

# SE 3XA3: Test Plan Snake 2.0

Team 30, VUA30  
Andy Hameed and hameea1  
Usman Irfan and irfanm7  
Vaibhav Chadah and chadhav

October 26, 2018

# Contents

<b>1</b>	<b>General Information</b>	<b>1</b>
1.1	Purpose . . . . .	1
1.2	Scope . . . . .	1
1.3	Acronyms, Abbreviations, and Symbols . . . . .	3
1.4	Overview of Document . . . . .	3
<b>2</b>	<b>Plan</b>	<b>3</b>
2.1	Software Description . . . . .	3
2.2	Test Team . . . . .	3
2.3	Automated Testing Approach . . . . .	3
2.4	Testing Tools . . . . .	4
2.5	Testing Schedule . . . . .	4
<b>3</b>	<b>System Test Description</b>	<b>4</b>
3.1	Tests for Functional Requirements . . . . .	4
3.1.1	Testing Functions & Methods . . . . .	4
3.1.2	Testing of Keyboard/Mouse Interactions . . . . .	6
3.1.3	Testing Game Ending . . . . .	9
3.2	Tests for Nonfunctional Requirements . . . . .	10
3.2.1	Look and Feel . . . . .	10
3.2.2	Usability . . . . .	10
3.2.3	Performance . . . . .	11
3.2.4	Operational and Environmental . . . . .	12
3.2.5	Maintainability and Support Requirements . . . . .	12
3.2.6	Security and Cultural . . . . .	13
<b>4</b>	<b>Tests for Proof of Concept</b>	<b>13</b>
4.1	Snake Dynamics . . . . .	13
4.2	Integration and System Testing . . . . .	15
<b>5</b>	<b>Comparison to Existing Implementation</b>	<b>16</b>
<b>6</b>	<b>Unit Testing Plan</b>	<b>16</b>
6.1	Unit testing of internal functions . . . . .	16
6.2	Unit testing of output files . . . . .	17

<b>7</b>	<b>Appendix</b>	<b>18</b>
7.1	Symbolic Parameters . . . . .	18
7.2	Usability Survey Questions? . . . . .	18

## List of Tables

1	Revision History . . . . .	ii
2	Table of Abbreviations . . . . .	2
3	Table of Definitions . . . . .	2

## List of Figures

Table 1: **Revision History**

Date	Version	Notes
10/25/2018	1.0	Usman added section 3.1, Vaibhav added section 3.2, Andy added section 4.
10/26/2018	1.0	Andy added section 1 and 7, Vaibhav added section 2 and 5, Usman added section 6.
Date 2	1.1	Notes

# 1 General Information

## 1.1 Purpose

Testing will be conducted to ensure that the software meets the requirements set in the original development as well as the new requirements that are set as enhancements to the original game. The generated test cases will also serve as guiding rules for developing code using TDD.

The reinvention of the Snake game as Snake 2.0 will involve new features such as custom speed, high score menu, customizable themes and a multiplayer mode if time permits. These new features, along with the requirements set for the original implementation of the game will be tested to detect any bugs or errors. The use of the Pygame library allows for a GUI that will enable integrated and system testing. Peers will be able to demo the game and catch any errors or bugs by doing so. White box testing will be used as well on the existing code that was used for the POC demonstration. Automated testing will be implemented using the unittest framework built into Python. Aspects that have not yet been implemented or would be hard to detect visually, like for example the speed of the moving snake, will be tested using static analysis along with automated testing.

## 1.2 Scope

Testing will cover all of the behaviours mentioned below. Note that this is a general overview and more details are provided further on in the document.

The expected behaviour is to have a menu that leads to the game screen. Once the game is initiated, the objective is for the snake to eat the food block and continue doing so until the snake dies. Each time the snake eats, the score should increase by one and the length of the snake body should increase by some predetermined number of blocks. The snake dies if it runs into itself by looping around its body or by hitting the edges of the game window. Looking at the Git repository for the original game, there are no test modules that can be seen so the test cases will be based on any test cases generated by the three team members.

