

# Testing plan

## Objective

---

The objective is to test the code that were implemented the last iteration. The testing of this application will be done using both manual testing and unit testing. For the unit testing, we will use JUnit 5 and create our individual test suite. With these unit tests, we will test the classes and their functions in isolation. The manual testing will test the functionality by simply making sure that the application behaves as intended. By manual testing, we will test out how the parts interact with each other and make sure that the game functions in a predictable manner.

## What to test and how.

---

We intend to test the use cases for this application by manual testing and then test the components in isolation by using unit tests.

## Time plan

---

Task	Estimated	Actual
Manual TC	60m	55m
Unit tests	40m	50m
Running manual tests	30m	15m
Code inspection	45m	30m
Test report	60m	80m

## Manual Test-Cases

---

### TC1.1 Start the game

**Use case:** Start Game

**Scenario:** Successfully start the application and the application will display the start menu.

**Precondition:** None

**Test steps**

- Start the app
- System displays the start menu

### Expected

- The system should print the start menu with multiple options

### Motivation / Notes

This is the most basic test case. This is just to make sure that the application does start without any errors and/or complication. Application starts without any issue.

## TC1.2 Quit the game

**Use case:** Quit game

**Scenario:** Successfully terminate the application without errors.

**Precondition:** None

### Test steps

- Start up the application
- System displays the start menu
- Choose “Quit” as menu option.

## Expected

- The system should terminate without any issue.

## Motivation / Notes

This test ensures that the application can successfully be terminated without any data loss or any errors.

## TC2.1 Complete a game of Hangman

**Use case:** Play game

**Scenario:** Successfully start a new game and correctly guess the word.

The main scenario of UC2 will be tested by running a game of “hangman”.

**Precondition:** None

### Test steps

- Start the app
- Play a game where we guess the correct word.
- Enter our name if we get a better score than the current, and afterwards we will be given the menu options.

## Expected

- The system should print the gallows correctly.
- If we beat the current score, we will be prompted to enter our name for the highscores.

## Motivation / Notes

This is the most important test, as this is the main functionality of this application. Here we will do a complete game of hangman and guess the correct word.

## TC2.2 Guess an incorrect word

**Use case:** UC1 Start new game

**Scenario:** Start a new game of hangman and fail to guess the correct word.

**Precondition:** None

### Test steps

- Start the app
- Choose the “New game” menu option.
- Play a game where we guess the incorrect word 8 times.

### Expected

- The system should show the text for failing to guess the correct word, show the whole gallows and show the menu options.

### Motivation / Notes

This test makes sure that we we get the correct gallows build up as well as get the correct screen when you lose the game.

## TC2.3 Return to the main menu

**Use case:** UC2 Play Game

**Scenario:** Cancel a game of hangman and return to the main menu

**Precondition:** None

### Test steps

- Start the app
- Choose the “start new game” menu option.
- When game has started, enter “00” to go back to the main menu

### Expected

- The system should cancel the game of hangman and display the main menu.

### Motivation / Notes

This was modeled in the previous iteration, and we should have a test to ensure that this part of the system is working as intended.

## TC3.1 Show the high score list

**Use case:** UC3 View High score

**Scenario:** Use the “show highscores” menu option.

**Precondition:** None

### Test steps

- Start the app
- Press “2” to enter the highscore list.

### Expected

- The system should show a sorted list with the current highscores. The application should then return us to the main menu.

### Motivation / Notes

This test is to ensure that we can successfully view the high score list and by that solidifying that we complete the requirement that is the “View High Score” use case.

## Test Report

---

### Test traceability matrix and success

Test	UC1	UC 2	UC 3	UC 4
TC1.1	OK	-	-	
TC1.2	OK	-	-	OK
TC2.1	OK	OK	-	-
TC2.2	OK	OK	-	-
TC2.3	OK	OK	-	-
TC3.1	OK	-	OK	-
COVERAGE & SUCCESS	OK	OK	OK	OK

### Automated unit test coverage and success

Test	HighScore	WordList
HighScoreTest	2 / 2 OK	-
WordListTest	-	3 / 4 OK

## Comment

Testing concludes that the system behaves as expected.



