TalkBox Testing Document

Version 2.0 - Final Submission, April 3rd 2019

EECS 2311 Submission: Group 12

Mohammed Nasiful Haque
Amaan Vania
Tony Ly

Revised	By
Feb 24th 2019, Midterm Submission	Mohammed Nasiful Haque
Apr 3rd 2019, Final Submission	Mohammed Nasiful Haque

Purpose

The purpose of this document is to give a summary of the testing procedures the TalkBox Application went through to verify the correctness of the software and how it ensures that the features that are mentioned in the Requirements document work as expected.
How these test cases were derived and the way they were used to test the applications is also going to be discussed in this document and also if these test cases were enough to ensure the applications work as expected.
It is also going to discuss the testing coverage of said testing in regards to both the TalkBox simulator and the TalkBox configuration app.

Test Plan

Front End Testing:

The functionality of the user experience interfacing with the software needs to be tested to make sure that the user doesn't get subjected to glitches and irritating bugs that ruin the user experience. In this software package, there are two sets of GUI. One of them is for the TalkBox simulator and the other one is the TalkBox Config App and each of them needed a different approach to testing as they each did completely different things with just a few similarities between them.

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 smooth experience for both the users of the TalkBox Application and also the Config Application. The Back-End mostly involved testing the functionality of both the applications and so all the features and the tasks the application was capable of, had to be tested to ensure that the application performed, as promised in the Requirements document.

Test Scenarios

Manual Based Testing:

This testing involves interfacing with the software through the user interface (UI). The UI is implemented using *JavaFX* and therefore is platform independent. This testing was performed through actual usage of the software and covered usage scenarios the user would find themselves in as per the requirements document. This was done repeatedly with multiple users and their feedback was then used to further improve the Graphical User Interfaces of both the simulator and the config app.

Test Cases

The test cases/scenarios were divided between the main applications of the project. Since each of the apps had their own functionality and responsibilities, the approach for the test cases/scenarios were just as different

TalkBox Configuration App:

- **The user shouldn't be able to delete more buttons than there are available:**
  Derivation: When the user selects to make a new project, they will be greeted with 6 available audio buttons to work right from the start. The problem arises when the user decides to click on the delete 'last' button more than 6 times which results in crashing the application. The same error occurs when the users try delete a specific button using the index option.
  Implementation: This test case checks the number of audio buttons currently active by keeping all their references in an Arraylist of audio buttons. Deleting a button removes an Audio Button reference from this list. This test case will only be successful if the delete button ceases to function when the size of this Arraylist becomes zero which means there weren't any Audio Buttons to remove.
  A similar approach is taken for 'Deleting an Audio button at a specific index' as there is function that checks that the index the user provided is within the bounds of the same Arraylist. This test case is successful if it correctly removes audio buttons from the list when the index is within bounds and also it notifies the user the index is not within bounds.
  From the perspective of adding and deleting buttons, this test case is enough as it covers the most important edge case. There is no upper limit on how many buttons the user wants to add so the only concern was how many can you remove.

- **No empty fields are allowed when the user 'Edits' a button in the Config App**
  Derivation: When the user is done selecting the number of buttons they will be met with the configuration app. One of the things that the config app allows to do is configure Audio Buttons with the appropriate title, audio and image. If any of those fields are left empty before clicking on the submit button, it results in the entire configuration app crashing. Incomplete audio buttons would be created and that would result in errors such as unwanted crashes when the config app tries to communicate and transfer these audio buttons over to the TalkBox simulator.

  Implementation: if we can successfully catch and act appropriately when the 'Title' textbox is empty i.e by using the 'getText' function of the textbox to see if it matches with an empty string, if not then we call the function that creates the audio button. The way we know the testcase is successful is when the configuration app behaves as expected when the title textbox is filled or it throws an error prompt when it is empty.

  This test was good enough for the Edit button because the only way the user could change the state of the configuration app (Edit already initialized buttons) was by filling in the fields in the Edit window correctly.

