* refers to the first braille cell of the scene. We then had to assert each pin state of the cell to be true or false, true meaning "1" and false meaning "0". To have the braille cell output "10101010" we asserted the first pin

of the braille cell to be true, the second to be false and so on until we had a pin state of "10101010".

Now that we have set the pin state of the first braille cell, it is time to clear it. We proceeded to use the "*scene.clearPins(0)"* command to specify that we wanted to clear all the pins of the first braille cell. We had to assert that every pin state was false, showing that the pins have been cleared, and thus resulting in a pin state of "00000000". We repeated this process two more times to prove that we can clear the second and third braille cells of the scene. The only difference is that to refer to the second and third braille cells of a scene we used the "*scene.getBrailleCells().get(1)"* and "*scene.getBrailleCells().get(2)"* commands respectively.

Section 2.2.2: *ScenarioAATest*

test01, test02, test03, test06, test07, and test08 are test cases written for the getter and setter methods of the **ScenarioAA** Class and will not be the focus of our discussion as mentioned in *Section 1.1 - Introduction*.

public void test04_getCurrentScene():

This test verifies that we can work in the scene we have selected from a list of scenes. To illustrate this let us examine *Teaching Scenario Builder*. In *Teaching Scenario Builder* there is a list of scenes that are displayed on the right-hand side of the scenario and new scenes are constantly added to the list whenever a new scene is created.

