

TESTING

CAMILLA HEIDING

Linnéuniversitetet

2019-03-08

Mail: cd223cd@student.lnu.se

Table of Contents

Objective	4
What to test and how.....	4
Time plan	4
Manual test-cases.....	6
TC1 Start Game.....	6
TC1.1 Start game and initiate new game.....	6
TC1.2 Try invalid input.....	6
TC2 Play single player game	6
TC2.1 Play new game	6
TC2.2 Return to previous game.....	7
TC3 Play multiplayer version	7
TC3.1 Player 1 wins	7
TC3.2 Player 2 wins	8
TC3.3 Change word after first input.....	9
TC4 Play game.....	9
TC4.1 Play game and win	9
TC4.2 Play game and loose	10
TC4.3 Enter same character several times during game	11
TC4.4 Enter invalid input	12
TC4.5 Return to menu	13
Test report for manual testcases	14
Comments.....	14
Automated test-cases.....	15
TC1 Test Game.java.....	15
TC1.1 getWord.....	15
TC1.2 setWord	15
TC1.3 gameSucceded.....	16
TC1.4 underscoresToString.....	17
TC1.5 guessLetter	17
Test report for TC 1.....	19
TC2 Test GuessedLetters.java	19
TC2.1 toString	20
TC2.2 addLetter.....	20
TC2.3 LetterAlreadyGuessed	21
Test report for TC2.....	22
TC3 Test Words.java	24

TC3.1 getRandomWord	24
Test report for TC3	24
TC4 Test MultitplayerGame.java	25
TC4.1 checkWord	25
Test report for TC4.....	26
Automated test report	26
Reflection.....	27
Use Cases from iteration 2	27
UC 1 Start Game	27
UC 2 Play Single Player Game	28
UC 3 Play multiplayer game	28
UC 4 Play game.....	29
UC 5 Quit Game.....	30

Objective

The objective is to test the code that was implemented in iteration two. This testing will only include testing for functionality and not quality.

What to test and how

There will only be dynamic tests since static tests is not a requirement at this stage. The use cases one to four will be tested using manual testing. There should be a test plan for each of these use cases, but these test plans should also go through different paths and not only the main scenario of the use case. This will be done manually since it mainly is input and output on the screen which is hard for a computer to evaluate.

As many methods as possible in the implementation of these use cases will be tested using JUnit. All methods will not be able to be tested using JUnit since many require user-input or only print text in the console without returning anything. The methods which is not tested with JUnit is hopefully covered by the manual testing.

Use case five was excluded due to time constraints. This was the least prioritized use case since you can always turn the game of by simply closing the environment it is run in, this should however be tested when there is time.

Checkboxes, test matrix and comments from testers are included in the end of manual testing section.

Time plan

Deadline: 8/3-2019

Read chapter 8 in *Software Engineering 10th ed. by Ian Sommerville*. This should take about an hour. Lectures on this subject should also be attended. There are five lectures which is estimated 2 hours each which result in 10 hours.

- Write detailed planning for this iteration which should take about 30min.
- Design manual testcases for the use cases which was modeled in iteration two. This should take about 5 hours.
- Create automated unit-tests which also should take about 5 hours.
- Perform the manual testcases that was created and write down any problems. This should take about 30minutes.

- Inspect the code that is tested and look for improvements. This should take an hour.
- Write a test report after performing the tests that was designed in this iteration including any problems that came up. This should take about 30 minutes.

Time log

Task	Estimated time	Actual time	Reflection
Reading literature	1 hour	1h 15min	
Attend lectures	10 hours	10 hours	
Write planning	30 min	20 min	
Write manual test cases	5 hours	6 hours 30 min	Some functions from the original UC was removed. But before this was done some time was consumed trying to solve such problems.
Create Unit Tests	5 hours	5 hours 30 min	The implementation was not well adjusted for testing, so time was consumed figuring out what was testable.
Running manual test cases	30 min	25min	
Code inspection	1 hour	1hour 30 min	Parts of code had to be reconstructed to be testable
Test report	30 min	20 min	

Manual test-cases

TC1 Start Game

Use case: UC1

TC1 should see that the program is able to start and that the correct menu choices are shown. Also, that at least one menu choice works (this will indirectly be tested in further test cases) and that an invalid input result in an error message and that the user is able to enter a new menu choice.

TC1.1 Start game and initiate new game

1. Start the system
2. System should show "1. Play new game", "2. Return to previous game", "3. Play multiplayer version", "4. Terminate program" and "Choose one of the above".
3. Enter "1" and press enter.
4. System should show "Initiating a new game will erase any previous game.", "Are you sure you want to initiate a new game?", "1. Yes", "2. No".

TC1.2 Try invalid input

See that an invalid input result in an error message and that user is able to enter a new menu choice.

Test steps

1. Start the system
2. System should show "1. Play new game", "2. Return to previous game", "3. Play multiplayer version", "4. Terminate program" and "Choose one of the above".
3. Enter "6" and press enter.
4. System should show: "Invalid input, enter a menu choice:"
5. Enter "X" and press enter
6. System should show: "Invalid input, enter a number:"
7. Enter "?" and press enter
8. System should show: "Invalid input, enter a number:"

TC2 Play single player game

TC2 should see that menu choices "1. Play new game" and "2. Return to previous game" works. That the user is asked to confirm initiating a new game before previous game is erased. See that player is able to return to a previous game.

