Player Test

Testing Document

Team 4

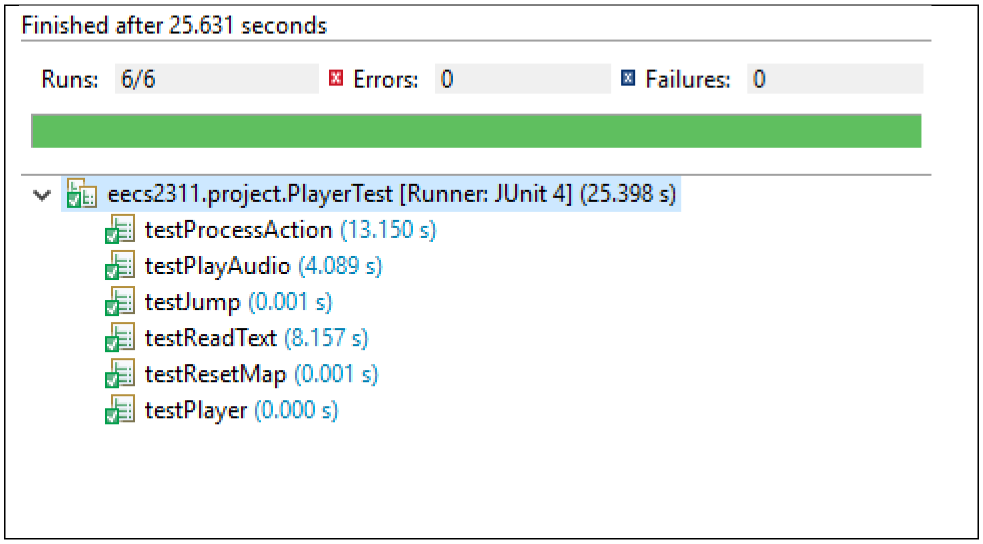
Allauddin, Qassim
Li, Derek
Mohamed, Yassin
Solovey, Artem

EECS 2311 - Software Development Project (Winter 2017)
Instructor: Bil Tzerpos

Table of Contents

1. Introduction

Testing was a critical part in the process of improving the code for the project as well as fixing bugs that were in the code. The tests were done on the Player. This class contains test cases that cover the sufficient inputs. Figure 1 shows all test cases passed. These tests were sufficient because they test most of the coverage and they test almost every scenario.



```
Finished after 25.631 seconds

Runs: 6/6    Errors: 0    Failures: 0

eecs2311.project.PlayerTest [Runner: JUnit 4] (25.398 s)
    testProcessAction (13.150 s)
    testPlayAudio (4.089 s)
    testJump (0.001 s)
    testReadText (8.157 s)
    testResetMap (0.001 s)
    testPlayer (0.000 s)
```

Figure 1: Tests all the cases.

2. Testing Player

We have a rule:

```
    public ExpectedException exceptionRule = ExpectedException.none();
```

Before the class starts we do:

```
    public static void setUpBeforeClass() throws Exception {
              p1 = new Player(new File("library.txt"));
```

Also before the tests we do:

```
    public void setUp() throws Exception {
              scan = new Scanner(new File("test.txt"));
        }
```

2.1 testPlayer()

The testPlayer throws an exception if the file is not found. It then creates a new player with a file nofile.txt. An example is shown in figure 2.

```
	public void testPlayer() throws Exception {
		exceptionRule.expect(FileNotFoundException.class);
		Player p2 = new Player(new File("nofile.txt"));
		}
```

Figure 2: shows the testPlayer method.

