2018

Testing Document

TREASURE BOX BRAILLE AUTHORING APPLICATION

EECS 2311 GROUP 3

Testing Document

Introduction

To provide the best results and good performance of the authoring app, some test cases were determined to provide the best user experience. These cases are of much important as they cover the basic functions that will be used by the user on each run. Additionally, all of the different features found within the authoring app were also tested to see if they were being implemented into the scenario parser successfully.

Test Case	Test Case Description	Procedure	Notes/Expected Result	Actual Result	Test Case Results
1.) Opens Scenario Text File	This test checks if the authoring app can open and read the specific scenario text file for reading and further modifications	1.) Open a sample scenario file. 2.) Check to see if the opened file is detected, and is same as the JFileChooser selected file within the authoring app.	Expectation: The opened file matches the same name and contents as the JFileChooser selected file.	As Expected	**Pass**
2.) Save Scenario Text File	This test checks if the authoring app can save the specified scenario text file and overwrite changes to the edited content	1.) Save scenario file as Scenario_test.txt 2.) Using JFileChooser, check the contents of the Scenario_test.txt file.	Expectation: The contents of the Scenario_test.txt file should match the content created within the authoring app.	As Expected	**Pass**
3.) Create New Scenarios	This test checks if the authoring app can create new scenarios,	1. Create Scenario button press is simulated 2. The content of the commands within the	Expectation: The contents should contain 2 lines, Cell [num] followed by Button [num]	As Expected	**Pass**

	populate the fields representing options with default text which can later be edited to personalize the scenario to user choosing	scenario are checked.	where numbers in this case should be 1 and 4 respectively.		
4.) Add New Event	This test checks if the scenario file can have a new event added in any sequence into the file, formatted correctly and neatly organized (This test checks for the Add User Input Button)	1. One of the features, User Input is inserted by calling a custom method created for testing. 2. Check the contents of the authoring app to see whether they contain the inserted feature code.	Expectation: The content in the authoring app should contain "/~user-input". Which matches the event(feature) that is being tested in this case.	As Expected	**Pass**
5.) Check If Valid Scenario File is Imported	This test checks if the imported scenario file is a valid scenario file (begins with Cell num1, Button num2) where num1 and num2 are valid integers representing the number of cells and	1. Given 2 files, checks use JFileChooser to load the contents of both files 2. Check to see if the content of both files matches the criteria for a file to be considered a scenario file.	Expectation: The content of one of the two files is supposed to match the criteria, starts with Cell 1 [enter] Button 4, whereas the second file does not match the criteria. The first file returns true, second returns false.	As Expected	**Pass**

	buttons that the scenario file corresponds to.				
6.) Create New Audio File	This test checks if a new audio file can be created with a name, and duration provided which then can be later imported into the scenario file to be incorporated into the scenario for audio use.	1. Using the recorder class, create a dummy audio file for 1 second with the name "tester1212.wav" 2. Using the File class, check if the newly created file indeed exists.	Expectation: A file named "tester1212.wav" exists in the AudioFiles folder as required by the creation through the AudioRecorder class.	As Expected	**Pass**
7.) Create Text to Speech Feature	This test checks if the text to speech feature can be created with a custom text that the scenario reader reads out when executing the scenario file. This test will check to see if its incorporated within the scenario command list	1. Using triggerScenario method in AuthUtil class, simulate the addition of adding a new text-to-speech text into the scenario file 2. Input "Hello Test" into the dialog box that pops up in test-mode asking for a text to show.	**Note**: As the feature requires a specific text to check for when testing, "Hello Test" was the chosen text to see if the feature works. **Expectation**: After typing "Hello Test", this text should show up as the contents of the scenario file being created and hence added into the commands list of the scenario.	As Expected	**Pass**