Precondition: Before TC2.2 is performed TC4 should be performed.

TC2.1 Play new game

TC2..1 test that player is able to play a new game but is asked to confirm to erase eventual earlier game.

Test steps

1. Start system and choose menu choice 1.
2. System should show "Initiating a new game will erase any previous game.", "Are you sure you want to initiate a new game?", "1. Yes", "2. No"
3. Enter "1" and press enter.
4. A game of hangman where no letters are guessed, and no parts of the hangman figure should be displayed, perform TC4.
5. After the game is played (either won or lost) the system should return to the main menu.

TC2.2 Return to previous game

TC2.2 test that player is able to return to a previous game.

Precondition: TC2.1 and TC4 must have been tested first.

Test steps

1. Perform step 1-4 in TC2.1 to initiate a new game, perform step 1-4 in TC4.1 to guess a letter and then perform TC4.5 to return to menu.
2. Enter "2" and press enter.
3. System should show "Press 1 to return to the Menu", "Press 2 to terminate program", "T _ _ t", empty lines, " _____ ", "Guessed letters:", "Enter menu choice or guess a letter:"

TC3 Play multiplayer version

Use case: UC3

TC3 should see that player is able to play a multiplayer version and that it works as intended.

TC4 must be tested before the last part of TC3 can be performed.

TC3.1 Player 1 wins

Use case: UC3

Player 2 should fail to guess the correct word and loses the game.

Precondition: TC4 must be tested for the four last steps in this test case to be able to be performed.

Test steps

1. Start system and choose menu choice 3.
2. System shows "Multiplayer version mean that one player enters a word and the other player get to guess the letters of the word. If player 2 manage to

guess the word that means he won, however if he does not player 1 win!",
"Player 1 enter a word:"

3. Enter the word "test".
4. System shows "Player 1 have now entered a word!", "Press 1 to play game with this word", "Press 2 to change the entered word".
5. Enter "1" and press enter.
6. System shows "Player 2 should now guess.", "Enter any character to start game"
7. Enter "x" and press enter.
8. System displays a hangman game (UC 4).
9. Guess the letters "q", "w", "r", "y", "u", "i", "o", "p" and "z" and enter "x" to continue when the game is lost.
10. System shows "Player 1 won the game! Congratulations!", "Enter any character to continue".
11. Enter "x" and press enter.
12. System returns to main menu.

TC3.2 Player 2 wins

Use case: UC3

Player 2 guess all the letters in the word correct at first try and win the game.

Precondition: TC4 must be tested for the four last steps in this test case to be able to be performed.

Test steps

1. Start system and choose menu choice 3.
2. System shows "Multiplayer version mean that one player enters a word and the other player get to guess the letters of the word. If player 2 manage to guess the word that means he won, however if he does not player 1 win!",
"Player 1 enter a word:"
3. Enter the word "test".
4. System shows "Player 1 have now entered a word!", "Press 1 to play game with this word", "Press 2 to change the entered word".
5. Enter "1" and press enter.
6. System shows "Player 2 should now guess.", "Enter any character to start game"
7. Enter "x" and press enter.
8. System displays a hangman game (UC 4).
9. Guess the letters "t", "e" and "s" and enter "x" to continue.
10. System shows "Player 2 won the game! Congratulations!", "Enter any character to continue".

11. Enter "x" and press enter.
12. System returns to main menu.

TC3.3 Change word after first input

Use case: UC3

Player 1 should be able to change the word that was entered before player 2 get to guess.

Test steps

1. Start system and choose menu choice 3.
2. System shows "Multiplayer version mean that one player enters a word and the other player get to guess the letters of the word. If player 2 manage to guess the word that means he won, however if he does not player 1 win!", "Player 1 enter a word:"
3. Enter the word "test".
4. System shows "Player 1 have now entered a word!", "Press 1 to play game with this word", "Press 2 to change the entered word".
5. Enter "2" and press enter.
6. System shows "Player 1 enter a word:".

TC4 Play game

Use case: UC4

TC4 tests the actual game of hangman. That the player is able to guess letters and that the program displays correctly where in the word it is placed or, if not part of the word, adds the letter to guessed letters and draw new part of hangman. In single player game the word is always "Test" during testing.

Precondition: A game must be able to be initiated.

TC4.1 Play game and win

Use case: UC4

Player guess all the letters in the word correct at first try and win the game.

Precondition: A game must be able to be initiated.

Test steps

1. Start system and choose menu choice 1 and confirm creating new game.
2. System shows:
 - "Press 1 to return to the Menu"
 - "Press 2 to terminate program"
 - Four separated underscores (one for each letter in the word),

- Five empty lines where the hangman figure is going to be drawn and a line with underscores representing the ground
 - "Guessed letters: "
 - "Enter a menu choice or guess a letter:"
3. Enter the letter "t" and press enter.
 4. System should show the same as before, but the first underscore should be changed to a "T" and the fourth to a "t".
 5. Enter the letter "S" and press enter.
 6. The third underscore should be changed to a "s"
 7. Enter the letter "e" and press enter.
 8. System shows a stickman raising his arms under a pole and the text "Yay, the man survived! Correct word: Test"
 9. System shows "Enter any character to continue"

TC4.2 Play game and loose

Use case: UC4

Player guess wrong letter 9 times so that the entire hangman is drawn and the game is lost.