2.2 testPlayAudio()

This method throws an exception. It first tests if the file is a .mp3 or a .wav. This is shown in figure 3. It then tests if the file exists in the project, this is shown in figure 4. Finally it tests if the audio file is unsupported and this is shown in figure 5.

```
	p1.testFlag = false;
		scan.findWithinHorizon("<audio3>\n", 0);
		p1.playAudio(scan);
			assertTrue(p1.testFlag);
```

Figure 3: tests if the file in the project is .mp3 or .wav.

```
	scan.findWithinHorizon("<audio1>\n", 0);
		exceptionRule.expect(FileNotFoundException.class);
			p1.playAudio(scan);
```

Figure 4: tests if the file exists in the project.

```
	scan.findWithinHorizon("<audio2>\n", 0);
		exceptionRule.expect(UnsupportedAudioFileException.class);
			p1.playAudio(scan);
```

Figure 5: tests if an unsupported file is in the project folder.

2.3 testReadText()

This method throws an exception. It first tests when the method reads text using the speak() method, this is shown in figure 6. It then tests when the method does not read the text using the speak() method, this is shown in figure 7.

```
        scan.findWithinHorizon("<readText1>\n", 0);
                        assertTrue("TTS method should run for letters,
        sentences, digits, and numbers", p1.testFlag);
```

Figure 6: tests when the method reads texts using the speak method.

```
        scan.findWithinHorizon("<readText2>\n", 0);
                p1.readText(scan);
                assertFalse("TTS method should not run", p1.testFlag);
```

Figure 7: test when the method does not read text using the speak method.

2.4 testActionPerformed()

We did not need to test this method, that is why we ignore it. We are testing this method when we are testing process action. More specifically when a button is clicked.

2.5 testResetMap()

This method uses a loop to set all the buttons to false in a hashmap, this is shown in figure 8.

```
        p1.resetMap();
                int i;
                for (i = 0; i < p1.buttonNumber - 1; i++) {
                        boolean button =
p1.buttonMap.get(p1.simulator.getButton(i));
                        assertFalse("Button should be false.", button);
                        }
```

Figure 8: sets all the buttons to false in the hashmap.

2.6 testProcessAction()

This method throws the index out of bound exception. If the correct button is pressed it will take us inside the <cs> tag in the txt file, otherwise it will go to the <cf> tag. It also tests when the button is clicked. We first initialise the method shown in figure 9.

```
        scan.findWithinHorizon("<case1>\n", 0);
                p1.readText(scan);
                p1.processAction(scan);
                        scan.nextLine();
```

Figure 9: initialisation of the class.

In out test button 1 is the correct answer. When the correct button is pressed the processAction sends the scanner to the correct <cs> tag, this is shown in figure 10.

```
        if (correctBtn == true) {
                        assertTrue(correctBtn);
                        assertEquals("Correct", scan.nextLine());
                        p1.readText(scan);
                        p1.processAction(scan);
                        scan.nextLine();
                        correctBtn = 
p1.buttonMap.get(p1.simulator.getButton(0));
                        if (correctBtn == false) {
                                assertFalse(correctBtn);
                                assertEquals("Fail", scan.nextLine());
                        } else {
                                fail("Must press 2 to test fail");
                        }
                } else {
                        fail("Must press 1 to test success");
                        }
```

Figure 10: The correct button should be true, in the scenario the .txt file the next line should be “correct”. It then skips the new line that was left in the scanner from the last call, and then the correct button should be false.

The scenario .txt file sets the correct button to ‘100’ which is out of bounds and tests the exception. This is show in figure 11.

```
        exceptionRule.expect(IllegalArgumentException.class
);
                        p1.processAction(scan);
```

Figure 11: tests the exception.

2.7 testJump()

This method throws the run time exception. This method read he next line after tag, then it looks forward until it finds the same line and then it reads the line after. In other words, it positions the scanner to the line after the tag after it looks through the file. This is show in figure 12.

```
        scan.findWithinHorizon("<jump_test>\n", 0);
                p1.jump(scan);
                assertEquals("<correct>", scan.nextLine());
                scan.reset();
                scan.findWithinHorizon("<jump_test2>\n", 0);
                exceptionRule.expect(IllegalArgumentException.class);
                        p1.jump(scan);
```

Figure 12: If the scanner did not jump properly the next line would be incorrect. It will also fail if the method throws an exception. After resetting the scanner, it then tests if the exception is thrown when it jumps to a line that does not exist.