Chris

## HighScoreTest

Since this is an important part of the game, I made sure to test that the persistence and the adding of new names is working as intended. So we are focusing on testing the “addEntry()” method.

This is called in the source code in the “central controller” class in the main “play game” loop.

```
if (guesses < 8) {
    int Score = (int)(System.currentTimeMillis() - timer)/1000;
    System.out.println(Score);
    int position = hs.checkForNewHighscore(Score);
    if(position < 5) {
        view.printString("New Highscore! Please enter your name.");
        String name = scanner.next();
        //System.out.println("Trying to score at position : "+position);
        hs.AddEntry(name, Score, position);
    }
}
```

We focused on two cases. Firstly, we tested just adding adding the name and score on the last place (5th place). Secondly, we tested adding the player to the first place with a better record, then all the players in the list should be pushed down one space.

```
@Test
public void test() throws IOException {
    /*
     * This test is designed to test the AddEntry function. Here we add a new score to the high score, and we make sure that the List
     * should be updated accordingly.
     */
    try {
        System.out.println("Starting first test");
        HighScore hs = new HighScore();
        String[] strings = hs.getHighScores();
        System.out.println("----- Old highScores");
        for (int i = 0; i < strings.length; i++) {
            System.out.println(strings[i]);
        }
        //assertEquals("fisk", strings[0]);
        hs.AddEntry("TestGuy", 5, 4);
        hs = new HighScore();
        strings = hs.getHighScores();

        System.out.println("----- New highScores");
        for (int i = 0; i < strings.length; i++) {
            System.out.println(strings[i]);
        }

        assertTrue(strings[4].equalsIgnoreCase("TestGuy : 5"));

    } catch (FileNotFoundException | UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Chris

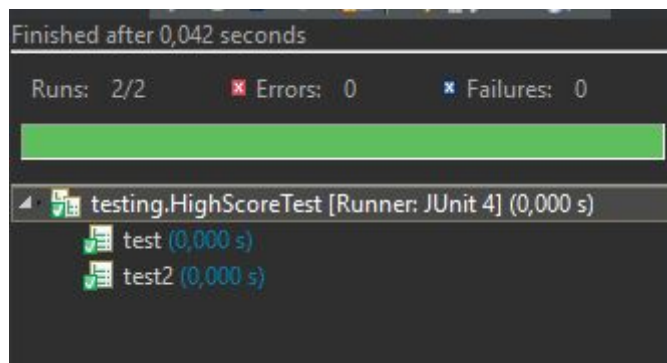
```
@Test
public void test2() throws IOException {
    /*
     * This test will try to add an entry at the top, and we will try to push all the other entries on the list.
     * We start up by manually setting the highScore to what we wish.
     */
    try {
        System.out.println("Starting Second test");
        HighScore hs = new HighScore();

        for (int i = 0; i < 5; i++) {
            hs.AddEntry("player "+i, 90+i, i);
        }
        String[] strings = hs.getHighScores();
        System.out.println("----- Old highScores");
        for (int i = 0; i < strings.length; i++) {
            System.out.println(strings[i]);
        }

