Contents

1 Front-End Tests

Many of the front-end tests are implemented using a `ViewBot`, a class that simulates a GUI.

1.1 PresenterTest

These test cases were derived by making one test for each important method of `Presenter`.

- `testConvert()`: Tests that the Presenter's convert method correctly interacts with the input (gets input text & instrument, uses the parser to convert the tab and sets the output correctly). It does not check that the converted MusicXML is correct - that is the backend tests' job.
  It is implemented by first loading a sample input from a file and converting it using the parser to get the expected output. Then, the ViewBot simulates inputting this text and selecting the correct instrument (guitar). Then, the presenter's convert() method is called. Finally, the ViewBot's output text is compared with the expected output.

- `testConvertAndSave()`: Tests that the Presenter's `convertAndSave()` method works properly. It is basically a combination of `testConvert` and `testSaveToFile`. In addition, it tests that the boolean argument of `convertAndSave` works properly.
  It is implemented by loading sample input and issuing a convertAndSave command, similarly to `testConvert`. Then, both the View's output and the test file used as output have their contents checked with the correct text. Like in `testConvert`, this output is generated by the backend code.

- `testLoadFromFile()`: Tests that the "Load from File" command works properly.
  It is implemented by using sample text in a file. A presenter uses its `loadFromFile()` command to load the text into a ViewBot, and the ViewBot's input text is compared with the text that was in the file.

- `testSaveToFile()`: Tests that the "Save to File" command works properly.
  Some text is put in the output of a ViewBot. Then, a Presenter uses its `saveToFile()` command to save the text to a file. The file's text is compared with the text that was inputted in the ViewBot.

This testing is sufficient because every method of the Presenter is covered by a test (except the constructor, which is trivial and has only one line of code). The Presenter's methods are simple enough that only one test is necessary for each.

1.2 PromptingTextAreaTest

These test cases were derived by making one test case for each of the important functionalities of the `PromptingTextArea`: the colour and font of the prompt, the prompt text disappearing when the text box is focused, typing text in the box, and setting the area's font.

- `testPromptColourFont()`: Tests that the text box's colour and font is set properly.
  This test works by creating a PromptingTextArea, and setting custom fonts with `setRegularFont()` and `setPromptFont()`. The prompt is disabled, and the active colour and font is checked for correctness. The prompt is enabled, and the same checks are performed.

- `testPromptFocusChanges()`: Tests that the text box reacts properly to focus changes.
  This test is implemented by simulating gaining and losing focus on the text box, and testing that the text box's text updates correctly.

- `testPromptTyping()`: Tests that the text box reacts properly to typing and when methods are run on it (setText and setPromptText).
  This test works by undergoing several operations (adding and deleting text, gaining and losing focus) while checking the text in the box is correct after each step. Typing and deleting text is simulated using the setText method.

- `testSetFont()`: Tests that fonts are set correctly.
  This test is implemented by disabling the prompt, and setting the font. The regular, prompt and active fonts are checked for correctness. Then, the prompt is enabled and the regular, prompt and active fonts are checked again. The test is repeated, but the prompt starts enabled and is switched to disabled in the second step. This is done because the setFont method behaves differently based on whether the prompt is enabled or disabled.

This testing is sufficient because every method of the PromptingTextArea is tested at least once, and all important or complex methods have tests dedicated to them and their related methods: `testPromptColourFont` tests setRegularFont() and setPromptFont(); `testPromptTyping` tests setText() and setPromptText(); `testSetFont` tests setFont(). All other public methods are simple getters or setters, or methods that trivially call one of the tested methods. In addition, the gain or loss of focus, an important feature of the `PromptingTextArea`, has its own dedicated testing method.

1.3 ViewTest

These tests were derived by making one test for each major method of the View interface.
All tests in this section are run once per View supported by the program, and once for the ViewBot. This ensures that all of the Views support every possible feature. Any test in this section that requires use of an unimplemented optional method is skipped.
All three of these tests are implemented by setting the paramater to some value, then comparing the value set to the value returned by the appropriate get method.

- `testInputText()`: Tests that all of the standard views can correctly get and set their input text.

- `testOutputText()`: Tests that all of the standard views can correctly get and set their output text

- `testInstrumentSelection()`: Tests that all of the standard views can correctly get and set their instrument selection

This testing is sufficient because, like in the Presenter, every important method in the View interface is tested by one test, except `showErrorMessage(String, String)`. The showErrorMessage method cannot be tested automatically (because I do not want to specify **how** an error message is shown, only that one is shown), and it is trivial enough that I am not worried about it breaking (As of the time this document was written, all implementations of this method have only one line of code). The View's methods are also simple enough that only one test per View is needed for each method.

2 Back-End Tests

2.1 ParserTest

- `testTokenizeGuitar()`: Tests that Guitar tablature is split into proper toknens.

- `testInvalidTuneException()`: Tests if the Guitar tablature tune is invalid.

- `testTokenizeDrum()`: [To-Do]

2.2 NoteTest

- `noteTest()`: Tests that the value and index of each of the twelve note types is correct.

- `testToNote()`: Tests that the text box reacts properly to focus changes.

- `testToNoteInvalid()`: Tests that trying to convert an invalid string to a note is invalid.