8.) Add a Pause into Scenario Test	This test checks to see if the pause is successfully incorporated into the commands list of the scenario.	1. Using triggerScenario method in AuthUtil class, simulate the addition of adding a new pause to the scenario. 2. In the dialog box that shows up, insert 5 to reference 5 seconds for the pause.	**Note**: As the pause features has a parameter (number of seconds to pause for), 5 seconds was chosen as a test number. **Expectation**: After typing "5" for the number of seconds, "/~pause:5" should show up in commands list of scenario.	As Expected	**Pass**
9.) Display Braille String Test	This test checks to see if a braille string is displayed in the commands list of a scenario file.	1. Using triggerScenario method of AuthUtil class, simulate the addition of adding a display braille string to scenario. 2. In the dialog box, enter the string to display using the pins.	**Note**: As the display braille string feature has a parameter (a string to display), "Test" was chosen as the test string. **Expectation**: After typing "Test" for the braille string, it should show up in the commands list of the scenario.	As Expected	**Pass**
10.) Set Specific Cell Test	This test sees if a specific cell is set based on the 8-digit binary number inputted within the app. It also checks to see if a valid cell is selected.	1. Using triggerScenario method of AuthUtil class, simulate the addition of set specific cell feature. 2. In first dialog box enter "1" to choose first cell.	**Note**: As the set specific cell feature has 2 parameters (a cell, and a 8-digit number), "1" was chosen for the cell and "10101010" was chosen for the 8-digit number. **Expectation:** After choosing "1" for cell and	As Expected	**Pass**

		3. In second dialog box enter "10101010" as the 8-digit binary number to set for the specific cell.	"10101010" for the number, the command should be added into the command list and be visible in scenario		
11.) Reset Buttons Test	This test serves to see if the reset buttons command is entered and visible in the commands list of the scenario.	1. Using the triggerScenario method of the AuthUtil class, simulate the addition of a reset-button feature.	**Expectation:** The "/~reset-buttons" command should be visible in the command list of the scenario that is being created.	As Expected	**Pass**
12.) Test Hotkeys	This test serves to ensure that the hotkeys that are provided within the program are correctly working properly.	1. Open authoring app 2. Test all the hotkeys as labeled in the manual e.g CTRL + N creates a new scenario	**Expectation:** All specified hotkeys work, and accurately do the task they were designed to do, hotkey doesn't compute the wrong function.	As Expected	**Pass**
13.) The Test Scenario Button Test	This test serves to identify if any issues occur with the test scenario button to see if the scenarios can be played.	1. Open authoring app 2. Create a dummy scenario 3. Add a dummy feature such as the text-to-speech 4. Click the Test Scenario button AND/OR CTRL + T	**Expectation:** The test scenario button will open up the simulator which will allow the user to see the output of their scenario.	As Expected	**Pass**
14.) Test Move up button for Scenario movement	This test serves to determine if scenarios feature	1. Open authoring app 2. Create dummy scenario	**Expectation:** The pause feature will show above the text to speech feature	As Expected	**Pass**

	commands can be moved up using within the scenario file being created.	3. Add dummy features such as a text to speech as well as a pause. 4. Click move up button AND/OR CTRL+ [Up Key] while the pause command is being selected.	from before, effectively reversing their order.		
15.) Test Move down button for Scenario Movement	This test serves to determine if scenarios feature commands can be moved down using the scenario file being created.	1. Open authoring app 2. Create dummy scenario 3. Add dummy features such as a text to speech as well as a pause 4. Click move down button AND/OR CTRL+ [Down Key] while the text-to-speech command is being selected	**Expectation:** The text to speech feature will show below the pause feature from before, effectively reversing their order.	As Expected	**Pass**
16.) Test Remove Feature for Scenario Command Removal	This test acts as a way to test if commands that were created can be removed from the command list.	1. Open authoring app 2. Create dummy scenario 3. Add dummy feature such as a text-to-speech feature. 4. Click the remove button AND/OR CTRL + X while selecting the text-to-speech scenario	**Expectation:** The text-to-speech feature will be removed from the commands list and only the number of cells and buttons will show.	As Expected	**Pass**

Discussion – How these Test Cases were Derived, Implemented and their Sufficiency

1. Opening the scenario file and reading the text file is one of the main tasks for this project as if file is mishandled while being imported, data corruption could incur causing the scenario file to misbehave and cause inconvenience to the end user. This test case was derived keeping the end-user in mind for providing user-friendly experience where everything is smooth from the get go. This test case was implemented by making a new instance of JFileChooser and setting it to one of the test scenario files provided, then the file name was tested to check if the name of the file was recognized by the JFileChooser.

