TEST DOCUMENTATION

Testing Philosophy

The Treasure Box Braille Scenario Editor for use with the Treasure Box Braille Learning System enables educators to create and edit scenario files . This scenario editor will be useable by all users regardless of visual impairment. To accommodate those users, a screen reader will be integrated with full functionality. Therefore, testing needs to reflect these multiple usage cases.

Test Plan

Front End Testing

The functionality of the user experience interfacing with the software needs to be tested as to make sure that the user doesn't get subjected to glitches and irritating bugs that ruin the user experience.

Back End Testing

In addition to testing a functional user experience. Testing the correctness of class methods with respect to their class access permissions will validate the quality of the software and ensure a correct educational experience for both the scenario editors and scenario users. Also going to test different types of files as inputs and to check their validity and how the GUI should deal with those. Also the templates that get generated need to have the proper structure so that it gets represented on the GUI well without errors

Test Scenarios

Visual Testing

This testing involves interfacing with the software through the user interface (UI). The UI is implemented using Swing, and therefore is platform independent. The UI is also going to support popular screen readers and they are going to be thoroughly tested to make sure that all the components that need to have the accessibility features do their job

Test Cases

Since the software usage revolves around using scenario text files and whatnot so our testing cases are going to be talking about file or I/O exception handling for the most part and then there are the other test cases and they will be listed below

1. <u>When the user has no file selected but still tries to play or edit a scenario (Error Type: File):</u> This is probably one of the most common error scenarios. This happens when a user tries to press on the buttons that make use of the file, but the user does this without selecting a specific file to do those actions on. So our GUI catches this error and displays it on a JOptionPane and notifies the user that he/she has not selected a file and the way they would select a file is by going through the Open Scenario button in the file menu. The GUI displays the same JOptionPane when the user decides to edit a scenario without selecting a file first. The testing for this case involved us passing a null file which imitated a scenario of no file by the user. The buttons play and edit scenario are the only files that require the user to select a file first and so we believe that testing it for these 2 buttons was sufficient.

2. <u>When the user selects a wrong file type (Error Type: File):</u>
   This is another case when the user actually selects a file, but the file happens to be a non-scenario file and to prevent the sound player to crash and log an error, we make our GUI handle this as well. What our GUI does in this case is that it first reads the file that was selected and decides whether or not the file is a text file and if it is of a different type then the GUI shows an JOptionPane which says that the file is a not a valid scenario file but if the file that was selected was .txt file or a “plain” file then our GUI opens a up a file reader and then reads the first 2 lines of the supposedly scenario file and checks whether or not the first 2 words of the first 2 line have the words “Cell” and the “Button” in them. If that’s not the case, then our GUI throws a JOptionPane saying the file you selected isn’t a valid scenario file. We tested this test scenario by passing in different file types and logged what the GUI did with different file types and if it only accepted scenario files. We believe that this was sufficient because the file that the user selects must be a scenario file and pretty much everything else our GUI does is dependent on selecting a valid file first or else the started code for the sound player wouldn’t run.

3. <u>Clicking on either the File or the Edit buttons makes sure that the ribbon for one button isn’t open while the other one is clicked (Type: GUI):</u>
   As you may have noticed we were inspired by the ribbon style buttons that can be found in MS Office. So clicking on the file ribbon opens it up but clicking on the edit ribbon also opens the ribbon for the edit button and the ribbon buttons that were made when the file button was clicked still stayed and they would have interfered each other. So in our test we had to make sure that non of the buttons on the file ribbon stayed on the main panel while the edit button was clicked and vice versa and we knew this was sufficient because there were only 2 buttons that support these ribbons and the only way to fix this was when one of the buttons was clicked it removed the other button’s ribbon

4. <u>Clicking on the moving up button on the top template and clicking on the move down button on the bottom template does nothing (Type: GUI templates):</u>
   Our Edit Scenario button generates a list of templates for all the lines that were in the file. So these templates have buttons and two of those buttons are called Move Up and Move Down which moves this template up or down respectively. This can also be a common error because clicking on the top most button on the top template would cause and IndexOutOfBounds exception and cause the program to crash since our program uses an arraylist to store all the template objects. So our test case made sure that clicking the Move Up button on the top most template or the Move Down button on the bottom most template forces those buttons on the GUI to not do anything in those 2 specific cases.

5. <u>Making sure that the accessibility descriptions that were set to most of the components for our GUI react to screen readers (Type: Accessibility):</u>
   Since the main point of the software was to make it accessible to users. We had to make sure that the main functionality of the software was made to be accessible. We tested our components to make sure whether it gets picked up by screen readers. And the way we did it

was actually use a screen reader specifically and the one we used was NVDA. Doing so made it sure that the buttons were saying what they need to say.

Test Coverage

Testing the coverage of the code is important because it lets the developers know how much of their code is being covered with the test cases they made. It gives an idea of how much they need to test their code and how much their code is run in an exhaustive usage of their software. Low code coverage indicates that more testing is necessary while high code coverage gives little information about testing quality

Our code has a coverage of 57% and it is a little lower than the average of 70% in big systems because our code uses the built in sound player and instead of letting that take care of the errors our code instead makes sure the file that sound player gets is a valid one which explains why we only use 47% of the starter code provided but our UI instead has a coverage of 92%