Table 2: **Table of Abbreviations**

<b>Abbreviation</b>	<b>Definition</b>
T1D1	Test 1 ID 1
TDD	Test-Driven Development
POC	Proof of Concept
GUI	Graphical User Interface
N/A	Not Applicable

Table 3: **Table of Definitions**

<b>Term</b>	<b>Definition</b>
Pygame	open source Python library used to create game graphics
White Box testing	a method of testing where the code is examined in order to create the test cases. This mostly corresponds to testing functional requirements based on the description of functions and methods in each component module
Black Box testing	a method of testing where the code is not examined in order to create the test cases. This mostly corresponds to non-functional requirements
Test Driven Development	a method of developing software using a set of test cases that are written prior to the code itself. The test cases can identify how functions and methods should behave
PyUnit	Testing framework for python software development

### **1.3 Acronyms, Abbreviations, and Symbols**

### **1.4 Overview of Document**

The document will summarize the test cases that will be conducted on Snake 2.0, a remake of the original snake game using Python and the Pygame library. Several testing techniques are used including automated testing, white box testing, black box testing, manual testing, integration and system testing and static analysis. The document will outline the plan for testing, a description of the test system with non-functional and functional test cases, unit testing and POC testing, and other details pertaining to the testing of Snake 2.0.

## **2 Plan**

### **2.1 Software Description**

The software will act as a medium of entertainment to the users. It is a snake game with added functionality such as different speed and theme options. The implementation of this software is done using Python.

### **2.2 Test Team**

The individuals responsible for testing are Vaibhav Chadha, Usman Irfan and Andy Hameed. Each person will be responsible for testing one's own work. For example, Vaibhav is working on the Graphical User interface of the main screen, hence is responsible for testing it. Usman and Andy will be collaboratively working on the snake game ( which includes recording highest score, current score, snake movement etc.) and will be responsible for testing them likewise.

### **2.3 Automated Testing Approach**

One of the more difficult parts of testing the software will be manual testing. The reason behind this is that a game can be tested better when played as the user can see errors and delays better.

However, automated testing will also be done in order to check certain functionality of the software. For this, PyUnit testing will be used.

## 2.4 Testing Tools

PyUnit testing will be used as a testing tool for this program.

## 2.5 Testing Schedule

See [Gantt Chart](#) for details about the testing schedule

# 3 System Test Description

## 3.1 Tests for Functional Requirements

### 3.1.1 Testing Functions & Methods

#### 1. TID1

Type: Functional, Dynamic, manual

Initial State: The desktop application starts waiting for the user to enter a command to begin.

Input: The user presses any button key.

Output: The desktop application begins moving the snake towards the Right.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press any key to see if it would run the game, making the snake move.

#### 2. TID2

Type: Functional, Dynamic, Manual

Initial State: The desktop application executes and displays a screen with a headline **High Score:** in the top

Input: NULL

Output: The game would display the highest score of the user from the day they started to play till the present date.

How test will be performed: When the game is played for the first time its Highest Score should be 0, the developer plays for a while and tests the score they made by playing should be the highest score when playing the second time.

When the game is restarted or turned off the game still holds the highest score.

Two more scenarios are present to test the highest score requirement. Take for example a current high score of 85:

1. The user beats the highest score, and the highest score is updated to the user's score when playing the next time.
2. The user is not able to defeat the highest score and the highest score will still be 85 when they play the game next time.

### 3. **TID3**

Type: Functional, Dynamic, Manual

Initial State: The desktop application executes and displays the snake at a random location.

Input: NULL

Output: The snake displays the snake at random location when played the next time.

How test will be performed: The user can track the location of the snake the first time the game is played. The game should be restarted to ensure that the snake's position changes every time the game starts.

### 4. **TID4**

Type: Functional, Dynamic, Manual

Initial State: The snake's food is at a random location.

Input: NULL.

Output: The food reappears on the screen at a random location when the snakes eat the previous one.

How test will be performed: The developer will test this requirement by moving the snake's head location equal to the food's location. When the snake eats the food, instantly another food should display on the screen at a random location.



### 3.1.2 Testing of Keyboard/Mouse Interactions

#### 5. TID5

Type: Functional, Dynamic, Manual

Initial State: The desktop application starts waiting for the user to enter a command to begin.

Input: The user presses F-11 key.

Output: The desktop application screen is changed to full-screen mode.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press the F-11 key to test if the size of the screen changes to full-screen mode.

#### 6. TID6

Type: Functional, Dynamic, Manual

Initial State: The game waits for the user to press a direction key to move the snake.