2. Saving a scenario file is equally important as opening a file as the scenario file is used only after it has been saved successfully. This test is meant to check if the scenario file is saved successfully to provide the playing app with the correct sequence of lines which are formatted properly. This test case was derived as a case to cover up the defects of saving files that might be read-only or may not be able to be overwritten. This test case should provide the user with a scenario file being created no matter the circumstances where the end-user doesn't need to go through technical troubles of overwriting a file and losing previous work or not being able to save to a file due to read-only property of files being true. This test was implemented by creating a new test scenario file using the JFileChooser and saving it using the same method as in the Authoring App with some small test string saved as the sequence.

3. Creating new scenarios is another integral part of the authoring app as it provides the user with a scenario file from the get go with a dummy scenario file to get started with. Although the process might not be appealing, this provides the user to test out each feature of the scenario file and modify to their liking as they please. This test case was derived for user-friendliness of the features to the end-users as having to learn a new software is a hard task especially if it handles working with creating scripts for external devices that cannot be easily tested. This was implemented by using the same method as the one used in Authoring App which adds sample scenario code to the text area which then gets compared to the specific text in the test case for comparison.

4. Adding new events is a feature that allows the ability to add a state in the script where something happens of important. This allows the user to perhaps ask a question (an event) which can be triggered at a stage of the program, and provides the user the ability to add many important features to the scenario file. This is important because it provides the handler with different things to work with for example, each question could be an event and asked in their own space. This test was derived from the requirements of the authoring app as it was needed to provide the end-user to add events into the scenario file app. This was implemented by using addUserInputString ()

method which adds the "/~user-input" to the text area which is compared in the JUnit.

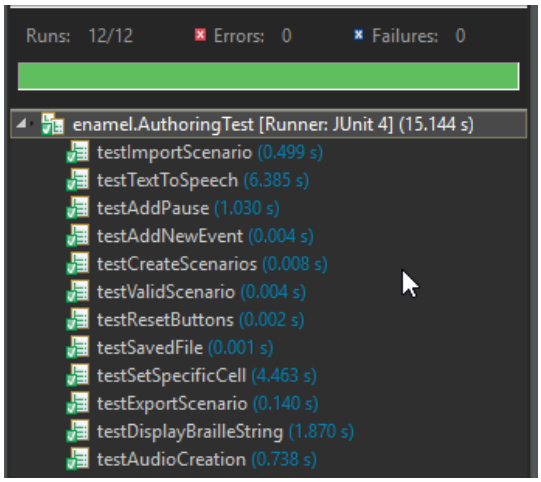
5. Checking the type of file being imported is important as not all text files are valid scenario files. A valid scenario file is recognized by the first line having Cell num1, followed by the second line having Button num2 where num1 and num2 are positive integer values representing the number of each components to have. This makes sure that these two cases are present when importing a file as if these two cases are not present, then the imported file is not a valid scenario file. This is important because without this check in place, the user would be working on an invalid file which will not function with the braille simulator causing errors. This test as derived as a mandatory requirement of the authoring app because these two lines are required for the scenario file to function properly. This test was implemented by checking two different files, one file being one of the provided default scenario files and another file being a normal text file. A Scanner was used to check the first two lines of the text file for the presence of those two keywords (Button and Cell) and if those existed, the file was given the true attribute as being a scenario file and false was given otherwise.

6. Providing the user to record audio was another requirement of the authoring app to go with the sound feature of the scenario file where a sound is played during the execution of the scenario file script. This was derived as a way to check if the required audio recording function was working and generating a .wav file on execution. This is an important test because it is vital to check for the generation of the audio file being created after a recording to see if it functions properly. This test was implemented by calling the AudioRecorder class specifying a name for the test audio file and its duration. The test name chosen is tester1212 whereas the duration chosen is 1 second for simplicity of the test. Once the recording finishes, a file is supposed to be created. The test checks for the existence of this file to determine if the test has passed or not.
7. After regular testing of the app, it was recognized that text to speech was a very common feature an would be used a lot more often that the other features listed within the toolkit. This test was derived as way to test one of the more common features as more usage would increase the chances of finding more bugs within the program. This is an important test because it is a lot more important to test the more commonly used features as they have the most potential of finding bugs due to higher occurrence rate. This test was implemented by using the triggerScenario method within the AuthUtil class to match the same conditions as an end-user would face/use, hence it's also considered sufficient.
8. After regular testing of the app, one other feature that was recognized as a common used feature was the pause feature. This test was also derived solely due to the fact that pausing would be very common and hence no bugs should occur as higher usage of something leads to higher chances of finding bugs and could disrupt the end user. This test was implemented by using the triggerScenario method within the AuthUtil class to

