

Testing Document

1. General Information
 - 1.1 Introduction.....
 - 1.2 Test Case.....
2. Overall Description
 - 2.1 Test Development.....
 - 2.2 Test Coverage/Test Sufficiency.....
 - 2.2.1 JUnit Test Coverage.....
 - 2.2.2 Authoring App Test Coverage.....
 - 2.3 Test Cases that were Run.....
 - 2.4 List of Features to be Tested.....
3. Conclusion
 - 3.1 Plans/Strategies for Future Testing.....

Section 1 - General Information

1.1: Introduction

This document explains our strategies on how we test our application, and we also discuss test sufficiency, test coverage, features to still be tested, advantages and disadvantages of our testing approach, and plans/strategies for future testing.

1.2: Test Case

Test cases are used to ensure that a program either passes or fails in terms of functionality.

Components included in a test case:

- Purpose
- Assumptions
- Pre-conditions
- Steps
- Expected outcome
- Actual outcome
- Post-conditions

Section 2 - Overall Description

2.1: Test Development

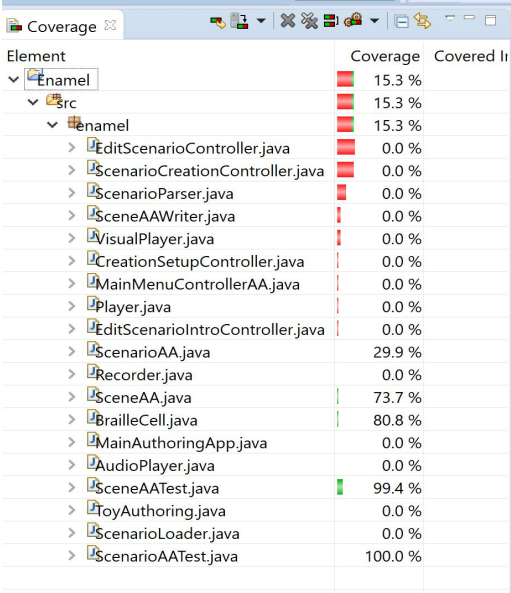
The test cases were derived after the code was implemented. We chose not to go with a Test Driven Development approach because it was more suitable for our team to test as we implemented code. Our approach to designing the app was to envision the functionality and versatility of what it could do. In doing so, we knew what classes and methods we wanted to create for our specific purpose. The tests were a by-product of this, and not the driving force.

This approach has its disadvantages and advantages. The disadvantage is that the objectives and tasks we would like accomplished become more open-ended and uncontained. It becomes more difficult to know what to work on next. However, the advantage of our approach is that we have more liberty in creating and designing the app because we are not restricted to implementing code based on specific test cases that were created prior to requirement refinement .

2.3: Test Coverage/Test Sufficiency

2.3.1: JUnit Test Coverage

The test coverage for the JUnit test cases is 15.3% as seen below in *Image 1*. There are a few reasons why the test coverage is low. First, notice the coverage for the ScenarioAA.java and SceneAA.java Classes. The coverage for these classes is 29.9% and 73.7% respectively. Most of the methods for these classes are getter and setter methods. Typically getters and setters do not require testing. It does not hurt to test these methods, but it is not crucial to do so either. The methods we focused on testing were the more meaningful methods that performed some functionality. For example, in the SceneAA.java class test cases were provided for the clearPins() method. This test was created to show that the pins of the braille cell could indeed be cleared. Another reason why the test coverage is low is because many of the classes were not tested at all. They were not tested because they either could not be tested or did not need to be tested. However, test coverage does include all code in the package even though it is really not relevant for our purposes. In short, the test coverage provided for JUnit is misleading because only 15.3% has been covered, but a lot of the code does not need to be included.

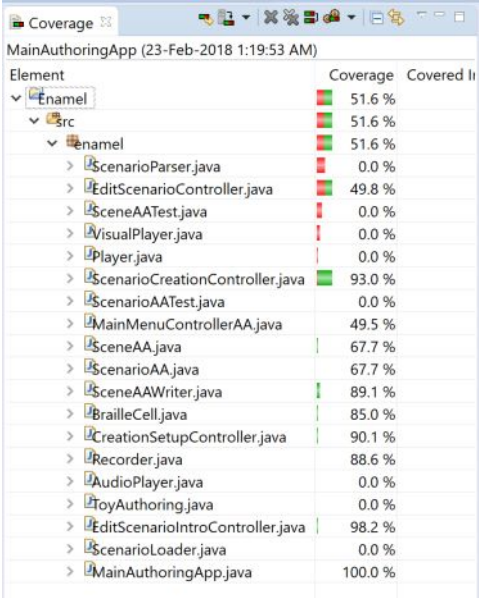


Element	Coverage	Covered In
✓ Enamel	15.3 %	
✓ src	15.3 %	
✓ Enamel	15.3 %	
> EditScenarioController.java	0.0 %	
> ScenarioCreationController.java	0.0 %	
> ScenarioParser.java	0.0 %	
> SceneAAWriter.java	0.0 %	
> VisualPlayer.java	0.0 %	
> CreationSetupController.java	0.0 %	
> MainMenuControllerAA.java	0.0 %	
> Player.java	0.0 %	
> EditScenarioIntroController.java	0.0 %	
> ScenarioAA.java	29.9 %	
> Recorder.java	0.0 %	
> SceneAA.java	73.7 %	
> BrailleCell.java	80.8 %	
> MainAuthoringApp.java	0.0 %	
> AudioPlayer.java	0.0 %	
> SceneAATest.java	99.4 %	
> ToyAuthoring.java	0.0 %	
> ScenarioLoader.java	0.0 %	
> ScenarioAATest.java	100.0 %	