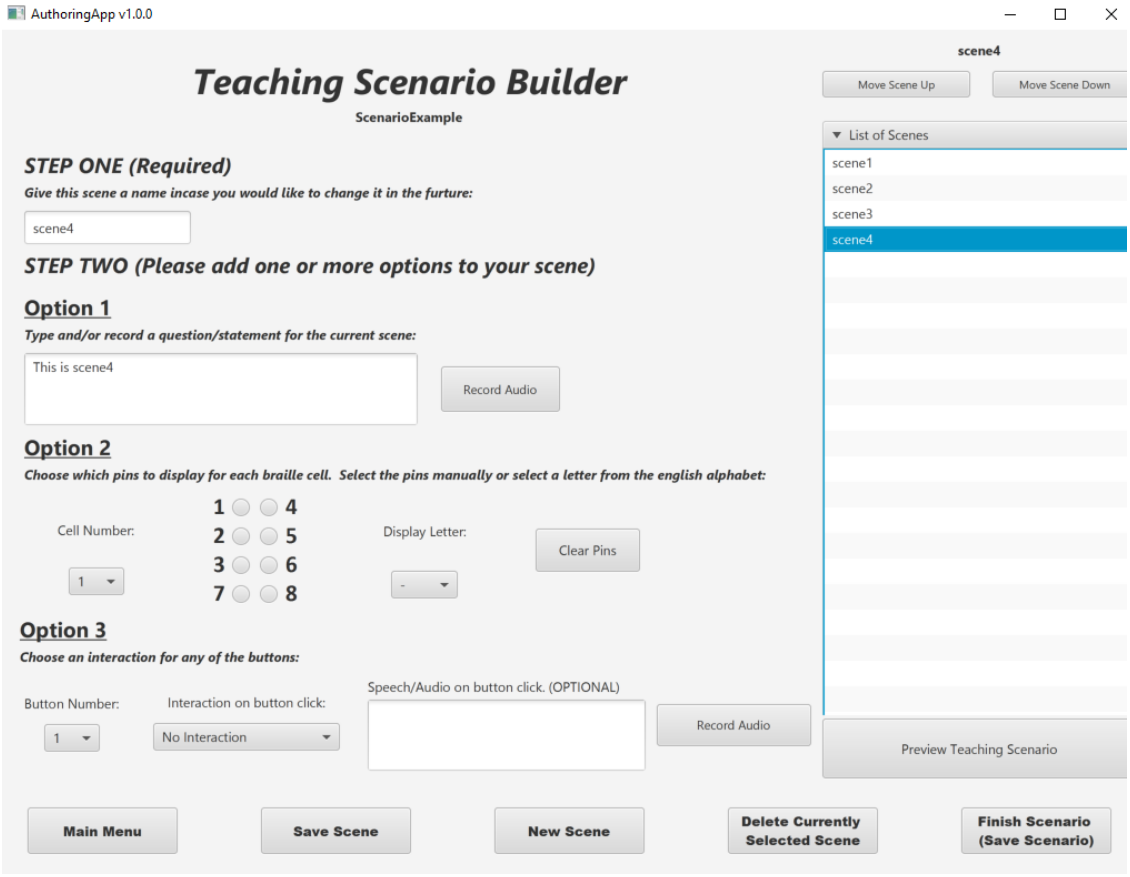

Figure 1.0: Teaching Scenario Builder

As shown from the image above, if we select *scene4* from the list we should now be working in *scene4*.

In this test we created a new **ScenarioAA** object called "s*"* that has been initialized to have an empty string, one button, and one braille cell. We also created a new **SceneAA** object with one button and one braille cell and called it "*scene2"*. We then got the current scene of the scenario by using the command, "*s.getCurrentScene()"* and stored this scene in a variable called "*originalScene"* by writing "*SceneAA originalScene = s.getCurrentScene()"*. We then added

"*scene2"* to the scenario object. At this time the scenario should have a list that contains "*originalScene"* and "*scene2"*. We then set the current scene at index zero (the scene at the top of the list) by writing "*s.setCurrentScene(0)"* and asserted that the current scene is "*originalScene"* by writing "a*ssertEquals(originalScene, s.getCurrentScene()).”* The test passed but just to be sure our code was working we created two more **SceneAA** objects called "*scene3"* and "*scene4"* and tested to see if we could get a scene that was not at the top of the list. We set the current scene to be at index two, meaning we expect to get back the third scene in the list, "*scene3"* (the list is 0 index based). We asserted that the new current scene was "*scene3"*, and our test passed, just as we expected. We have demonstrated that our code successfully selects the scene the user wants, which enables the user to go back to a scene and edit it if they so choose.

public void test10_newCurrentScene():

As done in the previous test, test04, we want to verify that this test can get a scene, but what is unique about this test is that which ever scene we get, we establish that this is now the **new** current scene of the scenario.

We start by creating a **ScenarioAA** object called "*scenario"* with one button and one braille cell. We then create two **SceneAA** objects called "*Scene1"* and "*Scene2"* and add these scenes to the **ScenarioAA** object by using the commands "*scenario.addScene(Scene1)"* and "*scenario.addScene(Scene2)"*. When a scenario is created, by default, an empty scene is created. This default scene is the current or initial scene when we enter the *Teaching Scenario Builder* interface. We then set "*Scene1"* to be the new current scene of the **ScenarioAA** object by writing "*scenario.newCurrentScene(Scene1)"* and then assert that the current scene we should get is "*Scene1"*, thus proving we can set the current scene to be any scene we choose.

public void test05_getScenario():

This test verifies that we can get the list of scenes that are created inside the Scenario object. We begin by creating a **ScenarioAA** object called "*s"* as well creating two **SceneAA** objects called "*scene1"* and "*scene2"*. We then add these two scenes to the **ScenarioAA** object by writing "*s.addScene(scene1)"* and "*s.addScene(scene2)"*. When a scenario is created, by default, an empty scene is created. This default scene is the current or initial scene when we enter the *Teaching Scenario Builder* interface. Therefore, there are three scenes in the **ScenarioAA** object; the two we just created, "*scene1"* and "*scene2*", and the default scene that was created upon entering the scenario builder.

Next, we created an empty list of scenes and called it "*listOfScenes"*. In this list we add the initial or current scene of the scenario object as well as the "*scene1"* and "*scene2"* objects to "*listOfScenes*". We could then assert that the scenes in our **ScenarioAA** object contained the same scenes that our "*listOfScenes"* contained, thus confirming that we can get a list of scenes from a scenario.

public void test09_findSceneIndex():

This test determines at which position is each scene stored in the list. In other words, is the scene we want located at the top of the list, somewhere in the middle of the list, or at the bottom?