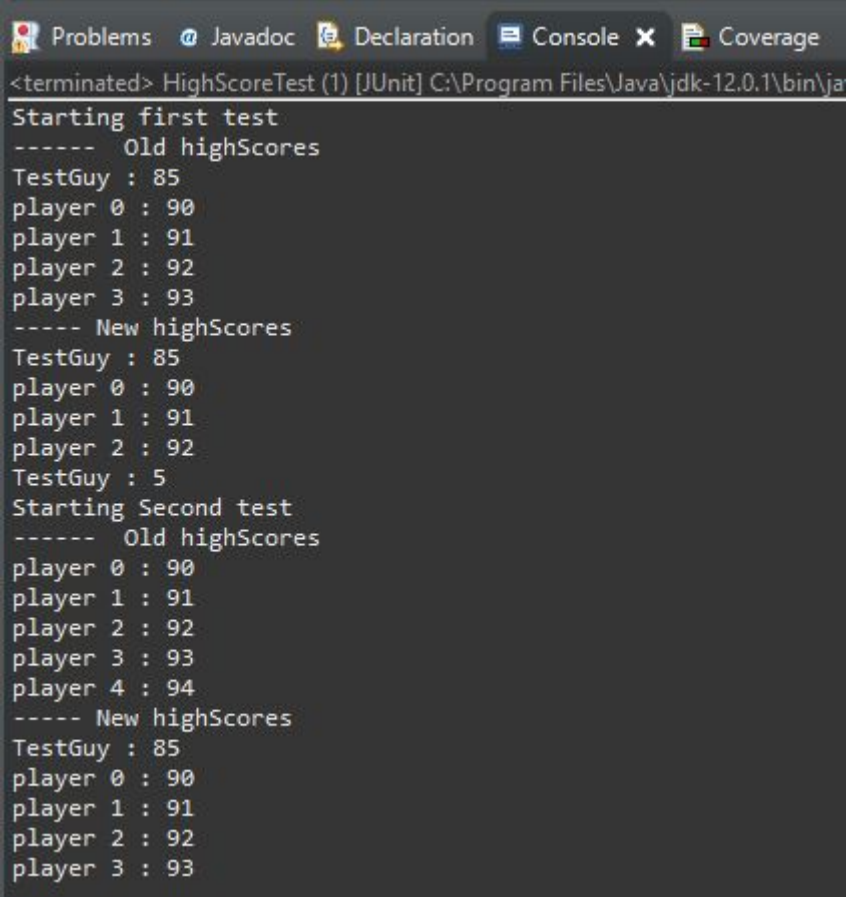
        hs.AddEntry("TestGuy", 85, 0);
        System.out.println("----- New highScores");
        hs = new HighScore();
        strings = hs.getHighScores();
        for (int i = 0; i < strings.length; i++) {
            System.out.println(strings[i]);
        }

        assertTrue(strings[4].equalsIgnoreCase("player 3 : 93"));
    } catch (FileNotFoundException | UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Here are the results.



And the console prints look like this.



```
Problems  Javadoc  Declaration  Console  Coverage
<terminated> HighScoreTest (1) [JUnit] C:\Program Files\Java\jdk-12.0.1\bin\ja
Starting first test
----- Old highScores
TestGuy : 85
player 0 : 90
player 1 : 91
player 2 : 92
player 3 : 93
----- New highScores
TestGuy : 85
player 0 : 90
player 1 : 91
player 2 : 92
TestGuy : 5
Starting Second test
----- Old highScores
player 0 : 90
player 1 : 91
player 2 : 92
player 3 : 93
player 4 : 94
----- New highScores
TestGuy : 85
player 0 : 90
player 1 : 91
player 2 : 92
player 3 : 93
```

## Notes

By doing these two tests, I realized that we are actually testing the persistence of the high scores as well, as in both tests we re-instantiate the high score class after the adding of new names. This means that I will not have to do separate tests for this functionality.

## WordListTest

For this part, we intend to test 2 functions. We will test the most important functionality of this part, which is the `randomWord()` method. The two cases we will try is to see if the words are truly random by checking if we are able to get the same word twice, and also if we are able to get different words. Lastly, I added two tests for an up and coming functionality, which handles the adding of new words to the application.

The `randomWord()` can be found in the `NewGame()` in the central controller class, and it is a central part of the game, and therefore should be tested.

```
public void NewGame() {
    view.printString("Good Luck! enter 00 to go back to main menu.");
    view.Gallows(0);
    word = list.randomWord();
    hidden = new boolean[word.length()];
    view.printString(BuildCorrectString(hidden));
    guesses = 0;
}
```