Input: The user presses UP key.

Output: The snake in the game would move up by one-unit length.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press the UP key to test if the snake moves in the upward direction.

#### 7. TID7

Type: Functional, Dynamic, Manual

Initial State: The game waits for the user to press a direction key to move the snake.

Input: The user presses DOWN key.

Output: The snake in the game would move down by one-unit length.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press the DOWN key to test if the snake moves in the downward direction.

## 8. **TID8**

Type: Functional, Dynamic, Manual

Initial State: The game waits for the user to press a direction key to move the snake.

Input: The user presses LEFT key.

Output: The snake in the game would moves left by one-unit length.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press the LEFT key to test if the snake moves in the left direction.

## 9. **TID9**

Type: Functional, Dynamic, Manual

Initial State: The game waits for the user to press a direction key to move the snake.

Input: The user presses RIGHT key.

Output: The snake in the game would moves right by one-unit length.

How test will be performed: The test will be done dynamically, that means once the program will be executed the developer will press the RIGHT key to test if the snake moves in the right direction.

## 10. **TID10**

Type: Functional, Dynamic, Manual

Initial State: The desktop application executes and displays three modes to be played.

Input: Mouse Cursor

Output: The application should open the specific mode the user has requested to play.

How test will be performed: Different modes in the game will be opened using the mouse cursor, their display or speed should be different from other modes. Easy having the slowest speed and allowing the snake to

exit from the one-direction boundary and enter from the other direction of the boundary (e.g. leaving from right side boundary and entering from the left side boundary). While playing the hard mode, the speed should be much faster than the Easy mode, and would not allow the snake to cross the boundary. If the snake touches the boundary the snake should die and terminating the game.

#### 11. **TID11**

Type: Functional, Dynamic, Manual

Initial State: The desktop application executes and displays three modes to be played.

Input: Mouse Cursor

Output: Changes the theme of the game application.

How test will be performed: If the game is initially set to Light mode. On clicking the Theme button the game changes the theme from Light to Dark theme.

#### 12. **TID12**

Type: Functional, Dynamic, Manual

Initial State: The initial length of the snake would be one-unit length.

Input: The user presses the Direction keys to control the snake

Output: The length of the snake should not equal to one-unit length when it dies (Hard mode would be an exception).

How test will be performed: The developer moves the snake by pressing the direction keys. When the snake's head location equals the food location, its length should be increased by five unit-length. When the snake dies its increase in length should be divisible by 5.

#### 13. **TID13**

Type: Functional, Dynamic, Manual

Initial State: The game is already executed, and the user is playing the game.

Input: The user presses the Space key to pause the snake.

Output: The snake's movement has been stopped.

How test will be performed: The developer will test this requirement by pressing the Spacebar key in between the game, will track down's snake location and see if the snake stops on the screen of the Spacebar key is pressed for the first time. To test the pause movement, we can think of pausing and resuming of the game as two states. If the Spacebar key is pressed odd time it will be in the pause state else, it will be in the resume state.

#### 14. **TID14**

Type: Functional, Dynamic, Manual

Initial State: The snake's movement has been stopped.

Input: The user presses the Space key to resume the snake's movement.

Output: The snake's movement has been resumed.

How test will be performed: The developer will test this requirement by pressing the spacebar key in between the game, will track down's snake location and see if the snake moves on the screen if the spacebar key is pressed for the second time. To test the pause movement, we can think of pausing and resuming of the game as two states. If the spacebar key is pressed odd time it will be in the pause state else, it will be in the resume state.

### 3.1.3 Testing Game Ending

#### 15. **TID15**

Type: Functional, Dynamic, Manual

Initial State: The snake is not one-unit length..

Input: NULL.

Output: The screen displays a screen biting itself, and a message prompts on the screen display "GAME OVER!".

How test will be performed: The developer will test this requirement by moving the snake's head location equal to the snake's body location. When the snake eats its body the snake's movement should stop and will be able to see the error message.

#### 16. **TID16**

Type: Functional, Dynamic, Manual

Initial State: The snake is red colour

Input: NULL.

Output: The screen displays a screen biting itself in red colour .

