Test Plan

**Introduction**

Service request number one proposed game enhancements for version 2.0 of the popular Hangman game distributed by Simple Games Collective.

The following document outlines the test plan to verify that each addition is as error free as possible and works as intended.

**Scope**

The original program was Hangman version 1.0 and had been verified to work error free. The test plan will cover the upgrades 1.1 through to 1.6 before the game is released as version 2.0 complete.

**Overview of testing procedures**

Program testing will consist of the following:

Unit Testing:

As all upgrades will be written in isolation, testing of each function individually to ensure the correct outputs are given will be vital. No syntax errors should occur at any stage. Once each function is working, integration into the main program can occur.

Integration Testing:

As all upgrades are validated for the correct outputs, they can then be integrated into the main program. Here they will be tested together to ensure no incorrect outputs or invalid inputs occur throughout the program.

Functional Testing:

This will test the flow of the program where screens should appear in sequential order, depending on the inputs given.

Volume, Load or Stress Testing is not required for this project and is not included in the Test Plan

**Testing Procedures**

All testing will be conducted in accordance with the Test Case documentation which can be found in appendixes one through seven. Refer to the testing folder for a template to use for each test.

The testing personnel are to ensure that all stated preconditions or assumptions have been met prior to conducting each test. All tests are to be conducted exactly as described in the ‘Test Procedure’ section of the Test Case document.

‘Actual Outcome’ sections should be completed by attaching a screen capture of the observed response.

‘Pass’ tests may be refiled into the testing folder, in test case order.

‘Failed’ tests should include precise descriptions of what was observed that deviated from the expected outcome, together with any relevant error messages or codes. They should then be returned to the development staff for correction and re-testing

Testers are to complete their details at the bottom of the test sheet.

Test cases may be conducted on numerous occasions. When filing these cases ensure they appear in historic sequence, with the most recent on top.

**Test Case Order**

* + 1. Program invocation
    2. Welcome screen with scoreboard
    3. Player name entry and validation
    4. Rules screen
    5. Dummy input
    6. Program invocation
    7. Word list selection screen
    8. Default list selected
    9. Custom word list screen
    10. Custom word list selection
    11. Return feature from custom word list screen to word list screen

1.2.1 Program invocation

1.2.2 Difficulty selection screen

1.2.3 Difficulty selection

1.3.1 Program invocation

1.3.2 Gameplay screen

1.3.3 Invalid entry

1.3.4 Incorrect guess

1.3.5 Correct guess

1.3.6 Win game

1.3.7 Lose game

1.3.8 Quit game

1.4.1 Program invocation

1.4.2 Game won

1.4.3 Game lost

1.4.4 Game quit

1.5.1 Program invocation

1.5.2 Final score screen

1.5.3 Game won

1.5.4 Game lost

1.5.5 New first score

1.5.6 New second score

1.5.7 New third score

1.5.8 No high score achieved

2.0 Full test case

**Recommendations**

It is recommended that this project move to the development stage.

APPENDIX 1 – Test Case – Version 1.1 upgrade.

**Test identifier:** 1.0.1 Program invocation

* + 1. Welcome screen with scoreboard
    2. Player name entry and validation
    3. Rules screen
    4. Dummy input

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “welcome\_screens.py” and at the bottom add the code:

print(welcome(30, 61))

Once testing is complete, remove the added code.

**Test procedure:**

1. In the Windows terminal, navigate to the folder in which the game is located and type “python welcome\_screens.py”. (1.0.1)
2. Observe initial program response. (1.0.2)
3. When prompted to enter a name, test the following inputs: (1.0.3)
   1. longname
   2. h man
   3. stick
4. Observe program response to valid name. (1.0.4).
5. Enter anything to continue (1.0.5)

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.0.1)
2. A game screen should appear with the current scoreboard. (1.0.2)
3. Observe any error messages or response from the program. (1.0.3)
   1. Error, no name entered
   2. Error, name too long
   3. Error, space in name
   4. Valid
4. After entering the valid name, a game screen should appear which uses the name supplied and showing the rules below. (1.0.4)
5. The user should be able to enter any value on this screen, exiting the program. (1.0.5)
6. The program should show the returned value, the name of the user ‘Stick’, before exiting.

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 2 – Test Case – Version 1.2 upgrade.

**Test identifier:** 1.1.1 Program invocation

* + 1. Word list selection screen
    2. Default list selection
    3. Custom list selection screen
    4. Return feature from custom word list screen to word list screen

1.1.6 Custom list selection

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “word\_selection.py” and at the bottom add the code:

print(word\_selection(30, 61, ‘Stick’))

Once testing is complete, remove the added code.