Precondition: A game must be able to be initiated.

Test steps

1. Start system and choose menu choice 1 and confirm creating new game.
2. System shows:
 - "Press 1 to return to the Menu"
 - "Press 2 to terminate program"
 - Four separated underscores (one for each letter in the word)
 - Five empty lines where the hangman figure is going to be drawn and a line with underscores representing the ground
 - "Guessed letters: "
 - "Enter a menu choice or guess a letter:"
3. Enter the letter "q" and press enter.
4. System adds "q" after "Guessed letters: " and add the vertical pole in the hangman figure:
5. Enter the letter "w" and press enter.
6. System adds "w" after "Guessed letters: ". and add the horizontal line in the hangman figure.
7. Enter the letter "r" and press enter.
8. System adds "r" after "Guessed letters: " and add the rope in the hangman figure
9. Enter the letter "y" and press enter.

10. System adds "y" after "Guessed letters: " and add the head in the hangman figure.
11. Enter the letter "u" and press enter.
12. System adds "u" after "Guessed letters: " and add the body in the hangman figure.
13. Enter the letter "i" and press enter.
14. System adds "i" after "Guessed letters: " and add the left arm in the hangman figure.
15. Enter the letter "o" and press enter.
16. System adds "o" after "Guessed letters: " and add the right arm in the hangman figure.
17. Enter the letter "p" and press enter.
18. System adds "p" after "Guessed letters: " and add the left leg in the hangman figure.
19. Enter the letter "a" and press enter.
20. System shows the complete hangman figure and the text "Oh no, he died!
Correct word: Test"
21. System shows "Enter any character to continue".

TC4.3 Enter same character several times during game

Use case: UC4

Player guesses the same character more than once, game should only react the first time.

Precondition: A game must be able to be initiated.

Test steps

1. Start system and choose menu choice 1 and confirm creating new game.
2. System shows:
 - "Press 1 to return to the Menu"
 - "Press 2 to terminate program"
 - Four separated underscores (one for each letter in the word)
 - Five empty lines where the hangman figure is going to be drawn and a line with underscores representing the ground
 - "Guessed letters: "
 - "Enter a menu choice or guess a letter:"
3. Enter the letter "q" and press enter.
4. System adds "q" after "Guessed letters: " and add the vertical pole in the hangman figure:
5. Enter the letter "q" and press enter.
6. Nothing should happen.

7. Enter the letter "w" and press enter.
8. System adds "w" after "Guessed letters: ". and add the horizontal line in the hangman figure.
9. Enter the letter "w" and press enter.
10. Nothing should happen.
11. Enter the letter "s" and press enter.
12. The third underscore should be changed to a "s"
13. Enter the letter "s" and press enter.
14. Nothing should happen.
15. Enter "z", "x", "c", "v", "b", "n", "m".
16. The game should be lost.

TC4.4 Enter invalid input

Use case: UC4

Player enters invalid input during game.

Precondition: A game must be able to be initiated.

Test steps

1. Start system and choose menu choice 1 and confirm creating new game.
2. System shows:
 - "Press 1 to return to the Menu"
 - "Press 2 to terminate program"
 - Five empty lines where the hangman figure is going to be drawn and a line with underscores representing the ground
 - Four separated underscores (one for each letter in the word),
 - "Guessed letters: "
 - "Enter a menu choice or guess a letter:"
3. Enter the text "qq" and press enter.
4. System shows "Invalid input, enter a letter or a menu choice:" and waits for new input.
5. Enter the text "3" and press enter.
6. System shows "Invalid input, enter a letter or a menu choice:" and waits for new input.
7. Enter the text "+" and press enter.
8. System shows "Invalid input, enter a letter or a menu choice:" and waits for new input.
9. Press enter, without any input.
10. System shows "Invalid input, enter a letter or a menu choice:" and waits for new input.
11. Enter the letter "q" and press enter.

12. System adds "q" after "Guessed letters: " and add the vertical pole in the hangman figure.

TC4.5 Return to menu

Use case: UC4

Player chooses to return to menu during game.

Precondition: A game must be able to be initiated.

Test steps

1. Start system and choose menu choice 1 and confirm creating new game.
2. System shows:
 - "Press 1 to return to the Menu"
 - "Press 2 to terminate program"
 - Four separated underscores (one for each letter in the word),
 - Five empty lines where the hangman figure is going to be drawn and a line with underscores representing the ground
 - "Guessed letters: "
 - "Enter a menu choice or guess a letter:"
3. Enter the text "1" and press enter.
4. System shows: "Are you sure you want to go back to the menu?", "Your game will be saved until a new game is initiated.", "1. Yes", "2. No"
5. Enter "2" and press enter.
6. System returns to the game.
7. Enter the text "1" and press enter.
8. System shows: "Are you sure you want to go back to the menu?", "Your game will be saved until a new game is initiated.", "1. Yes", "2. No"
9. Enter "1" and press enter.
10. System returns to the main menu (UC1).

