Simulator Testing Document

Team 1 | York University

EECS 2311 Project

Amir-Hossein Khademi, Kevin Banh, Zaeem Israr

2017

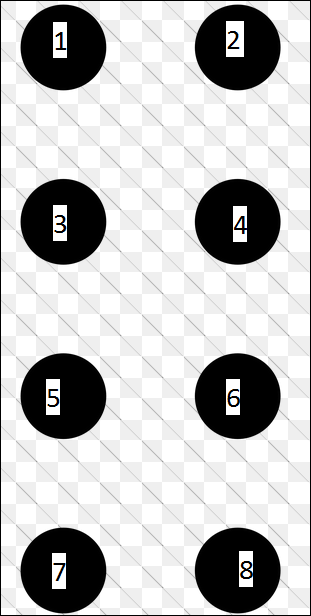
**Introduction**

This document describe the test cases of the Braille Simulator program. Each JUnit class will have their own methods that they will test along with a description for each class. For each method, there will be a description of the method, how the test cases were derived and implemented.

**Definitions**

For the purpose of this document, a “**cell**” is basically a Braille character. They are a rectangular box possibly containing raised bumps on them. These bumps are called “**raised pins**” and based on the number of them and their positioning, represents a character. In the English language, one Braille cell represents one English character. An English character in Braille typically requires six raised pins in a Braille cell. The pins are located in three rows, with two possible locations per row aligned with the other rows. The location where raised pins may occurred will be referred to as a “**pin**”. The hardware has eight pins, four rows with two pins each.

For the purpose of every document and the software, Cell 1 is the leftmost cell, Cell 2 is the one to the right of Cell 1, Cell 3 is right of Cell 2 and so on. In a cell, there are eight pins. Pin 1 is the top left pin, Pin 2 is the top right pin, Pin 3 is below the top left pin and et cetera, denoted by the image below.



Another thing is that the program stores Braille configuration through an array of Strings, with each String containing 8 characters. The characters are either a 1 or a 0 where 1 signifies pin is raised and 0 signifies pin is not raised. The element number in the array of String corresponds with which cell it is describing and the location of the 1s and 0s in the String corresponds with the pin number. For example {“10000110”, “01110000”}, means Cell 1 has a configuration of “10000110”. This means Pin 1, 6 and 7 are raised. For Cell 2, the configuration is “01110000” so Pin 2, 3 and 4 are raised. Refer to image above to see what where each pin number is located on an actual braille cell.

**PinTest.java**

The purpose of this class is to test the Pin class. This class is important because a braille cell consists of multiple pins and each Pin contains a Boolean value called “state” representing whether it is raised or not.

**testSetState()**

**Description**

Tests the “setState(**boolean** state)” method which sets the state of the pin.

**Explanation of Test Cases:**

The purpose of this method is to set the “state” of the pin to the state passed in into the method. This method was tested by passing true and false (all possible states) and ensuring that the state have been set successfully.

**testGetState()**

**Description**

Tests the “getState()” method which returns the “state” of the pin.

**Explanation of Test Cases:**

The test cases covers retrieving all the possible values state can hold besides null. The pin is automatically initialized to false in the constructor so the first usage of getState should return false. It was then set to true so it should return a “true” value.

**ButtonTest.java**

The purpose of this class is to test the Button class. This class contains an integer representing the button number. It will later allow users to activate certain button numbers for the Braille Simulator. (The program will create a certain number of buttons with varying button numbers based on how many number of buttons the user wants to simulate.)

**testGetButtonNum()**

**Description**

Tests the “getButtonNum()” method which returns an integer representing what number is the button. Since there are no ways of changing the button numbers, three different buttons are created for the test cases.

**Explanation of Test Cases:**

The test cases covers when the button number is positive, negative and zero. The method should return what the button number was initially set to from the constructor. Negative and zero values are allowed here in case a programmer wishes to just use the Button API. In the class Braille which aggregates Button, this is not allowed. The test cases here assert that the number retrieved is same as the one set in the constructor.

**CellTest.java**

The purpose of this class is to test the Cell class. This class is important because each braille character is represented by a braille cell (holding, in this case, 8 pins). (This class holds an array of Pins of length 8). This class allows users to set pin configuration, get pin configuration and set a particular pin to a particular state. Please see the last paragraph in **Definitions** to see how an eight-character string of “1s” and “0s” and pin numbers correspond with a pin configuration in real life.

**testSetPinConfig ()**

**Description**

Tests the “setPinConfig(**String** pinConfig)” method which sets the cell’s pin configuration.

**Explanation of Test Cases:**

The test cases tested invalid inputs such as invalid characters in the String pinConfig (not 1 or 0), pinConfig not being the length of 8 and both cases together. It also tested valid cases. In the event of an invalid input, an InvalidInputException would be thrown and the pin configuration of the cell would not change. In the event of a valid input, the pin configuration will be changed accordingly. The test cases covers all three possible cases of invalid inputs and enough valid cases to change the state of all the pins at least once. The test cases assert an InvalidInputException is being thrown (with right error message) and the pin configuration has not been changed in the event of an invalid input and in the event of a valid input, the pin configuration changed accordingly.

