EECS 2311

Authoring Application

Testing Document

Rajvi Chavada

Jinho Hwang

Andrew Maywapersaud

Table of Contents

List of Figures

1.0 Test Cases

The following tests were created using Junit to test the functionality of the authoring application.

Test number	Test Name	Result (pass/fail)
1	Test Phrase Parsing	Pass
2	Test Phrase Non-Integer Parsing	Pass
3	Test Scenario 1	Pass
4	Test Scenario 2	Pass
5	Test Scenario 3	Pass

Figure 1 illustrates the success of the tests when executed in a Junit testing framework.

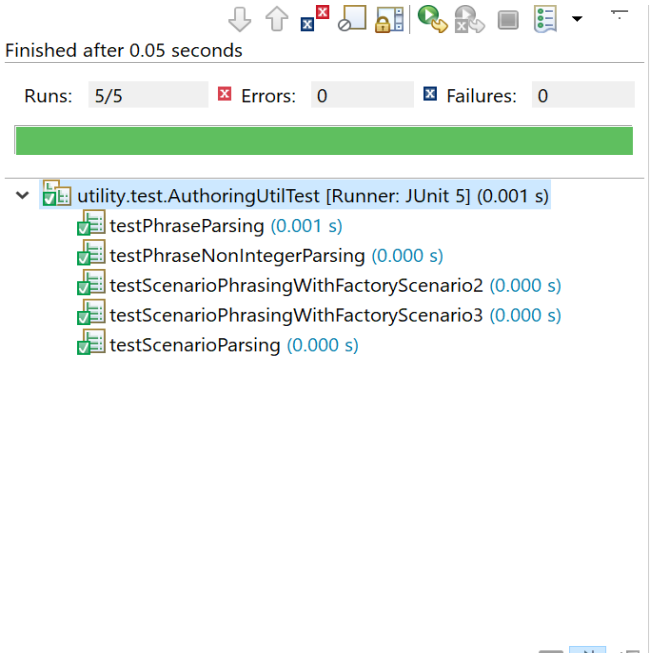
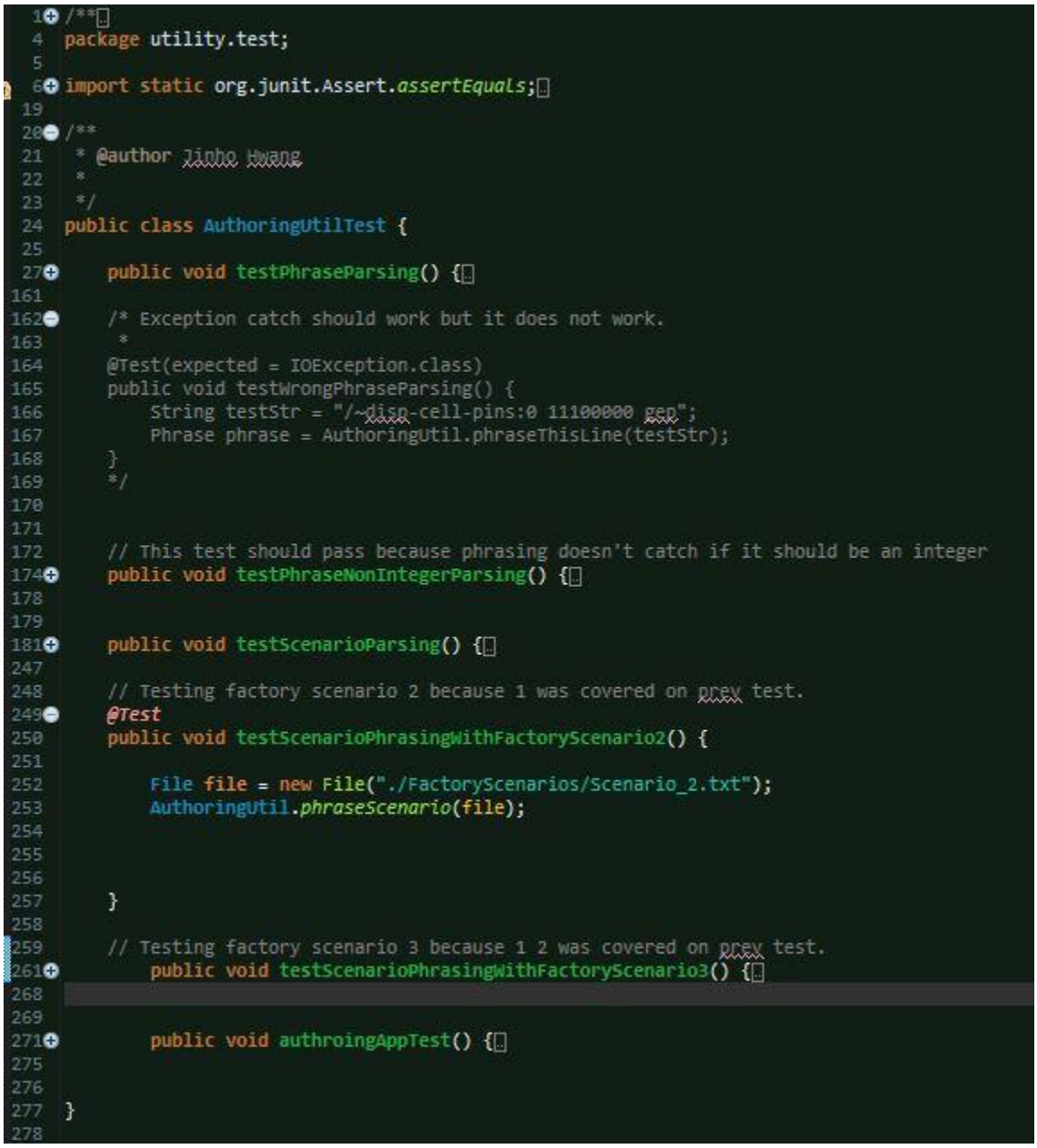


