Software Engineering:

Test Plans & Test Cases in Tic Tac Toe

Samantha M. Hipple

School of Technology and Computing, City University of Seattle

CS 504: Software Engineering

Kendra Schraml

May 30, 2022

Software Engineering:

Test Plans & Test Cases in Tic Tac Toe

Within the field of software engineering, testing plays an extremely vital role in ensuring that a software product does what it is supposed to do. Computer scientist, Tom Kilburn, is credited with writing the first piece of computer software back in 1948. Kilburn's program was designed to perform mathematical calculations using machine code instructions. Alongside this initial software product, came the first introduction to software testing in the form of debugging. Debugging is the process of isolating and fixing any faults or errors found within a software system and was the main method of testing during the industry's first few decades. In modern day practice, debugging is not even considered a part of software testing, but instead is applied after testing has discovered any errors or bugs in the system (IBM, *What is Software Testing?*).

Throughout this paper, we will be reviewing how software testing has evolved over the last near-century, why testing is important throughout the entire software development life cycle (SDLC), and what types of testing are commonly involved in the production of a software system. Afterwards, we will examine the software testing techniques used during the development of a simple, command-line, Tic Tac Toe game, providing a practical demonstration of the topic at hand.

Traditional vs Continuous Testing

Traditionally, the software testing process was separate from the rest of development and performed near the end of the SDLC, after the product has been built and executed. Oftentimes, software testers were only given a short window of time in which they could assess the final product before the expected release date.

Any defects found would then cause either a need to postpone the release date to fix the errors or a need to push the release of a defected product. Moving the testing activities to earlier phases of the SDLC has helped keep testing efforts a priority throughout development, instead of as an afterthought to development (IBM, *What is Software Testing?*).

By the 1980s, software development teams began moving away from only focusing on fixing software bugs, to testing their applications in real-world settings. By the 90s, software testing had been integrated into a larger, overall process called quality assurance (IBM, *What is Software Testing?*). Software quality assurance (SQA) is a focus throughout the entire software development process that includes: (1) a quality management approach that matches planned engineering activities with SQA team members of equal or greater skill-level, (2) pre-determined check-points to evaluate project performance data, (3) incorporation of a multi-testing strategy, (4) measuring change impacts to ensure that any fixes remain compatible with the whole project, and (5) managing good relations within the working environment to keep communication lines open and honest between teams (MKS075, 2021).

As Agile and DevOps development strategies were embraced by enterprises worldwide for their ability to enhance competitive advantages through improved delivery speeds and product quality, there was an added focus on the significant, positive impacts of continuous integration (CI), continuous delivery (CD) and continuous testing (CT) throughout the SDLC. Although the importance of CI and CD tools and implementations by the development team should not be understated, without CT ensuring that the integrated changes are compatible with the overall

system, CI and CD would be meaningless. Continuous testing is comprehensive in its scope - including teams, tools, testers, and services - and puts in place processes, systems, and automation that enables an accelerated time to market; a constant feedback loop from within and outside (e.g., end-users) the development teams; and desirable business outcomes such as the development of high-quality products and services, operational efficiency, responsiveness, competitive differentiation and enhanced customer service (Choudhary, 2020).

Why is Software Testing Important?

It is hard to argue against the need for quality control measures when building software products. Mistakes as simple as a typo, a misplaced indent or curly bracket, unexpected user inputs, etc. can easily break an entire program if they are not corrected prior to integration. Modern day customers expect software that is able to operate across multiple platforms, devices, browsers, and networks; applications that are high-performing, navigable, scalable, secure, user-friendly, and fast-loading. Late deliveries and software defects can have a significantly negative effect on customer confidence in a software company (IBM, *What is Software Testing?*).

Although software testing does create an additional startup cost for the development company, the money saved by identifying and fixing defects before a product is deployed can be exponential. The earlier a system fault is discovered in the SDLC, the cheaper is it will be to fix. "IBM estimates that if a bug costs $100 to fix in Gathering Requirements phase, it would be $1,500 in QA testing phase, and $10,000 once in Production" (McPeak, 2017). When the development process

leaves ample room aside for testing, high-quality, reliable software applications can be delivered with few errors (IBM, *What is Software Testing?*).