**Test procedure:**

1. In the Windows terminal, navigate to the folder in which the game is located and type “python word\_selection.py”. (1.1.1)
2. Observe initial program response. (1.1.2)
3. When prompted to select, test the following inputs: (1.1.3)
   1. 5
   2. one
   3. 1
   4. 2
   5. 3

(Note: when valid responses are entered the program will exit and values returned. Simply restart the program and return to the same point.)

1. Enter 4 and observe the response. (1.1.4)
2. Enter 0 on the custom list screen and observe response. (1.1.5)
3. Enter 4 to return to the custom list selection screen.
4. When prompted to select, test the following inputs: (1.1.6)
   1. 5
   2. one
   3. 1
   4. 2

(Note: when valid responses are entered the program will exit and values returned. Simply restart the program and return to the same point.)

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.1.1)
2. A game screen should appear with a menu containing three collections of words, and a choice to choose from a custom list. (1.1.2)
3. Observe any error messages or response from the program: (1.1.3)
   1. Error, nothing selected
   2. Error, invalid number
   3. Error, not a number
   4. Valid, returns (‘word’, ‘Beginner’)
   5. Valid, returns (‘word’, ‘Intermediate’)
   6. Valid, returns (‘word’, ‘Expert’)
4. After entering the number 4, a game screen should appear showing all the custom collections stored in the word\_lists folder. (1.1.4)
5. The program should take you back to the first list selection screen. (1.1.5)
6. After entering the number 4, a game screen should appear showing all the custom collections stored in the word\_lists folder. (1.1.4)
7. Observe any error messages or response from the program: (1.1.6)
   1. Error, nothing selected
   2. Error, invalid number
   3. Error, not a number
   4. Error, no words in selection
   5. Valid, returns (‘word’, ‘Animals’)

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 3 – Test Case – Version 1.3 upgrade.

**Test identifier:** 1.2.1 Program invocation

* + 1. Difficulty selection screen
    2. Difficulty selection

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “word\_selection.py” and at the bottom add the code:

print(difficulty\_selection(30, 61, ‘Stick’))

Once testing is complete, remove the added code.

**Test procedure:**

1. In the Windows terminal, navigate to the folder in which the game is located and type “python difficulty\_selection.py”. (1.2.1)
2. Observe initial program response. (1.2.2)
3. When prompted to select, test the following inputs: (1.2.3)
   1. 4
   2. one
   3. 1
   4. 2
   5. 3

(Note: when valid responses are entered the program will exit and values returned. Simply restart the program and return to the same point.)

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.2.1)
2. A game screen should appear with a menu containing three difficulty choices. (1.2.2)
3. Observe any error messages or response from the program: (1.2.3)
   1. Error, nothing selected
   2. Error, invalid number
   3. Error, not a number
   4. Valid, returns (7, ‘Easy’)
   5. Valid, returns (6, ‘Normal’)
   6. Valid, returns (5, ‘Hard’)

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 4 – Test Case – Version 1.4 upgrade.

**Test identifier:** 1.3.1 Program invocation

1.3.2 Gameplay screen

1.3.3 Invalid entry

1.3.4 Incorrect guess

1.3.5 Correct guess

* + 1. Win game

1.3.7 Lose game

1.3.8 Quit game

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “gameplay.py” and at the bottom add the code:

print(gameplay(30, 61, ‘Stick’, ‘TEST’, ‘Hard’, 5))

Once testing is complete, remove the added code.

**Test procedure:**

1. In the Windows terminal, navigate to the folder in which the game is located and type “python gameplay.py”. (1.3.1)
2. Observe initial program response. (1.3.2)
3. When prompted to select, test the following invalid inputs: (1.3.3)
   1. ab
   2. 1
   3. !
4. When prompted to select, test the following incorrect inputs: (1.3.4)
   1. A
   2. m
5. When prompted to select, test the following correct inputs: (1.3.5)
   1. T
   2. e
6. When prompted to select, use enter S to complete the word: (1.3.6)
7. Observe the program response. (1.4.6) (Note: As you just won the game, restart the program.)
8. When prompted to select, use following inputs to lose the game: (1.3.7)
   1. Q
   2. a
   3. z
   4. w
   5. X
9. Observe the program response. (1.4.7) (Note: As you just lost the game, restart the program.)
10. When prompted to select, enter 0 to quit the game: (1.3.8)
11. Observe the program response. (1.3.8)

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.3.1)
2. A game screen should appear with an image of a stage. Below it should be text saying you have five lives. Below in a separate box should be ‘ \_ \_ \_ \_ ‘ to represent the word you are trying to guess. Below in a separate box are the available letters. (1.3.2)
3. Observe the error messages and response from the program: (1.3.3)
   1. Error, nothing selected
   2. Error, more than one character
   3. Error, not a valid character
   4. Error, not a valid character
4. Observe the messages and response from the program: (1.3.4)
   1. Valid, incorrect guess
   2. Valid, incorrect guess