same conditions as an end user, as this is how the main program calls for the pause to be added to the commands list of the scenario reader. As this test checks to see if the pause feature can be added successfully, and there are number constraints added onto the feature, it is sufficient enough to show that bugs would occur very rarely if at all.

9. Display Braille String was one of the features within the program which had many was also shown to be one of the more common features from testing and hence it was chose not to be further tested. This test is important because displaying a braille string should only display a string and this was something that needed to be checked for as without testing, the program allowed numbers and all other characters in place which would cause bugs and glitches. This test was derived after experiencing different conditions using this feature which caused one of the testers to have glitchy scenario readings when tested for due to not knowing that only strings can be inputted. The test is also implemented using the triggerScenario method from the AuthUtil class which mimics the same output as it would to an end user which provides a good testing environment as it is tested in ideal consumer conditions. With that said, it is considered sufficient as the tests provide enough coverage to cover the basis of an end-time user.
10. Set Specific Cell was considered one of the features with multiple parameters and hence had a high change of being more glitchy than the other features in the toolkit. This feature allows users to set a specific cell with a specific braille pattern based on an 8-digit binary number. This test was derived with the idea to test all conditions that an end-user might try while using this feature which helped programming the error statements. This test was important because although, not a very common feature in our initial testing, it had 2 parameters which could cause for different type of errors which may or may not be accounted for. After testing, this test was shown to be sufficient as it tested and caught many types of errors.
11. The reset button test was also a very important test due to being one of the more popular features in our initial testing. This test was derived to see if the reset-button command would correctly input into the commands list after being called or pressed. This test was important and holds a very big responsibility as it has the power to reset the buttons and their functions within the braille scenario reader and hence should be tested to be bug free. Although there weren't many places a bug could be found, a few bugs relating to output to the commands list were found during testing an were resolved hence this test was very sufficient.
12. The test hotkeys test was a way to check if the hotkeys were working accurately amongst their specific functions. This test was derived in a way to check if all the hotkeys were mapped correctly and working as desired. This test was important because hotkeys are a major way for impaired people to quickly work through a computer app, and hence is a good time-saving function to allow for fast scenario making. This test was considered sufficient because it was a manual test, and since all hotkeys were working at all times tested, it showed that the test covered all of its basis.

13. The Test Scenario button is a very commonly used button within the authoring app as it's the only way to debug the scenario and see if it is working as desired. This test was derived because there were times when the test scenario button would glitch out and spit out an error. As an end-user, you want easy and bug-free debugging of a scenario code and so this tests purpose was to provide just that. This test is important to provide the user with a button to check if the scenario is working as desired. As the test scenario button uses a simulator library, the test itself was sufficient as the bugs were not in our control to mostly fix.
14. The move up button was also tested due to its importance while editing a scenario file. It was derived due to experiencing bugs in initial testing where the selected command in the commands list was not moving up to its projected location. This test was important because, end-users like to move things around a lot, and this button provides them the freedom to do just that. With that said, it needed to do that without causing any troubles and hence this test was made to check for bugs and handle errors that might potentially occur in using this button. This test in the end was deemed sufficient enough as it moved features up as required in all cases and was successful.
15. The move down button was also tested due to key important as well while editing a scenario file. It was derived as a way to check for any bugs after initially having found bugs with the move up button. It is very important because moving stuff down (alongside up) is a key feature while working within a "coding" environment. With that said, this function, with bug fixes relative to the move up, was found to be efficient and bug free for the most part and would be considered sufficient and successful.
16. The remove command in scenario editing was also a big feature that was very common and required much testing. This feature was derived due to the removal feature initially causing removal of other commands in the commands list. With further testing, the bugs were recognized within the remove button and were fixed. This was important to provide the end-user the utmost clarity and concise removal of commands without errors showing up. This function has now been tested manually and in all cases removed the specific lines successfully and hence is considered sufficient.