Test report for manual testcases

Test	UC1	UC2	UC3	UC4	UC5
TC1.1	1/OK	0	0	0	0
TC1.2	1/OK	0	0	0	0
TC2.1	1/OK	1/OK	0	0	0
TC2.2	1/OK	1/OK	0	1/OK	0
TC3.1	1/OK	0	1/OK	1/OK	0
TC3.2	1/OK	0	1/OK	1/OK	0
TC3.3	1/OK	0	1/OK	0	0
TC4.1	1/OK	1/OK	0	1/OK	0
TC4.2	1/OK	1/OK	0	1(OK	0
TC4.3	1/OK	1/OK	0	1/PROBLEM	0
TC4.4	1/OK	1/OK	0	1/PROBLEM	0
TC4.5	1/OK	1/OK	0	1/OK	0
COVERAGE & SUCCESS	12/OK	7/OK	3/OK	8/OK	0

Comments

In TC4.3 in step 6 and 10 nothing should happen when letter "q" and "w" are entered for the second time according to test description, but the letters are added after "Guessed letters:" again. Part of hangman is not drawn, and the number of guesses left is not affected.

In TC4.3 step 15 the nothing happens when the letter "c" is entered. It is neither added to guessed letters or among the underscores, no part of hangman is drawn either.

In TC4.3 step 16 the game is supposed to be lost but there is still one guess left, probably because of problem whit "c" in previous step.

In TC4.4 step 10 no error message is shown. Pressing enter without any input only skips a row and the system continue to wait for input.

Automated test-cases

TC1 Test Game.java

Each testcase in TC1 test a method from the class Game. Before each sub-testcase in TC1 a new instance of Game is created.

```
private Game sut;  
  
@Before  
public void setUp() {  
    sut = new Game();  
}
```

TC1.1 getWord

During testing the getWord should always return the word "Test".

```
public String getWord() {  
    return theWord;  
}
```

Figure 1 Coding of getWord method

```
@Test  
public void testGetWord() {  
    String expected = "Test";  
  
    String actual = sut.getWord();  
  
    assertEquals(expected, actual);  
}
```

Figure 2 Check that the correct word during testing is returned

TC1.2 setWord

The setWord method should only accept words that consists of letters and dashes, invalid word should cause the method to throw an exception.

```
public void setWord(String word) throws IllegalArgumentException {  
    if(checkWord(word)) {  
        theWord = word;  
        underscores = new String[theWord.length()];  
        for (int i = 0; i < theWord.length(); i++) {  
            if (theWord.charAt(i) == '-') {  
                underscores[i] = "-";  
            } else {  
                underscores[i] = "_";  
            }  
        }  
    } else {  
        throw new IllegalArgumentException();  
    }  
}
```

Figure 3 Coding of setWord

```

@Test
public void testSetWord() {
    String expected = "Expected";
    sut.setWord(expected);

    String actual = sut.getWord();

    assertEquals(expected, actual);
}

```

Figure 4 Test a word that only consists of letters

```

@Test
public void testSetWordWithDash() {
    String expected = "Grand-parent";
    sut.setWord(expected);

    String actual = sut.getWord();

    assertEquals(expected, actual);
}

```

Figure 5 Test a word that consists of letters and a dash

```

@Test(expected = IllegalArgumentException.class)
public void testSetWordInvalidWord() {
    String invalidWord = "*";

    sut.setWord(invalidWord); //this should throw an IllegalArgumentException
}

```

Figure 6 Test a word that consists of an unaccepted sign

TC1.3 gameSucceeded

This method should return true if the player have guessed all the letters in the word.

```

public boolean gameSucceeded() {
    for (int i = 0; i < underscores.length; i++) {
        if (underscores[i] == "_") {
            return false;
        }
    }
    return true;
}

```

Figure 7 Coding of gameSucceeded

```

@Test
public void testGameSucceededFalse() {
    boolean expected = false;
    sut.guessLetter("t");
    sut.guessLetter("e");
    sut.guessLetter("q");

    boolean actual = sut.gameSucceeded();

    assertEquals(expected, actual);
}

```

Figure 8 Test for when the method should return false


```

@Test
public void testGameSucceededTrue() {
    boolean expected = true;
    sut.guessLetter("t");
    sut.guessLetter("e");
    sut.guessLetter("s");

    boolean actual = sut.gameSucceeded();

    assertEquals(expected, actual);
}

```

Figure 9 Test for when the method should return true

TC1.4 underscoresToString

This method should return a string with as many underscores as there are letters in the word. If the word consist of dashes then the underscore should be replaced by a dash in the same position.

```

public String underscoresToString() {
    String underscoresString = "";
    for (int i = 0; i < underscores.length; i++) {
        underscoresString += underscores[i];
    }
    return underscoresString;
}

```

Figure 10 Coding of underscoresToString

```

@Test
public void testUnderscoresToString() {
    String expected = " _ _ _ _";

    String actual = sut.underscoresToString();

    assertEquals(expected, actual);
}

```

Figure 11 Test for a word that only consist of letters

```

@Test
public void testUnderScoresToStringWithDash() {
    sut.setWord("Grand-parent");
    String expected = " _ _ _ _ _ - _ _ _ _ _";

    String actual = sut.underscoresToString();

    assertEquals(expected, actual);
}