- **When the user leaves unedited (not filled) audio buttons in the middle of two other already filled audio buttons:**
  This occurs when a user decides to leave gaps in between the Audio buttons in the config app. The config app opens an error pane to warn the user that there can't be any gaps in between the audio buttons in the config app. Otherwise,it results in transferring unedited audio buttons for the talkbox simulator to build.
  The handling of these two scenarios ensure that the information that the TalkBox simulator gets is correct and so the Audio buttons the simulator builds is the same no. of audio buttons the user actually filled and edited in the Talkbox configuration app; There aren't any buttons that go missing when the configuration app communicates with the simulator and vice versa.

- **The File Choosers can only open files of specific types:**
  Derivation: Every single action that opens the FileChooser in this application makes the user intentionally limit the types of files they can choose. This can be seen whether the user wants to open or edit an existing .tbc file from the welcome screen. the Jfilechooser only pick .tbc files. This was done so that the application need not make any further checks as to what types of files the user uploaded and can get on processing the file knowing sure that the file that user uploaded is of the type the application can definitely work with. Otherwise, the configuration app would crash because the file they would be serializing and deserializing wouldn't be the type of file that can be serialized/deserialized.

Implementation: The way this was implemented was using the *Extensions filter* method of the JFileChooser and then limiting the choices to just be .tbc files. Therefore every time the user gets around to opening the Filechooser for a specific purpose, they can only select files with extension '.tbc'. The text case is successful if the file the user chose was of the type .tbc and this was done by checking the extension of the file the user just chose by matching it with string ".log".

This approach was good enough because the user uploads a file of the correct type as ensured by the Filechooser.

- **Testing the Drag & Drop feature:**
  Derivation: One of the features that were implemented in the Config App was the ability to Drag and Drop Images into the Edit buttons that get generated. We needed to ensure the drag and drop feature actually worked. There were a few things that needed to be considered. We had to check if the drag and drop action updated the text field label of the audio button that the image was dragged on to. This is crucial because dragging an image on to an audio button should only update that specific audio button but all the remaining Audio button states need to be remain unchanged.

  Implementation: We tested it by generating different number of buttons on the GUI and then dragging and dropping images on to the intended audio buttons and opening up the Edit window of that specific button to see if the path of the image was successfully updated and then checked if it carried over correctly to the TalkBox simulator app when the Run button was pressed. Also checked if the file was properly saved when using this feature and if the .tbc file that was saved could be opened up the TalkBox simulator. The test case was successful if after going through the actions of dragging and dropping. The textfields of title, audio and video were filled with the same title, audio and video the user intended to put in there in the first place

- **Testing the Autofill Feature:**
  Derivation: The autofill feature is a quality of life feature added to the configuration app that allows the user to fill an audio button automatically by suggesting a word. Depending the word the user suggested, a text to speech audio file of that word is generated alongside a related image of that word and also the title of the audio button is set in accordance to the word the user selected.

  Implementation: This feature had numerous test cases. They involved checking the if the connection request sent to the API returned an actual image url which needed to be download automatically. Then the text to speech feature was tested by making sure that the word the user entered during autofill is exactly the same word that gets converted to text to speech in the form of an audio file. Lastly the title was checked before and after the audio button is created to ensure that the audio buttons weren't in an 'incomplete' state. If all three of these fields were put in then the test case is successful.

- **Testing the Record Audio feature:**
  Derivation: Another noteworthy feature of the TalkBox config app was allowing the user to record their own audio and then saving it on their machine for use in the config app. There was a need to know if the recording audio function worked because the whole idea of personalizing the experience was achieved through this feature.

