Test Cases
Braille Application

Paul Sison, Jonas Laya, Samuel On, Jamie Dishy

Overview

The following text outlines test cases that have been implemented based on classes BrailleCell as well as VisualPlayer in order to ensure that the programming within the given classes is sufficient. It will further discuss how these test cases were derived along with the overall testing coverage according to the program EclEmma.

JUnit Test Class: BrailleCellTest

The BrailleCellTest class tests ten methods based on the BrailleCell class to ensure that methods operate as expected. EclEmma, a program that predicts the sufficiency of completed tests, approximates that these ten tests are 88.6% complete.

testGetNumberOfPins()

Given that the braille cells currently has 8 pins, this test ensures that the method returns a length of 8 pins.

testLowerOnePin()

This test first ensures that the input argument is an integer between and including 0 until 7; otherwise, it will throw an *IllegalArgumentException*. Next, it tests that the particular pin number corresponding to the integer input is lowered when specified. This test case ensures that the pin is deemed 'false' to signify that it is lowered as instructed.

test05_RaiseOnePin()

Based on the method *RaiseOnePin( )* in BrailleCell, this test ensures that an *IllegalArgumentException* is thrown when the input integer that signifies the pin to be raised on the braille cell is out of bounds. This involves any integer outside of the range from 0 to 7. In this example, the specified pin to be raised is associated with integer 8, which is out of bounds. As such, the message returned is "Invalid index," as anticipated.

test06_RaiseOnePin()

Based on the method *RaiseOnePin()* in BrailleCell, this test ensures that any pin which correlates to an integer between and including 0 to 7 is raised and is further associated with 'true' to signify its raised status.

testClear()

This test ensures that all of the pins in the cell are lowered when the method “clear()” is used. The approach is to ensure that all of the pins associated with integers between and including 0 to 7 are coined with the term ‘false’ to affirm that they are lowered.

test03_SetPinsEx1()

Based on the method *setPins(String pins)* in class BrailleCell, this test case ensures that the number of pins listed strictly sum to 8. This is done using a try-catch block, where if one enters a number of pins summing to 11, an *IllegalArgumentException* will occur along with the specified message explaining that the string passed is either smaller or greater than 8.

Test04_SetPinsEx2()

Based on the method *setPins(String pins)* in class BrailleCell, this test case ensures that the string of pins listed is strictly comprised of binary numbers 1 and 0. This is done using a try-catch block, as any numbers other than 1 and 0 listed in the string result in an *IllegalArgumentException* along with a messaged stating that the string passed is invalid.

testGetPinState()

This test ensures that the correct value is coined with the pin given its state. In this case, if the fourth pin is to be raised, as specified by the integer 4, then that pin will be associated with ‘true’ to signify that it has been raised.

test01_DisplayCharacterEx()

This test ensures that the user input is a character strictly found in the English alphabet. It does so by using a try-catch block, where a symbol that s not found in the English alphabet, ‘$’, is input. This results in an *IllegalArgumentException* along with a message stating that this character is not standard.

Test02_DisplayCharacter_SetPins()

This test ensures that the given alphabetical letter correlates to the correct pins elevated or lowered on the braille cell. To do so, it first asserts the states of each of the 8 pins that correlate to letter ‘A’. It then proceeds with a for-loop and asserts that the given indices between 0 and 7 correlate with either term ‘true’ or ‘false’. If the term is ‘true’, this means that the pin will be elevated on the braille, and vice-versa if ‘false’. The overall result on the braille cell will represent the letter ‘A’ and thus proves that the method *DisplayCharacter(char a)* is sufficient.

JUnit Test Class: VisualPlayerTest

The VisualPlayerTest class tests 8 methods based on the *VisualPlayer* class to ensure that methods operate as expected. Given 8/8 runs, there are 5 sufficient test methods, along with 2 errors and 1 failure. EclEmma, a program that predicts the sufficiency of completed tests, approximates that these ten tests are 70.7% complete.

testSetButton()

This test ensures that the user input to set a given button must be a positive integer. To test for this, a try-catch block is used. Given a user input of -1, an *IllegalArgumentException* occurs along with a message stating that a non-positive integer was entered.

testGetButton()

This test ensures that user integer input that represents the index of a given button is smaller than 'buttonNumber' and is also positive. To do so, a try-catch block is first used to present an *IllegalArgumentException* when index 9 is input, along with a message stating that the button index is not valid. Further, another try-catch block is used to present an *IllegalArgumentException* when negative index -1 is input, along with a message stating that the button index is also not valid. Based on these tests, the method *GetButton* is sufficient to perform the anticipated tasks.

testSetCell()

This tests that the user integer input to modify the cell parameter is positive. Using a try-catch block, when a user inputs, for example, negative integer -1, an *IllegalArgumentException* appears along with a message stating that the input is non-positive and thus is not valid.

testRemoveButtonListener()

The purpose of this test is to remove the ActionListener given user integer input that is smaller that the button number and greater than zero. This test involves three try-catch blocks that apply for separate situations. The first try-catch block is meant to pause execution by 4 seconds for the *runnable* to catch up prior to calling other methods. This occurs because the *runnable* is not finished invoking by the time other methods take place. Next, the second try-catch block passes an *IllegalArgumentException* when the user input is 9 because it goes beyond the boundaries. Lastly, the third try-catch block passes an *IllegalArgumentException* when the user input is negative, such as -1. This test finishes by adding to

the array ActionListener, followed by removing it, and completes by showing that the ActionListener ultimately has a length of 0.