```

Figure 12 Test for a word that consist of letters and a dash

TC1.5 guessLetter

This method should replace the underscores when the player guesses the correct letter. If the player guesses the wrong letter methods which is part of other classes should be performed and will be tested elsewhere.

```

public void guessLetter(String input) {
    Boolean correct = false;
    char guess = Character.toLowerCase(input.charAt(0));
    for (int i = 0; i < theWord.length(); i++) {
        if (guess == Character.toLowerCase(theWord.charAt(i))) {
            underscores[i] = " " + theWord.charAt(i);
            correct = true;
        }
    }
    if (!correct) {
        if (guessedLetters.letterAlreadyGuessed(guess) == false) {
            guessedLetters.addLetter(guess);
            stickman.addPart();
            numberOfGuessesLeft--;
        }
    }
}
}

```

Figure 13 Coding for guessLetter

```

@Test
public void testRightGuess() {
    sut.guessLetter("t");
    String expected = " T _ _ t";

    String actual = sut.underscoresToString();

    assertEquals(expected, actual);

    sut.guessLetter("e");
    expected = " T e _ t";

    actual = sut.underscoresToString();

    assertEquals(expected, actual);
}

```

Figure 14 Testing correct guesses

```

@Test
public void testWrongGuess() {
    sut.guessLetter("q");
    sut.guessLetter("w");
    sut.guessLetter("r");
    sut.guessLetter("y");
    sut.guessLetter("u");
    sut.guessLetter("i");
    sut.guessLetter("o");
    sut.guessLetter("p");
    sut.guessLetter("z");
    String expected = " _ _ _ _ ";

    String actual = sut.underscoresToString();

    assertEquals(expected, actual);
}

```

Figure 15 Testing incorrect guesses which should not affect the underscores

Test report for TC 1

Runs:	10/10	Errors:	0	Failures:	0
▼ HangmanGame.TestGame [Runner: JUnit 5] (0,000 s)					
testUnderscoresToString (0,000 s)					
testRightGuess (0,000 s)					
testGameSucceededTrue (0,000 s)					
testSetWordInvalidWord (0,000 s)					
testUnderScoresToStringWithDash (0,000 s)					
testWrongGuess (0,000 s)					
testGetWord (0,000 s)					
testSetWordWithDash (0,000 s)					
testGameSucceededFalse (0,000 s)					
testSetWord (0,000 s)					

Figure 16 All tests passed

Element	Coverage
▼ Hangman	40,3 %
▼ Src	36,5 %
▼ HangmanGame	36,5 %
> JMain.java	0,0 %
> JGame.java	49,8 %
> JMultiplayerGame.java	0,0 %
> JStickman.java	73,6 %
> JGuessedLetters.java	54,0 %
> JWords.java	92,2 %
> Tests	51,5 %

Figure 17 Percentage of code executed during the test

Comments

All tests passed. The main focus was to test Game.java which had 49,8% of its code run during the tests. The code that was not tested contained mostly input from a user or output to the console which is hard to test with JUnit. Hopefully this is covered by the manual testing. Since Game.java depend on other classes a lot of code from other classes was also run.

TC2 Test GuessedLetters.java

Each testcase in TC2 test a method from the class GuessedLetters. Before each sub-testcase in TC2 a new instance of GuessedLetters is created.

```
GussedLetters sut;  
  
@BeforeEach  
public void setup() {  
    sut = new GussedLetters();  
}
```

TC2.1 toString

This method should return a String of all the letters that is currently guessed.

```
public String toString() {  
    String result = "";  
    for (int i = 0; i < guessedLetters.size(); i++) {  
        result += guessedLetters.get(i) + " ";  
    }  
    return result;  
}
```

Figure 18 Coding of toString

```
@Test  
public void testToString() {  
    sut.addLetter('a');  
    String expected = "a ";  
  
    String actual = sut.toString();  
  
    assertEquals(expected, actual);  
  
    sut.addLetter('b');  
    sut.addLetter('x');  
    expected = "a b x ";  
  
    actual = sut.toString();  
  
    assertEquals(expected, actual);  
}
```

Figure 19 Testing that the format of the String is correct

TC2.2 addLetter

This method should add the parameter letter if the letter is not already added.

```
public void addLetter(char letter) {  
    if (letterAlreadyGuessed(letter)==false) {  
        guessedLetters.add(letter);  
    }  
}
```

Figure 20 Coding of addLetter

```

@Test
public void testAddLetter() {
    sut.addLetter('a');
    sut.addLetter('b');
    String expected = "a b ";

    String actual = sut.toString();

    assertEquals(expected, actual);
}

```

Figure 21 Testing that adding two different letters works correctly

```

@Test
public void testAddLetterLetterAlreadyAdded() {
    sut.addLetter('a');
    sut.addLetter('a');
    sut.addLetter('c');
    sut.addLetter('c');
    String expected = "a c ";

    String actual = sut.toString();

    assertEquals(expected, actual);
}

```

Figure 22 Testing that adding the same letter twice should not result in doublets

TC2.3 LetterAlreadyGuessed

This method should determine if the parameter letter is already a guessed letter.

```

public boolean letterAlreadyGuessed(char letter) {
    for (int i = 0; i < guessedLetters.size(); i++) {
        if(letter=='c') {
            return true;
        }
    }
    return false;
}