How test will be performed: The developer will test this requirement by intentionally killing the snake. The function should change the snake's colour from green to red. If the snake's colour is red before replaying the game, the function passes its test (exception: the user presses the restart button and doesn't want to play the game till the end).

### **3.2 Tests for Nonfunctional Requirements**

#### **3.2.1 Look and Feel**

##### **1. TID17**

Type: Structural, Dynamic, Manual

Initial State: The game should be installed on the device.

Input/Condition: The game is opened and ran on the device.

Output/Result: The User Interface should open with different buttons, alongside with a playground with a snake in it.

How the test will be performed: The program will manually run on the device and checked by the human eye to see if it meets the criteria.

#### **3.2.2 Usability**

##### **1. TID18**

Type: Structural, Dynamic, Manual

Initial State: The program will be running for a human nearing 10 years of age or above

Input/Condition: The program will be set on its default settings

Output/Result: The person testing should be able to understand the game and play it. He/she should be able to customize themes and speed of the game.

How the test will be performed: A younger human of nearing age 10 will be asked to operate this game and recorded if he/she is able to operate it successfully or not.

### **3.2.3 Performance**

#### **1. TID19**

Type: Functional, Dynamic, Manual

Initial State: The program will be running with the main user interface open.

Input/Condition: The button is pressed.

Output/Result: The response time for button should be less than half a second.

How the test will be performed: It will be performed using human actions. The response would be times to be as precise as possible. Also, it will be taken into consideration that the user doesn't have to wait for a long observable time.

#### **2. TID20**

Type: Structural, Dynamic, Manual

Initial State: The snake Game will be running on the device.

Input/Condition: Various speed inputs for the snake.

Output/Result: Different snake speeds according to what the user has decided

How the test will be performed: The game will be played with inputting different speeds. Then, the speed difference will be observed as the game progressed through and taken care that the game goes at the constant speed at each level.

### 3. **TID21**

Type: Structural, Dynamic, Manual

Initial State: The snake Game will be running on the device.

Input/Condition: The snake will be moving around and keys will be pressed to change directions.

Output/Result: Snake should change directions promptly.

How the test will be performed: While the game is going on, the buttons will be pressed to change the direction of the snake.

#### 3.2.4 **Operational and Environmental**

##### 1. **TID22**

Type: Structural, Dynamic, Manual

Initial State: The program will be moved on a USB.

Input/Condition: The USB will be inserted into any other working computer/ Desktop.

Output/Result: The game should be able to run on it as long as the device is powered and in working state.

How test will be performed: Many different laptops, alongside with desktops, will be used to test. The game will be played on different devices with different specifications to make sure that the game is playable regardless of the specs of the device.

#### 3.2.5 **Maintainability and Support Requirements**

##### 1. **TID23**

Type: Functiona Dynamic, Manual

Initial State: The program will be moved to a Windows, Mac OS and Linux operating devices.

Input/Condition: The program will be executed.

Output/Result: The game should run.

How test will be performed: The game will be taken and transferred to the systems operating on different OS's. For this, the target is Windows device, Mac OS device and a Linux Device.

### **3.2.6 Security and Cultural**

#### **1. TID24**

Type: Structural, Dynamic, Manual

Initial State: The program will be running.

Input/Condition: All the interfaces running.

Output/Result: No offensive or illegal content on the entire application.

How test will be performed: The application will be executed and each page and option will be approached to make sure there is no offensive or illegal content. Also, there is a Static module to this requirement where all the files (including code) will be looked to make sure about no offensive or illegal content.

## **4 Tests for Proof of Concept**

The POC consists of a simple demonstration of the moving snake in the game window along with a start menu. The food item will not be created in the demo, instead, the testing will only involve the movement of the snake and the main menu that has been created at the start of the game.

### **4.1 Snake Dynamics**

#### **Snake Movement and Speed**

##### **1. TID25**



Type: Dynamic

Initial State: The snake body - graphically represented by a red square - is initially motionless. It exists somewhere within the frame of the window.

Input: Keyboard Event - user clicks on one of the directions on the keyboard arrow pad.

Output: Snake moves according to the direction chosen. This can logically be represented by the expression `keyboardEvent.direction == snakeMovementDirection`. Note that the variables used are arbitrary and are dependant on Python syntax.