We start by creating a new **ScenarioAA** object called "*scenario"* with three buttons and one braille cell. When the scenario was created, a default scene was created. In our test case we referred to this default scene as *"s0"*. We then created four **SceneAA** objects and called them *"s1"*, *"s2"*, *"s3"*, and *"s4"* respectively and then added each of these scenes to the scenario. Lastly, we asserted that *"s0"* at index 0 is true, *"s1"* at index 1 is true and so on and so forth. We know the index of "s1" is at index 1 in the list because it was the second element added to the list, so we could expect the test to be true when we made that assertion. The same logic applies for the remaining **SceneAA** objects. This test demonstrates that we can locate where each scene is in the list.

Section 2.2.3: *SceneAAWriterTest*

public void test01_writeQuestion():

This test verifies that the instructor can write any sentence, statement, or question within the current scene. They may want to write a question to test the student's knowledge, or perhaps add a statement to teach a new concept. This is accomplished in the *Teaching Scenario Builder*.

We began writing the test by creating a **SceneAAWriter** object called "*sceneWriter"*. The object takes two parameters of type **SceneAA** and a variable of type Boolean, so we created a new **SceneAA** object called *"scene"* and a Boolean variable called "*isFirstScene"* and initialized its value to true and then we passed in these two variables as the parameters for "*sceneWriter"*. Next, we created a string variable called "*Question"* and stored in it the string *"Is this a question?"*. We then set the scene object to *"Question"* so now our scene contains the string *"Is this a question?"*. Next, we added the same string to "s*ceneWriter"* and lastly asserted that the string contained in our scene object was indeed the string "*Is this a question?*".

We did two more tests to determine what would happen if we had a null string and an empty string. In either of these cases we would not want anything to be written in the scene. Testing for the null and the empty string is the same process as described in the previous paragraph. Now we know that if we receive an empty or null string then we will not write anything into the scene.

public void test02_writePins():

This test is like test02 and test07 described in *Section 2.2.1 – SceneAATest*. This test requires us to be able to write the pin states that a braille cell displays which is precisely what test07 does. This test also checks for boundary conditions. For example, how do we write the pin state for braille cell number six if our **SceneAA** object only created five braille cells? This exact same condition was tested in test02. Therefore, this test case will not be elaborated on because it is the exact same process as described in test02 and test07 in *Section 2.2.1 – SceneAATest*.

private List<String> getExpectedSceneBeginning(boolean first):

This is not a test case, this is a method that was written for tests 3-13. To provide more context for why this method is relevant it is important to realize that in the **SceneAAWriter** class there is a method called "*public List<String> getSceneAsTxt()"*. It is in this method where all the "sounds" are present. When a new **SceneAAWriter** object has been created there is a default "sound". Every newly create **SceneAAWriter** object must have these "sounds" as default. So, rather than continually creating the **SceneAAWriter** object for test cases 3-13, we created one method, and then test 3-13 would call this method to have the necessary default values. This is nothing more than efficient programming, it saves time and code is not redundant.

Tests Cases 3 to 13:

Tests cases 3 to 13 are very similar. While they do test different parts of the application, the tests are implemented the same way. For this reason, we have chosen to explain all these test cases together and will distinguish any unique differences.

Generalization of Test Cases 3 to 13

We created a **SceneAAWriter** object that takes in two parameters, a **SceneAA** object and Boolean variable, thus we also created a **SceneAA** object and a Boolean variable for these test cases. Also, in each test case we create a List<String> variable called "*expected*" and assign to it "*getExpectedSceneBeginning(boolean)"* which contain all the initial or default value "sounds" a recorded audio message would contain. From there, depending on which interaction we wanted to test, we would add additional "sounds" to the expected list, and this would test that the desired interaction would be set.

Differences between Test Cases 3 to 13

Test 3, *public void test03_writeAudioOption1* tests to determine if the user can record audio.