```

Figure 23 Coding of letterAlreadyGuessed

```

@Test
public void testLetterAlreadyGuessedFalse() {
    boolean expected = false;
    sut.addLetter('s');
    sut.addLetter('q');

    boolean actual1 = sut.letterAlreadyGuessed('t');
    boolean actual2 = sut.letterAlreadyGuessed('a');
    boolean actual3 = sut.letterAlreadyGuessed('r');
    boolean actual4 = sut.letterAlreadyGuessed('x');
    boolean actual5 = sut.letterAlreadyGuessed('o');

    assertEquals(expected, actual1);
    assertEquals(expected, actual2);
    assertEquals(expected, actual3);
    assertEquals(expected, actual4);
    assertEquals(expected, actual5);
}

```

Figure 24 Testing that not yet guessed letters make the method return false

```

80  @Test
81  public void testLetterAlreadyGuessedTrue() {
82      sut.addLetter('a');
83      sut.addLetter('x');
84      sut.addLetter('o');
85      boolean expected = true;
86
87      boolean actual1 = sut.letterAlreadyGuessed('a');
88      boolean actual2 = sut.letterAlreadyGuessed('o');
89      boolean actual3 = sut.letterAlreadyGuessed('x');
90
91      assertEquals(expected, actual1);
92      assertEquals(expected, actual2);
93      assertEquals(expected, actual3);
94  }

```

Figure 25 Testing that already guessed letters make the method return true

Test report for TC2

Runs: 5/5 ❌ Errors: 0 ❌ Failures: 2

TestGuessedLetters [Runner: JUnit 5] (0,000 s)

- testToString() (0,000 s) ✓
- testLetterAlreadyGuessedTrue() (0,000 s) ❌
- testAddLetter() (0,000 s) ✓
- testLetterAlreadyGuessedFalse() (0,000 s) ✓
- testAddLetterLetterAlreadyAdded() (0,000 s) ❌

Figure 26 Two tests failed

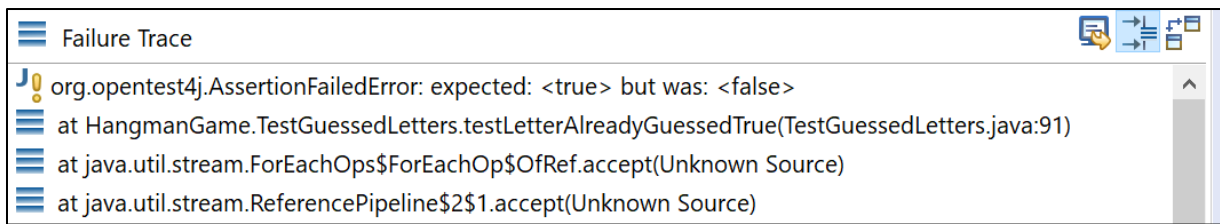


Figure 27 testLetterAlreadyGuessedTrue returned false in line 91 when true was expected

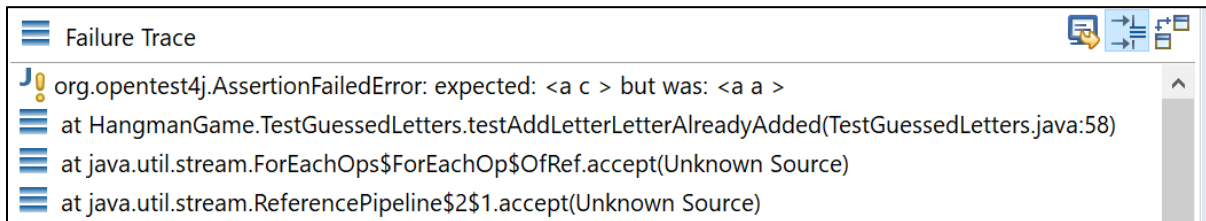


Figure 28 testAddLetterLetterAlreadyAdded present that 'a' was added twice and 'c' was not added at all when each should have been added once

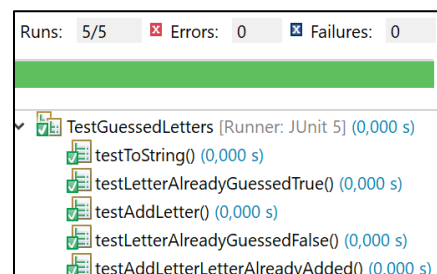
Element	Coverage
▼ Hangman	16,4 %
▼ Src	6,0 %
▼ HangmanGame	6,0 %
> JGame.java	0,0 %
> JMain.java	0,0 %
> JMultiplayerGame.java	0,0 %
> JStickman.java	0,0 %
> JWords.java	0,0 %
> JGuessedLetters.java	100,0 %
> Tests	41,7 %

Figure 29 Percentage of code executed during the test

Comments

Two tests failed, addLetter and letterAlreadyGuessed. Since addLetter depend on letterAlreadyGuessed the problem is likely to be in letterAlreadyGuessed. Looking at the code in this method one sees that the parameter letter is only compared to 'c' and not the actual letters that is already guessed. The correct implementation would be:

```
public boolean letterAlreadyGuessed(char letter) {
    for (int i = 0; i < guessedLetters.size(); i++) {
        if (letter == guessedLetters.get(i)) {
            //if(letter=='c') {
                return true;
            }
        }
    }
    return false;
}
```



The main focus was to test GuessedLetters.java. 100% of the code in GuessedLetters was executed and 0% of other classes, that is because GuessedLetters do not depend on any other classes.

TC3 Test Words.java

Each testcase in TC3 test a method from the class Words.