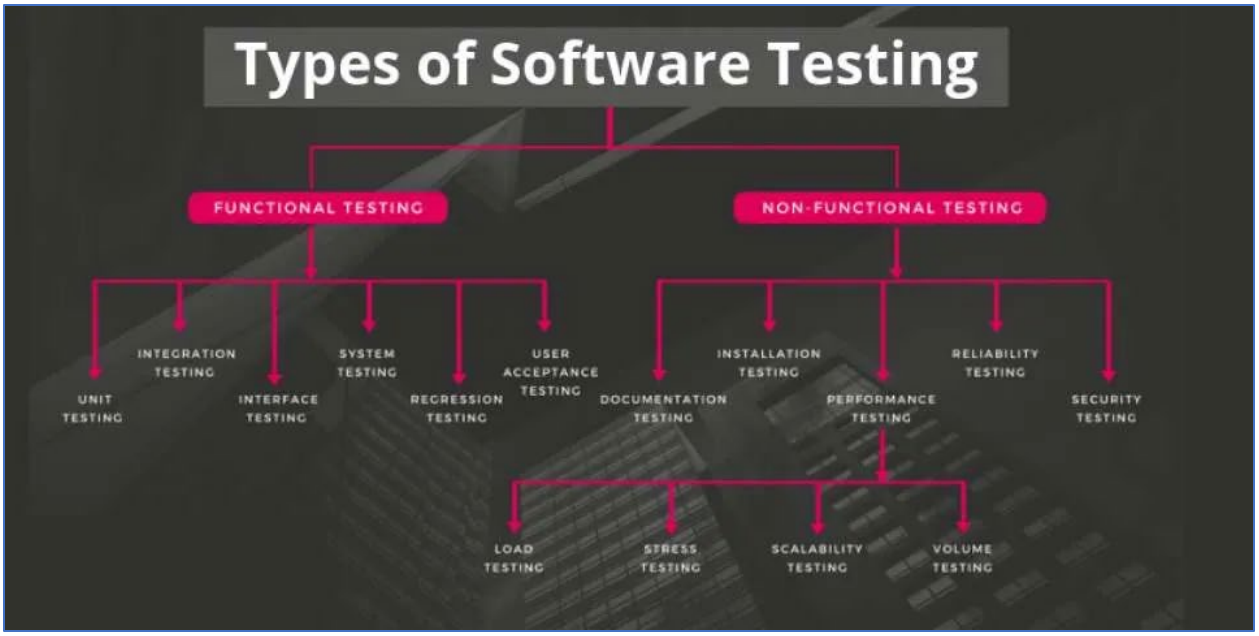
Types of Software Testing

The two main categories in software testing are (1) functional testing and (2) nonfunctional testing. Functional testing inspects each and every functional component of a software system to ensure that the product does what it is designed to do. Nonfunctional tests are performed after functional testing and focus on polishing the end product. Figure 1 below provides a visualization of what types of tests are commonly involved in these two categories (Medewar, 2022).

Following the diagram from left to right under the functional testing branch, we can see the scope of each subsequent strategy incorporates more and more of the overall system. Unit testing focuses on validating the smallest, testable

Figure 1

Types of software testing diagram (Medewar, 2022)



components of the overall system (e.g., class methods within a module). Integration testing is meant to ensure that the software's units function properly when working together. Regression testing ensures that newly added features do not break or degrade the system's overall functionality. Lastly, acceptance testing is done to ensure that the entire system works as intended. Functional tests can be performed either manually or with the use of automated tools (Medewar, 2022).

In review of the nonfunctional testing branch, we see that performance testing contains a sub-branch that includes load testing, stress testing, scalability testing, and volume testing. This is because performance testing is meant to verify how the system performs under different workloads. Load testing is used to determine how much load a system can manage before performance begins to degrade. Stress testing is used to determine how much strain the system can handle before it fails. Scalability testing checks how the system manages increases in the number of users, data, and traffic (Medewar, 2022).

Tic Tac Toe - A Practical Application

In order to demonstrate a practical understanding of testing in software engineering, the rest of this paper will focus on the testing used during the development of a command-line Tic Tac Toe game written in Python. At this point in time, development is working towards the program's second release. As features are added or modified, both unit test scripts have been developed to ensure that any changes maintain compatibility with the rest of the program. Additionally, as game loops are created and modified, as inputs are requested, as exceptions are caught, etc., the game is constantly being ran to test for proper end-user functionalities.

The testing of our Tic Tac Toe program will only involve the game's functional components. While nonfunctional testing has grown in its importance over the years to ensure proper performance and security for end users, our Tic Tac Toe game is simply not that serious of a program – it is not even being released as an executable to the public. Instead, the game is simply an exercise in writing good code with an object-oriented focus, making it perfect for learning to write test scripts for unit tests!

Testing in Python

To begin our formal Tic Tac Toe testing, we first had to choose a Python test runner and library. The three most popular test runner modules for Python are: **unittest**, **nose** or **nose2**, and **pytest**. The **unittest** module is built into Python's standard library. This module requires the test scripts to be written as class methods and provides specific assertation methods from the **unittest.TestCase** class to create the test comparisons in a way that the rest runner will understand. Both **nose**(**2**) and **pytest** are external testing libraries designed to support the execution of **unittest** test cases (Shaw, 2018).