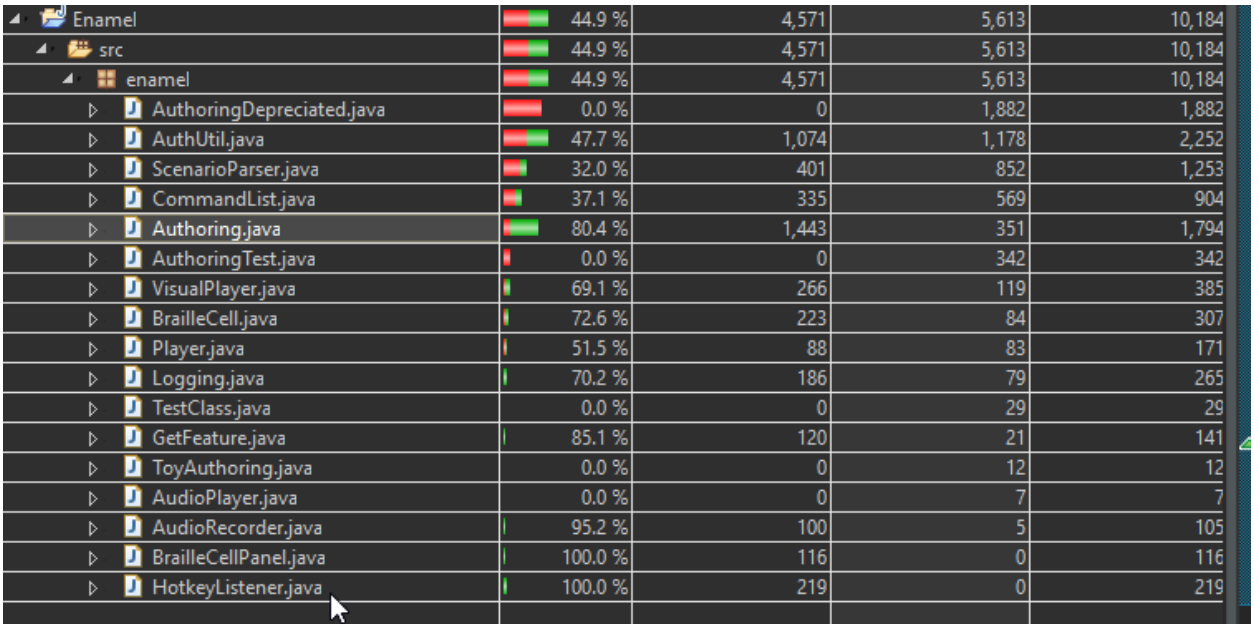


J Unit Testing Results

The J Unit results were all successful and ran fairly fast. These test cases were very helpful in determining bugs in initial testing as well as making sure the end user will not experience application breaking bugs.