(Note: lives should decrease by one, a body part appears in the image, letter disappears from available letters)

1. Observe the messages and response from the program: (1.3.5)
   1. Valid, correct guess. ‘T \_ \_ T’
   2. Valid, correct guess. ‘T E \_ \_’

(Note: no lives should be lost, image remains the same, letter replaces ‘\_’ and letter disappears from available letters

1. Enter the inputs to complete the word and win the game. Observe the response of the program. It should return (True, the time taken). (1.3.6)
2. Enter the inputs to lose the game. Observe the response of the program. It should return (False, the time taken). (1.3.7)
3. Enter 0 to quit the game. Observe the response of the program. It should return (None, the time taken). (1.3.8)

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 5 – Test Case – Version 1.5 upgrade.

**Test identifier:** 1.4.1 Program invocation

* + 1. Game won
    2. Game lost
    3. Game quit

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “game\_over.py” and at the bottom add the code:

print(game\_over(30, 61, ‘Stick’, ‘TEST’, ‘Hard’, True))

Once testing is complete, remove the added code.

Repeat this for each of the following code:

print(game\_over(30, 61, ‘Stick’, ‘TEST’, ‘Hard’, False))

print(game\_over(30, 61, ‘Stick’, ‘TEST’, ‘Hard’, None))

**Test procedure:**

1. For each test use the Windows terminal to navigate to the folder in which the game is located and type “python game\_over.py”. (1.4.1)
2. Observe initial program response.
   1. First time (1.4.2)
   2. Second time (1.4.3)
   3. Third time (1.4.4).
3. Enter anything to exit the program.

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.4.1)
2. A game screen should appear. It should contain the game over text, an image, the completed word, and conclusion text.
   1. First time through, the image should not have a stick hanging and the text should congratulate the user for winning. (1.4.2)
   2. Second time through, the image should have a stick hanging and the text should say the user lost. (1.4.3)
   3. Third time through, the image should have a stick hanging and the text should say the user quit. (1.4.4)

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 6 – Test Case – Version 1.6 upgrade.

**Test identifier:** 1.5.1 Program invocation

* + 1. Game won
    2. Game lost
    3. New first score
    4. New second score
    5. New third score
    6. No high score achieved

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Open the file “scoring.py” and at the bottom add the code:

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Custom’, ‘Normal’, True, 120))

Once testing is complete, remove the added code.

Repeat this for each of the following code:

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Custom’, ‘Normal’, False, 30))

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Beginner’, ‘Normal’, True, 60))

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Beginner’, ‘Normal’, True, 75))

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Beginner’, ‘Normal’, True, 90))

print(scoring(30, 61, ‘Stick’, ‘TEST’, ‘Beginner’, ‘Normal’, True, 120))

**Test procedure:**

1. For each test use the Windows terminal to navigate to the folder in which the game is located and type “python game\_over.py”. (1.5.1)
2. Observe initial program response.
   1. First time (1.5.2)
   2. Second time (1.5.3)
   3. Third time (1.5.4)
   4. Fourth time (1.5.5)
   5. Fifth time (1.5.6)
   6. Sixth time (1.5.7)
3. Enter anything to exit the program.

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given. (1.5.1)
2. A game screen should appear. It should contain a leader board, the calculated score, and some text about how the user scored.
   1. Score of 4 and placed into scoreboard.
   2. Score of 16 and not placed on scoreboard.
   3. Score of 6 and 1st place on scoreboard.
   4. Score of 4.8 and 2nd place on scoreboard.
   5. Score of 4 and 3rd place on scoreboard.
   6. Score of 3 but not high enough for scoreboard.

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]

APPENDIX 7 – Test Case – Version 2.0 Complete.

**Test identifier:** 2.0 Full test case

**Test components / pre-requisites / assumptions:** This test requires access to a pre-installed Python interpreter on a Windows personal computer, with the game files in a folder that is accessible through the windows terminal navigation system.

Play through four time, once for each word collection Beginner, Intermediate, Expert and Custom. Ensure you choose a different difficulty each time.

**Test procedure:**

1. In the Windows terminal, navigate to the folder in which the game is located and type “python main.py”.
2. Play through the game, noting any errors.
3. Record final scoring screen.

**Expected outcome:**

1. The program should be invoked by the terminal when the input is given.
2. No errors should occur.

**Actual outcome:** [tester records actual observations]

**Test results:** Pass / Fail

**Tester details:**

[Name]

[Contact details]

[time / date]