Tests 4 to 10 are all tests that are written for the "*public void writeInteractions()"* method in the **SceneAAWriter** class. These tests determine if the interaction has been set, if there is text written for the interaction, and if the interaction has an audio name associated with it. The three interactions are; *"Play Correct Audio Clip", "Play Wrong Audio Clip"*, or *"Skip to Next Scene"*.

Tests 11 to 13 are written for the "*public List<String> getSceneAsTxt()"* method in the **SceneAAWriter** class. These tests determine if the *"Repeat Question Text"* interaction has been set or not and test 12 determines what happens when there is more than one scene in the Scenario list.

Section 3 - Test Case Sufficiency

Section 3.1: *JUnit Test Cases*

SceneAATest Coverage

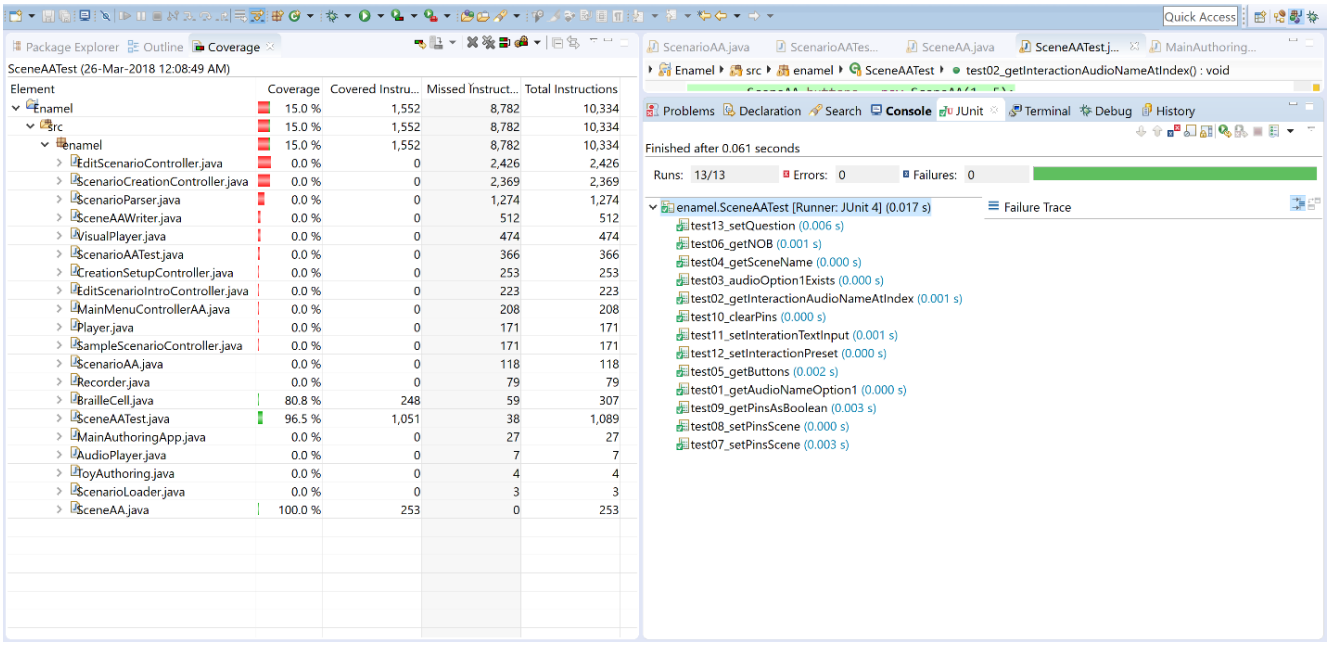

Figure 2.0: SceneAATest Coverage

ScenarioAATest Coverage

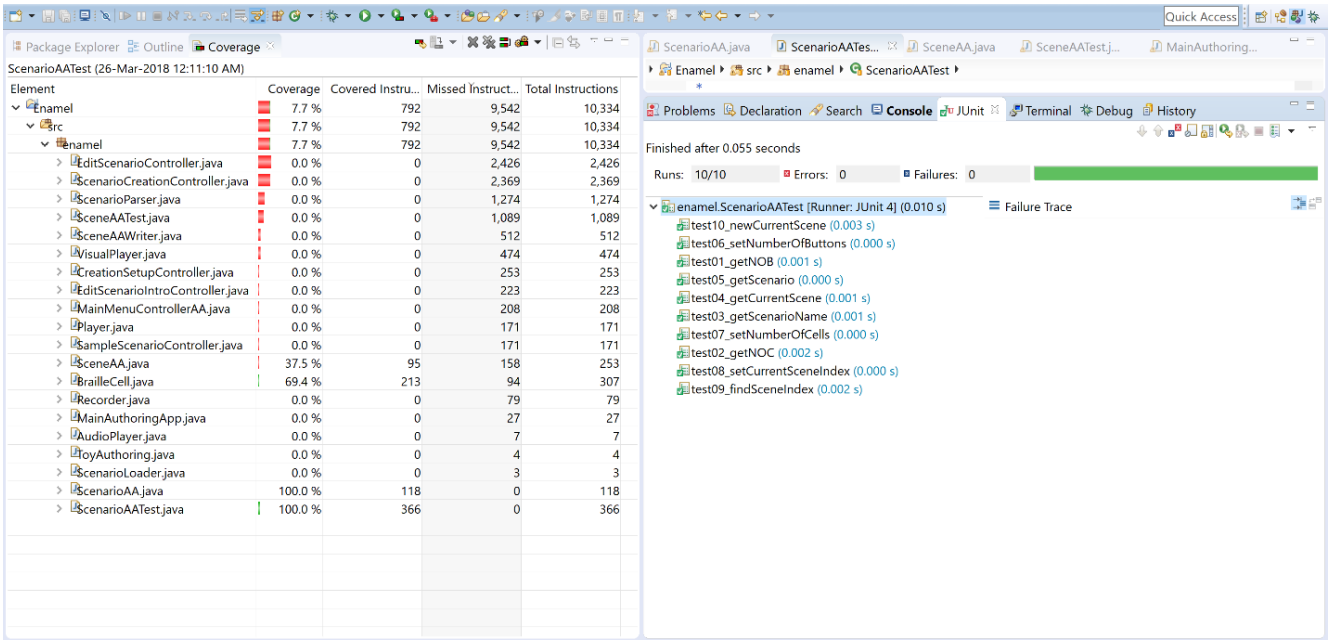

Figure 3.0: ScenarioAATest Coverage

SceneAAWriter Test Coverage

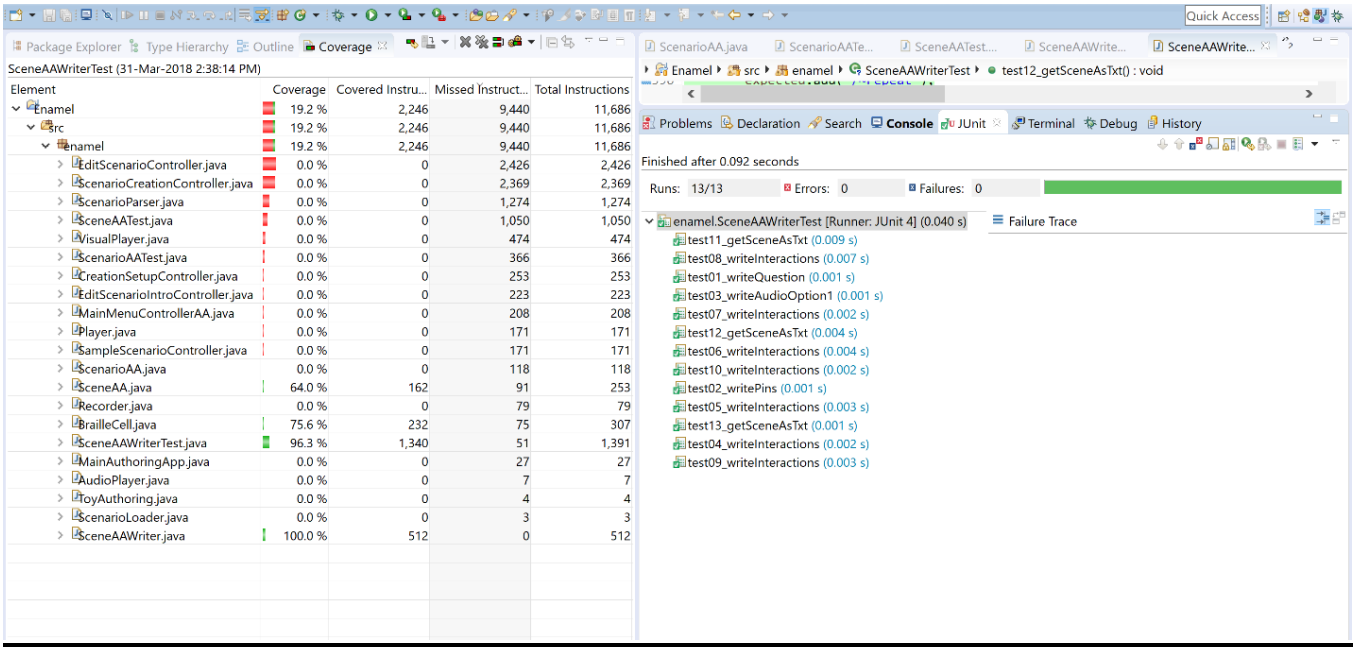

Figure 4.0: SceneAAWriterTest Coverage

The **SceneAA**, **ScenarioAA**, and **SceneAAWriter** classes all have 100% test coverage, however, this does not necessary mean that the tests are sufficient. Here we will explain why the test cases developed are a sufficient means of testing our code. To prove that our test cases are sufficient, let us understand what test case sufficiency means.