  Implementation: The way it was tested was, assuming the user successfully followed the steps of recording the audio (Clicking the record button, speaking into it, having a microphone installed and then saving the file). We tested this with various recorded audio files with different lengths using an external music player (outside of the TalkBox environment) to ensure that the files that were recorded worked the way the Requirements document intended.
  Also this feature gives a warning to the user if they don't have a microphone installed. The way it was tested was by disabling the microphone feature and our own machines and then opening up the Configuration Application to see if the Record Audio feature was able to notice that there wasn't any microphone available for the application to use. The test case is successful if after the recording was finished, the audio button plays as expected, otherwise the Recording feature gives you a warning saying that it didn't find any devices to record audio on.

TalkBox Simulator:

The TalkBox simulator is mostly dependent on what the Configuration app feeds it. So the test scenarios and cases are fewer in number.

- **All the buttons on the welcome screen work as expected:**
  The buttons that first greet the user in the welcome screen of the TalkBox app had to be checked to ensure that there weren't any dead links connected to the buttons (or the buttons being dead themselves). There are a total of four events that can happen in the simulator welcome screen. All the buttons below were tested to work consistently on multiple platforms across various devices to ensure a positive user experience.
    1) Creating a new .tbc file
    2) Open an existing .tbc file
    3) Help
    4) GitHub link

- **The connection between the TalkBox Simulator and the Configuration App:**
  Derivation: The TalkBox simulator depends on everything the Configuration app sends. This is important to test because this connection is extremely important for the Simulator to be able to simulate what the config app wants to send.

Implementation: The way this was tested was that we had to check if the Save button in the configuration app actually saves the file in the path that the user selects. So then upon opening the simulator, the user will able to access that .tbc file by clicking on the 'Open an existing talk file' button to select it. To ensure that all of the above was carried out properly, the number of buttons that were opened in the simulator is to be the same as the number of buttons the user filled out in the config app. The test case is successful if the no. of audio buttons created in the config app is the same in the Simulator app on top of the contents that the user had filled into these audio buttons in the config app. This ensures that the connection between the two applications was successful and the serialization object that is serializing and deserializing the files is working as expected.

TalkBox Logger:

- **The FileChooser can only open files of type '.log':**
  Derivation: In order to ensure that the logger doesn't read any files other than the ones that were made my the logger application in the simulator and config apps themselves. The file chooser is only limited to opening files with extension '.log'.

  Implementation: The way this was implemented was using the *Extensions filter* method of the JFileChooser and then limiting the choices to just be .log files. Therefore every time the user gets around to opening the Filechooser for a specific purpose, they can only select files with extension '.log''. The text case is successful if the file the user chose was of the type .log which was done by getting the extension of the file the user chose and checking if matched the string ".log".

  This approach was the way to go because then logger would actually able to properly parse the file without worrying about indexes that are out of bound because the log files are created in a way that respects the indexing of file text.

Testing 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/scenarios they made. It gives them 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 The applications have an average coverage of 76% (Excluding the logger app) and it is a little higher than the average of 70% in traditional software systems which is surely a good sign due to the fact that we were able to reach 75% of the codebase through our test cases and test scenarios resulting in an above average testing coverage.

The config app when it is run gets a test coverage of 78% and it uses all the classes in making sure it works as intended. Depending on the number of error panes the user runs into this percentage can vary as this coverage percentage was arrived by going through the maximum possible number of errors that the user can run into before actually using the software as intended.

The simulator app, on the other hand, gets to a testing coverage of 74% of the entire application. This is almost very similar to the config app due to the fact that these two applications work hand in hand in making the user experience possible. They exchange data through the Serialization object in order to ensure that there is a separation of concern between the two applications.

Last but not the least, the logger app has a very high testing coverage of 95.6%. This is due to the fact that the logger app is inferior in features compared to the config and the simulator app and therefore it is very likely for a user of the logger app to go through what it has to offer resulting in a way above average testing coverage.

The highest coverage classes are obviously the classes that deal with building the GUI as that is where the majority of the code resides. The other helper classes get their fair share of the coverage depending on what the user wants to achieve within the application