Here we have the tests.

```
@Test
public void testRandomWord() {
    /*
     * This test verifies that we can get random words.
     * In this test we want to make sure that we can get the same word multiple times.
     */
    WordList wl = new WordList();
    String word = wl.randomWord();
    String randomWord = wl.randomWord();
    for (int i = 0; i < 1000; i++) {
        randomWord = wl.randomWord();
        if (randomWord.equalsIgnoreCase(word)) {
            break;
        }
    }

    assertTrue(randomWord.equalsIgnoreCase(word));
}
```

```

@Test
public void testRandomWord2() {
    /*
     * This test is to verify that we can get random words and
     * not just the same word all over.
     */
    WordList wl = new WordList();
    String firstWord = wl.randomWord();
    String secondWord = wl.randomWord();

    System.out.println("The first word is : "+firstWord);

    boolean answer = true;
    for (int i = 0; i < 100; i++) {
        if( !firstWord.equalsIgnoreCase(secondWord)) {
            answer = false;
            break;
        }
        secondWord = wl.randomWord();
    }
    System.out.println("The second word is : "+secondWord);

    assertFalse(answer);
}

```

Problems Javadoc Declaration Console X Coverage

<terminated> WordListTest (1) [JUnit] C:\Program Files\Java\jdk-12.0.1\bin\javaw.exe (22 aug. 2019 12:39:01)

The first word is : bacon

The second word is : ostkaka

And lastly, we have the two tests for the new functionality. This has not been implemented to the system yet, since it is not working as intended yet.

```

@Test
public void testAddWord() {
    /*
     * This test is to ensure that we can add words to our wordlist
     * and have the chance to get them as a random word.
     * This test could fail due to the word being random however,
     * the chances are very slim that this will occur with the low amount of possible words
     * and the amount of attempts to get the new word.
     */
    WordList wl = new WordList();
    wl.addWord("testWord");
    String randomWord = wl.randomWord();
    for (int i = 0; i < 1000; i++) {
        randomWord = wl.randomWord();
        if (randomWord.equalsIgnoreCase("testword")) {
            break;
        }
    }

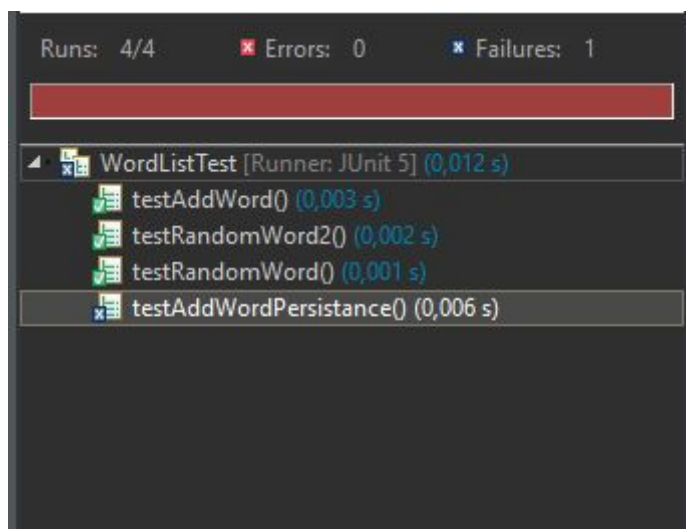
    assertTrue(randomWord.equalsIgnoreCase("testword"));
}

```

```
@Test
public void testAddWordPersistance() {
    /*
     * This test is to ensure that the words we add stay
     * even after we shut down the application.
     */
    WordList wl = new WordList();
    wl.addWord("testWord");

    wl = new WordList(); // re-initializing the WordList.

    String randomWord = wl.randomWord();
    for (int i = 0; i < 1000; i++) {
        randomWord = wl.randomWord();
        if (randomWord.equalsIgnoreCase("testword")) {
            break;
        }
    }
    assertTrue(randomWord.equalsIgnoreCase("testword"));
}
```



## Notes

As we can see, the application behaves as expected except for the new functionality. Adding words seem to be working, but there is an issue with the persistence. This means that I might have to re-write large parts of the WordList class to fit this new functionality.

## Reflection

This actually took a lot more time than I had anticipated, as I had taken some of the functionality for granted, which means that I did not do test cases for these at first. Since some of these are use cases, I figure I had to ensure that they are working as intended, and therefore they also require testing as that is the receipt that they are functional. The takeaway for this assignment is to assume that everything is broken and all needs to be tested, even the things you take for granted such as exiting the game or starting the game.