**testGetPinConfig()**

**Description**

Tests the “getPinConfig()” method which returns a string representing the cell configuration. (See **Definitions** for more information).

**Explanation of Test Cases:**

The test cases covers getting the initial default pinConfig. The test then uses the setPinConfig to change the pin configuration, using a combination of valid and invalid inputs. If the input was valid, the getPinConfig should retrieve an updated pin configuration while if it is not valid, it will retrieve the last valid pin configuration.

**testSetPin ()**

**Description**

Tests the “setPin (**int** num, **boolean** state)” method which sets a specific pin to a specific state.

**Explanation of Test Cases:**

The test cases covers setting different pin numbers to different states. It also covers the scenario when the pin configuration is not default (i.e. setPinConfig was used) and setting a pin to a state that it is currently is in. It also covers setting invalid Pin numbers (greater than 8 or equal to or less than 0). In the case of a valid change, the tests would assert the pin configuration was changed accordingly while an invalid input would mean the pin configuration would not change and an InvalidInputException would be thrown. In the second case, the test, in addition to checking the pin configuration did not change, would assert that an InvalidInputException is thrown with the right error message.

**BrailleTest.java**

The purpose of this class is to test the Braille class. This class is important because it represents the Braille Simulator without a GUI. This class allows users to translate English to Braille, return pin configuration of all the cells, have a button event, clear all cells and set a particular pin of a particular cell to a particular state. **Note:** Multiple Braille objects have been created as cell and button numbers cannot be changed after being initialized in the constructor.

**testTranslate ()**

**Description**

Tests the “setTranslate(**String** message)” method. The method in question attempts to translate the message into an array of String representing pin configuration. An InvalidInputException is thrown if translation is unsuccessful (i.e String does not contain only alphanumeric symbols or space, or if String size exceeds number of cells available).

**Explanation of Test Cases:**

The test cases covers translating valid inputs of varying sizes such as “C1” and “w” into Braille objects with different amount of cells. It also tests what would happen if a user decides to make the Braille Object have a cell number of 0, which in this case, sets the Braille object to have a cell size of one. In addition to testing random inputs, it also tests all possible valid inputs in terms of characters (i.e. every lower case letter, upper case letter, numbers and space) ensuring everything gets translated successfully. The tests also cover invalid inputs such as invalid characters, String to be translated is too long and both cases together.

In the event of a valid input, the message should be translated successfully and the pin configuration and cell configuration would get updated. In the event of an unsuccessful translate (due to invalid inputs), the tests asserts an InvalidInputException is being thrown with the correct error message and also the pin and cell configuration not being changed.

**testGetCellConfig ()**

**Description**

Tests the “getCellConfig()” method. The method returns an array of String representing the current cell and pin configuration. See more in **Definitions.**

**Explanation of Test Cases:**

The test cases cover retrieving cell and pin configuration of Braille objects with different cell sizes. It also covers the scenario where the cell configuration has been changed (translate) at least one time using a combination of valid and invalid inputs and the case where Braille’s cell and pin configuration are in their default state. If the input was valid, the getCellConfig should retrieve an updated cell and pin configuration while if it is not valid, it will retrieve the last valid cell and pin configuration.

**testButtonEvent()**

**Description**

Tests the “buttonEvent (**int** buttonNum)” method. The method takes in an integer representing which button was pressed and either returns an empty string or a string that tells which button was pressed depending on the value of buttonNum and how many buttons were activated.

**Explanation of Test Cases:**

The test cases cover the cases where the button being pressed is inactive and when it is active. It uses multiple Braille objects with different button sizes (less than 0, greater than 10, and in between those two numbers) to show that at minimum, 0 buttons will be activated and maximum is 10 buttons. The test cases cover every valid button being pressed and the cases where the button pressed would be valid for one object, but not the other Braille object. Valid buttons being “pressed” should not return an empty string.

**testGetNumCells()**

**Description**

Tests the “getNumCells ()” method. The method returns an integer denoting how many cells are activated.

**Explanation of Test Cases:**

The test cases cover the cases where the user attempts to create a Braille object with a valid number of cells and the other case where an “invalid” number is used and the constructor defaults to 1 cell activated. The test cases asserts that the correct number of cells are activated.

**testClearAllCells ()**

**Description**

Tests the “clearAllCells()” method. The method clears all the raised pins in every cell. See more in **Definitions.**

**Explanation of Test Cases:**

The test cases cover the scenarios of clearing the pins off cells that do not have any raised pins and cells that have some pins already set. It also covers Braille objects with different number of cells activated. Testing is simple, assert all the activated cells have a configuration of “00000000”.