TC3.1 getRandomWord

This method should return a random word from a text-file. During testing this list only consist of one word which is "Test".

```
public String getRandomWord() {  
    Random rand = new Random();  
    String theWord = words.get(rand.nextInt(words.size()));  
    return theWord;  
}
```

Figure 30 Coding of getRandomWord

```
@Test  
public void testGetRandomWord() {  
    Words sut = new Words();  
    String expected = "Test";  
  
    String actual = sut.getRandomWord();  
  
    assertEquals(expected, actual);  
}
```

Figure 31 Testing that the word returned is "Test"

Test report for TC3

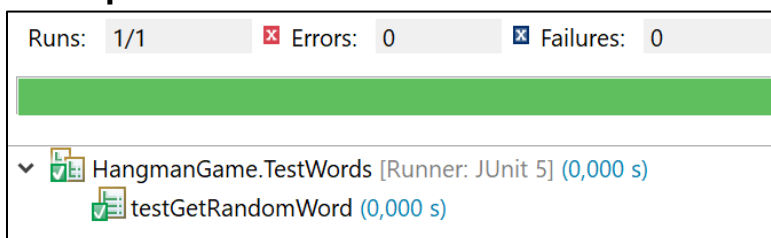


Figure 32 The test succeeded

Element	Coverage
▼ Hangman	4,0 %
▼ src	4,2 %
▼ HangmanGame	4,2 %
> Game.java	0,0 %
> Main.java	0,0 %
> MultiplayerGame.java	0,0 %
> Stickman.java	0,0 %
> GuessedLetters.java	0,0 %
> Words.java	92,2 %
> tests	3,5 %

Figure 33 Percentage of code executed during the test

Comments

List should probably be tested with a list of different words aswell. 92,2% of the class to be tested was executed.

TC4 Test MultitplayerGame.java

Each testcase in TC4 test a method from the class MultiplayerGame.

TC4.1 checkWord

This method should check that the parameter word only contain letters and dashes.

```
public boolean checkWord(String word) {  
    for (int i = 0; i < word.length(); i++) {  
        if (!Character.isLetter(word.charAt(i)) & word.charAt(i) != '-') {  
            return false;  
        }  
    }  
    return true;  
}
```

Figure 34 Coding of checkWord

```
@Test  
public void testCheckWordValidWord() {  
    MultiplayerGame sut = new MultiplayerGame();  
    boolean expected = true;  
  
    boolean actual = sut.checkWord("Example");  
  
    assertEquals(expected, actual);  
}
```

Figure 35 Testing a word which only consist of letters

```
@Test  
public void testCheckWordValidWordWithDash() {  
    MultiplayerGame sut = new MultiplayerGame();  
    boolean expected = true;  
  
    boolean actual = sut.checkWord("Grand-parent");  
  
    assertEquals(expected, actual);  
}
```

Figure 36 Testing a word which consist of letters and a dash

```
@Test  
public void testCheckWordInvalidWord() {  
    MultiplayerGame sut = new MultiplayerGame();  
    boolean expected = false;  
  
    boolean actual = sut.checkWord("*,*");  
  
    assertEquals(expected, actual);  
}
```

Figure 37 Testing an invalid word

Test report for TC4

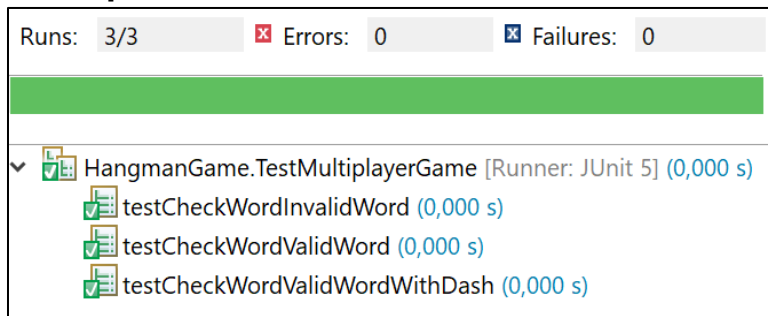


Figure 38 No tests failed

Element	Coverage
▼ Hangman	17,7 %
▼ Src	20,3 %
▼ HangmanGame	20,3 %
> Game.java	21,7 %
> Main.java	0,0 %
> MultiplayerGame.java	27,2 %
> Stickman.java	19,3 %
> GuessedLetters.java	12,9 %
> Words.java	92,2 %
> tests	11,2 %

Figure 39 Percentage of code executed during the test

Comments

A fairly low percentage of MultiplayerGame was executed (27,2%). Most of the code in this class involved input from a player and output to the console which is hard to test using JUnit.