Fig.1 Test Case Success

Figure 2 highlights the test class.



```
/**
package utility.test;

import static org.junit.Assert.assertEquals;

/**
 * @author Jinho Hwang
 *
 */
public class AuthoringUtilTest {

    public void testPhraseParsing() {

    /* Exception catch should work but it does not work.
     *
    @Test(expected = IOException.class)
    public void testWrongPhraseParsing() {
        String testStr = "/~disp-cell-pins:0 11100000 gep";
        Phrase phrase = AuthoringUtil.phraseThisLine(testStr);
    }
    */


    // This test should pass because phrasing doesn't catch if it should be an integer
    public void testPhraseNonIntegerParsing() {


    public void testScenarioParsing() {

    // Testing factory scenario 2 because 1 was covered on prev test.
    @Test
    public void testScenarioPhrasingWithFactoryScenario2() {

        File file = new File("./FactoryScenarios/Scenario_2.txt");
        AuthoringUtil.phraseScenario(file);



    }

    // Testing factory scenario 3 because 1 2 was covered on prev test.
        public void testScenarioPhrasingWithFactoryScenario3() {


        public void authroingAppTest() {


}
```

Fig.2 Tests Ran

2.0 Description of Test Cases

1.) Test Phrase Parsing

This test was created to detect an error in phrasing. The error being thrown is known. Therefore, if the test passes, we can feel confident that the error is being caught. The error in question is a parsing error. A parsing error will result in incorrect use cases.

2.) *Test Phrase Non-Integer Parsing*

This test validates that the current parsing method, when creating a skip button phrase in a scenario, does not check if the command used the correct syntax of using an integer value after the word button.

Tests 3 to 5.) *Scenario Tests*

These tests ensure that the scenarios being loaded in the scenario editor and displayed in the visual player are being properly loaded.

3.0 Test Case Efficiency

Relative to the current release of the application, these tests suffice as the main functionality of the application is to parse scenario files and display them on the visual player.

For instance, when you launch the application and go to scenario editor, there are three scenario text files that can be executed and interacted with on the simulated braille cells. Moreover, you can create your own scenarios and have them parsed and read.

Our test cases check if the syntax used in creating commands is correct. By extension, any commands created from the correct syntax will be valid under the same constraints. Therefore, there will be no parsing errors.

The rest of the functionality, such as creating audio files, loading files, running files and removing files can be deemed efficient by using the modes and noting if they act as intended or not. For example, when implementing the voice recorder, we simply launch the application and check if it recorded and saved an audio file. These actions occur when specific events occur thus we fire these events and observe what happens. In contrast, actions that require a specific algorithm or process to execute such as parsing a scenario cannot be tested or relied on via empirical evidence and needs to be unit tested.

4.0 Coverage

First, it must be noted that our coverage is diluted due to the creation of utility classes that we currently use throughout the project. These utility classes perform operations that we tend to use multiple times and thus save time in coding. The utility classes are not tested and thus do not contribute to coverage. However, they contribute to the overall scope of the project which skews the overall coverage view. These classes can be found in the utility package.

Figure 3 highlights the current coverage of our project calculated by installing the EClemma plugin.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Enamel	58.5 %	4,620	3,280	7,900
src	58.5 %	4,620	3,280	7,900
enamel	36.4 %	972	1,697	2,669
ScenarioParser.java	14.5 %	181	1,071	1,252
ScenarioParser	14.5 %	181	1,071	1,252
VisualPlayer.java	57.0 %	273	206	479
VisualPlayer	37.2 %	83	140	223
AudioPlayer.java	57.6 %	265	195	460
AudioPlayer	41.1 %	78	112	190
BrailleCell.java	57.7 %	177	130	307
BrailleCell	57.7 %	177	130	307
Player.java	44.4 %	76	95	171
Player	46.9 %	61	69	130
utility	54.0 %	1,108	942	2,050
AuthoringUtil.java	49.3 %	827	851	1,678
AuthoringUtil	49.3 %	827	851	1,678
Phrase.java	72.0 %	226	88	314
Phrase	72.0 %	226	88	314
Language.java	94.8 %	55	3	58
Language	94.8 %	55	3	58
gui.controllers	79.8 %	1,828	464	2,292
CreateCommandPopUpBoxControlle	81.4 %	443	101	544
ScenarioEditorController.java	80.6 %	416	100	516
ScenarioMakerController.java	82.8 %	439	91	530
VoiceRecorderController.java	83.1 %	399	81	480
MainMenuController.java	38.6 %	22	35	57
TwoChoiceBoxController.java	60.0 %	42	28	70
ErrorListReportPopUpBoxController.ja	76.7 %	46	14	60
TextAnswerBoxController.java	60.0 %	21	14	35
startProgram	0.0 %	0	75	75
AuthoringApp.java	0.0 %	0	75	75
gui.layouts	91.1 %	658	64	722
ScenarioMaker.java	92.7 %	139	11	150
VoiceRecorder.java	85.9 %	67	11	78
ScenarioEditor.java	85.9 %	61	10	71
CreateCommandPopUpBox.java	93.0 %	93	7	100
ErrorListReportPopUpBox.java	91.2 %	73	7	80
TextAnswerBox.java	90.5 %	67	7	74
TwoChoiceBox.java	93.2 %	82	6	88
MainMenu.java	93.8 %	76	5	81
utility.test	60.7 %	54	35	89
AuthoringUtilTest.java	60.7 %	54	35	89

Figure 3. Test Coverage

According to the coverage, approximately 59% of the code is covered.