**testSetCellPin ()**

**Description**

Tests the “setCellPin(**int** cell, **int** pin, **boolean** state)” method. The method allows a selected pin of a selected cell to be raised or lowered. Throws an InvalidInputException if change is unsuccessful (i.e cell number is less than zero or greater than number of cell activated or pin number is less than 1 or greater than 8).

**Explanation of Test Cases:**

The test cases covers setting different pin numbers of different cells to different states. It also covers the scenario when the pin configuration is not default (i.e. translate was used), setting a pin to a state that it is currently is in and Braille objects with different cell sizes. It also covers invalid Pin numbers inputs (greater than 8 or equal to or less than 0) and invalid cell number inputs. In the case of a valid change, the tests would assert the pin and cell configuration was changed accordingly while an invalid input would mean the pin and cell configuration would not change and an InvalidInputException would be thrown. In the second case, the test, in addition to checking the pin and cell configuration did not change, would assert that an InvalidInputException is thrown with the right error message.

**BrailleClientTest.java**

BrailleClient.java extends Braille.java and creates a GUI with the give parameters; those being the number of buttons and the number of cells. The parameters can be entered through the GUI or through the code. Some of the methods here are only tested through the code (i.e. translate method where something is translated). It is simply not feasible to use a Robot class to type in the text box of the GUI and hit the translate button. Others like the button events are tested both in the code and using the Robot class to press keyboard buttons.

**testTranslate ()**

**Description**

Tests the “setTranslate(**String** message)” method. This method attempts to translate the message into an array of Strings representing the pin configuration; each String being 8 characters long and consisting of 1s and 0s. It updates the GUI accordingly with red and black rectangles according to what the string message is.

**Explanation of Test Cases:**

Running the test creates a GUI which then translates and displays the message accordingly. A robot class is then used to take a screenshot and verify that the correct message was translated by comparing the colour of pixels on the screenshot. It also verifies different sizes such as a braille with a cell number of 0, which in this case, sets the Braille object to have a cell size of one. In addition to testing random inputs and verifying the pixels, it also tests all possible valid inputs in terms of characters (i.e. every lower case letter, upper case letter, numbers and space) ensuring everything is getting translated correctly.

In the event of a valid input, the message should be translated and shown successfully in the GUI. In the event of an unsuccessful translate (due to invalid inputs), the test throws an InvalidInput Exception and also ensures that the state of the cells remains unchanged.

**testButtonEvent()**

**Description**

Tests the “buttonEvent (**int** buttonNum)” method. The method takes in an integer representing which button was pressed and either returns an empty string or a string that tells which button was pressed depending on the value of buttonNum and how many buttons were activated.

**Explanation of Test Cases:**

The test cases cover the cases where the button being pressed is inactive and when it is active. In this case again robot was used to keyPress and keyRelease the buttons used and compare it to the expected key press to ensure the right button was clicked.

**testClearAllCells ()**

**Description**

Tests the “clearAllCells()” method. The method clears all the raised pins in every cell so it is showing every cell as a red square in the GUI.

**Explanation of Test Cases:**

The test cases work by translating different sizes of strings and then clearing all the translated cells. Again a robot will take a screen shot of the GUI and compare the pixels with an expected pixel to verify that the cells are being successfully clear.

**testSetCellPin ()**

**Description**

Tests the “setCellPin(**int** cell, **int** pin, **boolean** state)” method. The method allows a selected pin of a selected cell to be raised or lowered. Raising it will make the chosen cell and pin to black while making it red if lowered on the GUI.

**Explanation of the test case:**

The test cases cover setting different pin numbers of different cells to different states. After doing so a robot will take a screen shot of the GUI then compare the pixels of the pins on each cell to an expected pixel verifying that it is being done successfully. It also covers the scenario when the pin configuration is not default (i.e. translate was used), sets the pins accordingly and repeats the process to ensure the pin was set successfully. It also covers invalid Pin numbers inputs (greater than 8 or equal to or less than 0) and invalid cell number inputs. In the case of a valid change, the tests would assert the pin and cell configuration was changed accordingly while an invalid input would mean the pin and cell configuration would not change and an InvalidInputException would be thrown. In the second case, the test, in addition to checking the pin and cell configuration did not change, would assert an InvalidInputException is thrown with the right error message.

**Sufficient Coverage**

While it is explained above why the test cases are sufficient (covers most cases for each method), EclEmma was also used to provide a quantitative answer. Running the Test Suite, notice how all publically available classes (part of API) all have 100% coverage excluding Braille which has 99.7%. Overall the project has approximately 88.3% coverage which is pretty good.

The reason why it is not higher is because EclEmma included the client code which plays no role in the final product. Another reason is that in some of the test classes like CellTest, there were methods that had invalid inputs and therefore would throw an exception. The method calls would be counted as “missed” even though it is intended. Finally, some classes involving the GUI were too difficult to test.