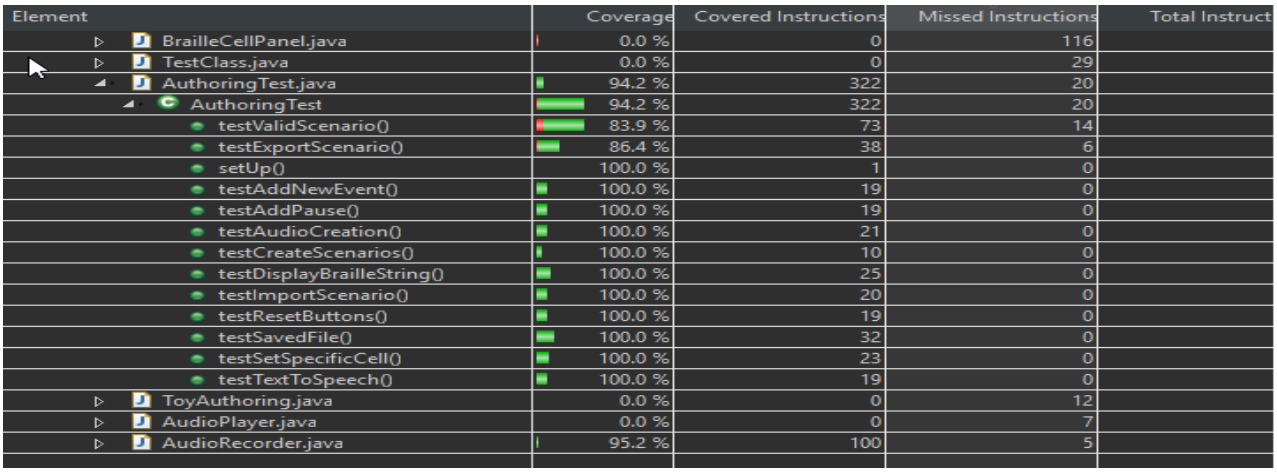
Test Coverage

Due to the testing cases being limited, the coverage percentage isn't as high as it can be for the whole project. As of now, the coverage for the entire project is nearly 45% whereas the coverage for the main file being worked on is 80.4%. This does not heavily impact the state of the program as coverage is only a tool to see which lines of code is used within a program and which are not. As there are many error handling statements which were not used in this one occurrence of the runtime of the program, the AuthUtil class and CommandList class did not have as high of a coverage as expected. With that said, the test cases tested implemented within the authoring app are sufficient enough to manage the most common functions and provide better coverage. An image of the coverage is shown below:



Enamel	44.9 %	4,571	5,613	10,184
src	44.9 %	4,571	5,613	10,184
enamel	44.9 %	4,571	5,613	10,184
AuthoringDepreciated.java	0.0 %	0	1,882	1,882
AuthUtil.java	47.7 %	1,074	1,178	2,252
ScenarioParser.java	32.0 %	401	852	1,253
CommandList.java	37.1 %	335	569	904
Authoring.java	80.4 %	1,443	351	1,794
AuthoringTest.java	0.0 %	0	342	342
VisualPlayer.java	69.1 %	266	119	385
BrailleCell.java	72.6 %	223	84	307
Player.java	51.5 %	88	83	171
Logging.java	70.2 %	186	79	265
TestClass.java	0.0 %	0	29	29
GetFeature.java	85.1 %	120	21	141
ToyAuthoring.java	0.0 %	0	12	12
AudioPlayer.java	0.0 %	0	7	7
AudioRecorder.java	95.2 %	100	5	105
BrailleCellPanel.java	100.0 %	116	0	116
HotkeyListener.java	100.0 %	219	0	219

An image of the separate AuthoringTest class coverage is shown below:



Element	Coverage	Covered Instructions	Missed Instructions	Total Instruct
BrailleCellPanel.java	0.0 %	0	116	
TestClass.java	0.0 %	0	29	
AuthoringTest.java	94.2 %	322	20	
AuthoringTest	94.2 %	322	20	
testValidScenario()	83.9 %	73	14	
testExportScenario()	86.4 %	38	6	
setUp()	100.0 %	1	0	
testAddNewEvent()	100.0 %	19	0	
testAddPause()	100.0 %	19	0	
testAudioCreation()	100.0 %	21	0	
testCreateScenarios()	100.0 %	10	0	
testDisplayBrailleString()	100.0 %	25	0	
testImportScenario()	100.0 %	20	0	
testResetButtons()	100.0 %	19	0	
testSavedFile()	100.0 %	32	0	
testSetSpecificCell()	100.0 %	23	0	
testTextToSpeech()	100.0 %	19	0	
ToyAuthoring.java	0.0 %	0	12	
AudioPlayer.java	0.0 %	0	7	
AudioRecorder.java	95.2 %	100	5	

Conclusion

Although the coverage results are lower than 50%, the test cases that were implemented all passed and their importance was in our opinion very high as these test cases test the main functions of the authoring app. Furthermore, the coverage of the test cases was at a high of 94.2% and hence showed that the coverage of the test cases was much better. All in all, our main goal is to provide the end-user with workable, and efficient features that they can use and not have to deal with bugs. In that prospect, despite the lower overall coverage, the test cases cover all the necessary parts of the application for an end-user to use the application in a bug-free environment and have a pleasant experience overall.