How test will be performed:

- A method will be created under the POC test class where the keyboard event is manually set to the code representing each of the directions on the arrow keypad - up, down, left and right. The direction inserted will be asserted equal to the direction of the moving snake, set by some variable.
- After starting the game, the user will click on each one of the four directions and verify whether or not the snake is moving in the corresponding direction, using the graphical interface created with Pygame.

## 2. TID26

Type: Dynamic, Functional testing

Initial State: The snake body - graphically represented by a red square - is initially motionless. It exists somewhere within the frame of the window.

Input: Keyboard Event - user clicks on one of the directions on the keyboard arrow pad.

Output: Snake moves accurately according to the speed set in the snake module. Statically, this can be represented for the vertical movement of the snake by this expression:

$$(snakeFinalPosition - snakeInitPosition) == (speed \times timeElapsed) * vel$$

where vel is the distance defined for 1 single step and speed is the delay between each step in milliseconds

How test will be performed: A method will be created under the POC test class where the keyboard event is manually set to the code representing each of the directions on the arrow keypad - up, down, left and right. The logical expression above is implemented into an assert statement verifying that the distance moved corresponds to the speed and velocity that were used as well as the time that has elapsed - this can be obtained from the time object in Pygame.

## 4.2 Integration and System Testing

### 1. TID27

Type: Integration, System testing

Initial State: The game menu is loaded onto the window with options for starting the game and quitting the game.

Input: The following sequence of inputs

- (a) User clicks start game
- (b) Keyboard Event - user clicks on one of the directions on the keyboard arrow pad.
- (c) user exits the window by clicking the exit tab on the top right corner of the screen

Output: Snake game runs as intended. The "start game" option leads the user to the game screen where the snake body sits motionless. The user moves the snake body using the keyboard arrow pad, moving in directions that correspond appropriately to the arrows clicked and in the correct sequence. The game window is closed once the user clicks the exit button on the top right corner.

How test will be performed: Several peers will be asked to test the game from start to finish for this integration and system test.

## 5 Comparison to Existing Implementation

Currently, we have the following tests that compare to the existing comparison:

1. TID3 : In the Proof of Concept, it has already been tested that the snake appears at a random position everytime a new game is played.
2. TID6, TID7, TID8, TID9 : In the code, its already tested that the snake moves in the direction of the button pressed as soon as it is pressed.
3. TID19 : The current code for the game meets the requirement as it launched the operation of a button as soon as it is pressed. For this, the "Play Game" button and "Quit" button has been tested.
4. TID23 : The present code was transferred on windows and Mac OS devices and was working completely fine.
5. TID25 : All the mentioned testing has been done for the POC.

## 6 Unit Testing Plan

The PyUnit testing framework would be used to test our desktop application.

### 6.1 Unit testing of internal functions

The PyUnit testing framework will be used to test our source code's functions, this is an automated testing unit, and it provides classes which can ease different testing functions. By using PyUnit we can check the robustness of our program, if wrong inputs are given will the program be able to handle such cases without crashing. Besides, the requirement of the program can be tested to see if our program matches with the functional and non-functional requirements of the program. For example, the function that moves the snake in X-axis and Y-axis can be tested by entering the snake's X, Y location, the axis it wants to go and its direction (up or down for y-axis, and left or right for x-axis). The goal is to test as many functions examining all possible cases which can make our application run smoothly without crashing.

## **6.2 Unit testing of output files**

The testing of the output files through unit testing will tell the developers if all the test cases designed by them run efficiently. The snake's movement would be compared to the actual output if the user is pressing the UP key and the snake is moving in the respective direction it would pass the unit testing. Testing the output files can also help us to find that if different modes of the game are selected then different rules of the games should be followed. The game being played in the Hard mode could be tested that the snake is not allowed to cross boundaries and this could be compared with automated testing allowing us to know if our output files have passed their unit test.

## 7 Appendix

### 7.1 Symbolic Parameters

N/A

### 7.2 Usability Survey Questions?

The following questions will be asked to peers when conducting integrated and system testing:

- Does the game lag at any point?
- Does the game maintain consistent speed performance as you advance through the game?
- Is the main menu clear and understandable?
- Is the game exciting to play?
- Is the game visually appealing? Does it look visually complete?

Other questions will be asked to validate the game, mostly focusing on non-functional requirements whose completeness is subjective to the user.