Image 1: JUnit Test Coverage

2.3.2: Authoring App Test Coverage

The test coverage for the Authoring App is 51.6% as shown below in *Image 2*. This is a more realistic test coverage than that shown for the JUnit test coverage. In short, we now the Authoring App works because we can interact with it. Even if the JUnit coverage does not prove we have sufficient coverage, we know the App works. All methods that have been implemented are working. There are a couples reasons why the Authoring App coverage is not 100%. For instance the JUnit tests do not run while the Authoring App is running so this takes away from the total coverage. Secondly, in order to get the Authoring App to display a test coverage you need to interact with the App. The more you explore and use buttons the larger the test coverage will be. I cannot do an exhaustive test in which I interact with every button and combination of button in the App. It's just not practical, but I did as much as I could. Also for some reason the Load button does not attribute to the test coverage when it is used in the App even though it should. We are still looking into the problem and will have it resolved by release of the final product.



Element	Coverage	Covered Lines
MainAuthoringApp	51.6 %	
src	51.6 %	
enamel	51.6 %	
ScenarioParser.java	0.0 %	
EditScenarioController.java	49.8 %	
SceneAATest.java	0.0 %	
VisualPlayer.java	0.0 %	
Player.java	0.0 %	
ScenarioCreationController.java	93.0 %	
ScenarioAATest.java	0.0 %	
MainMenuControllerAA.java	49.5 %	
SceneAA.java	67.7 %	
ScenarioAA.java	67.7 %	
SceneAAWriter.java	89.1 %	
BrailleCell.java	85.0 %	
CreationSetupController.java	90.1 %	
Recorder.java	88.6 %	
AudioPlayer.java	0.0 %	
ToyAuthoring.java	0.0 %	
EditScenarioIntroController.java	98.2 %	
ScenarioLoader.java	0.0 %	
MainAuthoringApp.java	100.0 %	

Image 2: Authoring App Coverage

2.4 Test Cases that were Run

The following is a list of test cases used. To view their implementation, please refer to the code.

SceneAATest.java

- public void test03_getSceneName()
- public void test04_getButtonsNull()
- public void test05_getButtonsEmpty()
- public void test06_getButtonsNonEmpty()
- public void test07_getNOB()
- public void test08_setQuestion()
- public void test09_getSceneName()
- public void test10_clearPins()
- public void setPinsScene()
- public void getPinsAsBooleanTest() throws InterruptedException
- public void setInteractionPreset()
- public void setInteractionTextInput()
- public void getButtons()
- public void getAudioNameOption1()
- public void getInteractionAudioNameAtIndex()
- public void audioOption1Exists()

The majority of these tests are for getter and setter methods. However, there are a few methods worth mentioning. The *public void setPinsScene()* method was created to test if the pins could be set. For this test the desired pins were set to be “10000001” and at every index of the pin (indices 0 to 7) we asserted true or false. True means 1 and false means 0.

The *public void test09_clearPins()* method tested to see if the pins would clear (be set to ‘0’). We selected with an arbitrary pin state “10101010”, then proceeded to use *scene.clear(0)* to clear all the pins and then assert that at every index of the pin we would return false to prove that the pins have been cleared (set to zero) resulting in a pin state “00000000”.

One more method worth mentioning is the *public void getPinsAsBooleanTest() throws InterruptedException*. While it is a getter method, it was more complex than just “getting” a value. For this test we created a *BrailleCell* object and stored it in an arraylist that tells us what letter was being displayed by the pins. For example, we wanted to test that the letter ‘a’ was being displayed by the pins so the first boolean value in the list asserted to true and the remaining boolean values asserted to false which displayed “10000000” which is braille for the letter ‘a’.

ScenarioAATest.java

- public void test01_getNOB()
- public void test02_getNOC()
- public void test03_getScenarioName()
- public void test05_getScenario()
- public void test07_setNumberOfButtons()

- `public void test08_setNumberOfCells()`

These tests were for getter and setter methods so there isn't much worth explaining that wouldn't be stating the obvious. For this reason, we leave out discussion for these test cases. Full implementation of these tests can be seen in the code.

2.5: List of Features to be Tested

- Creating any number of scenarios the teacher would like to create
- Enabling the teacher to create a scenario as long or as short as they would like
- Being able to edit the scenario
- Making buttons functional
- Ensuring users can interact with GUI properly

Section 3 - Conclusion

3.1: Plans/Strategies for Future Testing

Moving forward we will have a more complete test coverage. While it is important to have more tests it's more important to test important pieces of code. It's great to have 90% test coverage, but if the bulk of the code lies in the 10% of the code that has not been tested then the test coverage can mislead people into thinking there are a sufficient amount of tests completed. The point is not to have 100% test coverage (although that is ideal, but usually not practical) but to test the most vital pieces of code that make the App work. This was and continues to be our objective, to test the most important code and of course we will strive to test as much of the code as we can.

We will continue to first implement the code and then test it. We like the freedom to create code and then test it as mentioned earlier in section 2.5: *The Challenges of Testing*.