It is important to know that with large software projects, there is no way to guarantee with absolute certainty that you have accounted for all possible defects. There is any number of unfortunate events that could happen to any component involved. That is why software development is not about perfection, but sufficiency. Sufficiency means that at some point the developer(s) must declare their product "sufficient" for release. This does not mean that there are no bugs or defects, but that they have been reduced significantly and should not be a concern to the user.

Our tests are sufficient because we have accounted for boundary conditions and tried many ways to "break" our code. If we write a test that "breaks" our code this potentially means there is a flaw in the design or the method being tested has not been implemented correctly. For example, let us examine the second test written for the **SceneAA** class, "*public void test02_getInteractionAudioNameAtIndex()*." The purpose of this method is to determine if we can set an interaction audio name for a button created in the scene object. If the scene has only created five buttons, we would expect the test to fail if we tried to get a sixth button. This is a boundary condition because it tests to determine if an error shows up when we try to get a non-existent button. If this test did not fail, then we would be concerned because the method is not achieving what we have designed it to do. We also check to see if we can get the zeroth button. This test should also fail because we have designed our code in such a way that a scene must

have at least one button. We also check to see if we can get a button between one and five. We should be able to get a button in this range because the scene object has created five buttons. We would be concerned if we could not access a button within the scene. Thus, from this simple example we can verify that our test is sufficient because we have exhausted all the possible ways in which our test could fail.

This is just one of many tests we have created in which we account for boundary conditions. For a more in-depth discussion on each test case derived please refer to *Section 2.2 – How Test Cases Were Derived and Implemented.* Section 2.2 has already described in detail all the boundary conditions accounted for in the test cases.

Section 4 - Conclusion

Test cases are written to verify that a system under test satisfies requirements or works correctly. Test classes were only written for the components of our system that could be tested. Much of our code is XML which cannot be tested by writing a JUnit test case. The only way to verify that this part of the code works is to run the "*MainAuthoringApp"* and interact with the Graphic User Interface (GUI). From there it will be evident if components are working or not. We have gone into extensive detail about how our test cases were derived, implemented, and why they are sufficient. We have discussed that no software is without bugs and defects, but it is the duty of the developer(s) to come to a point in the development and declare the software to be good enough to use. This can only be achieved through extensive and appropriate testing. Testing helps find bugs in the code and leads to an improved software and that is why testing is essential.