Automated test report

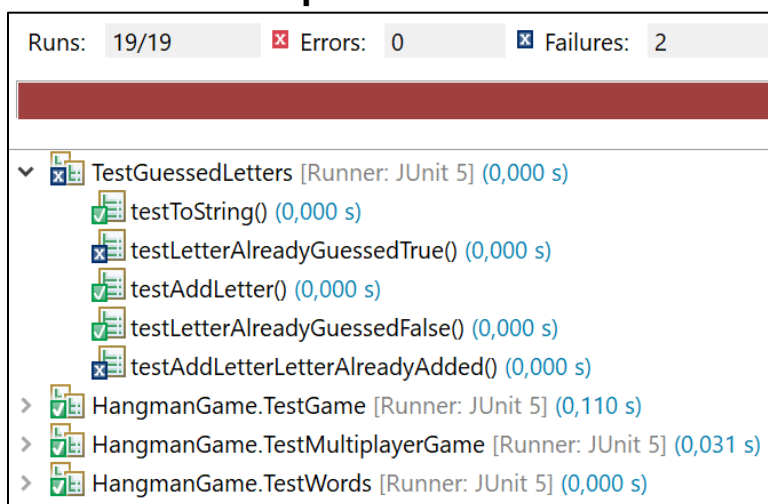


Figure 40 Two tests failed among all testcases











Element	Coverage
▼ Hangman	 58,3 %
▼ Src	 43,3 %
▼ HangmanGame	 43,3 %
> Main.java	 0,0 %
> Game.java	 49,8 %
> MultiplayerGame.java	 27,2 %
> Stickman.java	 73,6 %
> Words.java	 92,2 %
> GuessedLetters.java	 100,0 %
> Tests	 94,5 %

Figure 41 Percentage of code executed during the tests

Comments

The testcases exercise about half of the code in the implementation. Big parts of the game involve interaction between the user and the system which is hard to simulate in automated testing and must be tested using manual testing. Two tests in the test cases failed but the cause was detected and fairly simple to correct.

Reflection

I have been pretty careful testing my program while implementing it in iteration two, therefore there was not many problems detected during this iteration. Perhaps I have missed something which would have been detected if another person was designing the testcases since it often is harder to detect your own faults. I realized that my coding was pretty hard to make automated tests for when there was so much interaction with the player. It became easier to test methods if they was broken up into shorter methods.

Use Cases from iteration 2

UC 1 Start Game

Precondition: none.

Postcondition: the game menu is shown.

Main scenario

1. Starts when the player wants to begin a session of the hangman game.
2. The system presents the main menu.
3. The player makes the choice to play a single player game.
4. The system opens a single player game (see Use Case 2).

Repeat from step 2

Alternative scenarios

3.1 The player makes the choice to play a multiplayer game.

1. The system begin a multiplayer game (see Use Case 3)

3.2 The player makes the choice to quit the game.

1. The system quits the game (see Use Case 5)

3.3 Invalid menu choice

1. The system presents an error message.
2. Go to 2 in main scenario.

UC 2 Play Single Player Game

Precondition: System is running

Postcondition: A single player game has been played

Main scenario

1. Starts when player wants to play single player game.
2. The player chooses to play a new game.
3. System creates new game and play the game (See Use Case 4).
4. Return to menu (See Use Case 5)

Alternative scenarios

2.1 The player chooses to return to previous game.

1. Play previous game (See Use Case 4). If there is no previous game, play a new game.

UC 3 Play multiplayer game

Precondition: The system is running

Postcondition: A multiplayer game has been played.

Main scenario

1. Starts when the player wants to play a multiplayer game.
2. Rules of multiplayer version is shown and player 1 is asked to enter a word.
3. Player enters a word.
4. System set the word and ask for confirmation that the player want to use this word.

5. Player confirms
6. System display that player 2 should now guess the word, player is asked to confirm to continue.
7. Player confirms.
8. The game is played (See use case 4).
9. Player 2 managed to guess the word.
10. Program display that player 2 won the game and is asked to confirm to continue.
11. Player confirms
12. Return to menu (See Use Case 5).

Alternative scenarios

3.1. The player enters invalid word

1. Error message is shown
2. Player is asked to enter a new word.
3. Player enters a new word.

5.1 Player does not confirm

1. Player is asked to enter a new word
2. Player enters a new word.
3. Go to 4 in main scenario.

7.1 Player does not confirm

1. System continues to wait, player must confirm.

9.1 Player 2 did not manage to guess the word

1. System display that player 1 won the game and is asked to confirm to continue.
2. Go to 11 in main scenario.

UC 4 Play game

Precondition: The system is running.

Postcondition: A hangman game has been played.

Main scenario

1. Starts when the player wants to play a game.
2. System show choices and clues and tell player to enter a letter or a choice.
3. Player enters a letter.
4. System tells where in the word that letter is placed.
5. System presents that game is won and ask player for confirmation to continue.

6. Player confirms.
7. Return to previous state.

Alternative scenarios

3.1 The player makes invalid input.

1. Error message is shown.
2. Player is asked to make a new input
3. Player makes new input

3.2 The player makes the choice to quit the game.

1. The system quits the game (see Use Case 6)

3.2 The player makes the choice to return to menu.

1. Player is asked to confirm.
2. Player confirms.
 - 2.1 If player do not confirm: Go to 2 in main scenario.
3. The game returns to the menu. (See UC 1)

4.1 Word does not contain letter.

1. Add part of hangman.
2. Letter is added to guessed letters.
3. Go to 2 in main scenario.

5.1 All the letters in the word is not guessed yet

1. Go to 2 in main scenario.

5.2 Max number of wrong guesses is reached.

1. Game presents game over and ask player to confirm to continue.
2. Player confirms
3. Return to previous state.

UC 5 Quit Game

Precondition: The system is running.

Postcondition: The system is terminated.

Main scenario

1. Starts when the player wants to quit the game.
2. The system prompts for confirmation.
3. The player confirms.
4. The system terminates.

Alternative scenarios

- 3.1. The player does not confirm
 - 1. The system returns to its previous state