Our example testing focuses mainly on the use of Python's built-in **unittest** module for both writing and running unit tests, with occasional use of test methods from an external Python library called NumPy. The following sections of this report will review ten separate test scripts that were written to assess the base methods within our game's **TicTacToe** class. These handful of test scripts are in no way considered a comprehensive testing of our game, but instead are a simple introduction to how unit test cases can be written and evaluated through the use of a test runner module.

Creating and Running unittest Methods

All of our Tic Tac Toe test cases are considered unit tests. That is, they evaluate the actual output of our most basic **TicTacToe** class methods against an expected output using assertation statements. The ten base methods we test include activities such as creating, displaying, and resetting our gameboard; the player choosing their game marker (X or O); properly assigning game markers to players; randomly choosing first player; generating random moves for the AI; obtaining proper gameboard coordinates for a specified move; placing player markers on the gameboard; and completing a human player move from input to marker placement.

Figure 2 below displays the libraries that we imported into our **test** module in order to create our unit tests, isolate functional behaviors, ascertain expected outputs, and engage our test runner. Additionally, prior to creating our test class, we created a variable named **Tic** that is initialized as a **TicTacToe()** class object. This way we can more readily call any properties and methods necessary from our **tictactoe** module as we progress.

Figure 2

*Importing the required libraries for our **TicTacToeTest(unittest.TestCase)** class*

```
1   import unittest
2   import numpy as np
3   from tic_tac_toe import TicTacToe
4   from io import StringIO
5   from unittest.mock import patch
6
7   # create a public TicTacToe class object to use in the method unit tests
8   Tic = TicTacToe()
9
10  # create a class for our tic_tac_toe module unit test cases
11  class TicTacToeTest(unittest.TestCase):
12      """Unit tests for our TicTacToe class."""
```

Figure 3

Evaluating the generation of our base gameboard

```
14      # 01. test create_board()
15      def test_create_board(self):
16          """Test that the gameboard is properly created."""
17          Tic.create_board() # call method we are testing
18          values = ['1', '2', '3', '4', '5', '6', '7', '8', '9']
19          test_board = np.reshape(values, (3, 3)) # construct our expected result
20          # compare using np.testing due to type of ndarray
21          np.testing.assert_array_equal(Tic.board, test_board)
```

1. Create, Display, and Reset the Gameboard

The first thing we did when creating our first unit test, is call the **TicTacToe** method that we wish to test, **Tic.create_board()**. We then created a variable that holds our expected output, **test_board**, and initialized it to hold values 1-9, as strings, shaped into a three-by-three grid.

The method **np.testing.assert_array_equal()** was required to run our assertation statement instead of one of the **unittest** methods due to our gameboard's data type. The **unittest** method that compares two lists recognized that the data type is actually a **numpy** ndarray and produced a traceback report that suggested the use of **numpy**'s built-in array testing methods instead. Figure 3 above displays the code used to create our first test script.

As demonstrated in Figure 4, our second unit test focuses on displaying our gameboard properly. In theory, this test could be done by simply running the code instead as the output of this method prints to screen. However, in the spirit of making our tests viable for potential future automation, we rigged up a test that captures the printed output and transforms it into a new **StringIO** object from the **io** module using the **patch** method within the **unittest** module. We then compared the captured output's value to a newly created, long-hand version, of our

Figure 4

Assessing the gameboard display

```
23      # 02. test display_board()
24      def test_display_board(self):
25          """Test that the gameboard is properly displayed."""
26          with patch('sys.stdout', new=StringIO()) as output:
27              Tic.display_board() # capture output of the method we are testing
28          # construct expected output
29          values = ['1', '2', '3', '4', '5', '6', '7', '8', '9']
30          test_board = np.reshape(values, (3, 3))
31          test_display =   ('\t-------------------------------\n')
32          test_display +=  ('\t|         |         |         |\n')
33          test_display += (f'\t|    {test_board[0, 0]}    |    {test_board[0, 1]}    |    {test_board[0, 2]}    |
            \n')
34          test_display +=  ('\t|         |         |         |\n')
35          test_display +=  ('\t-------------------------------\n')
36          test_display +=  ('\t|         |         |         |\n')
37          test_display += (f'\t|    {test_board[1, 0]}    |    {test_board[1, 1]}    |    {test_board[1, 2]}    |
            \n')
38          test_display +=  ('\t|         |         |         |\n')
39          test_display +=  ('\t-------------------------------\n')
40          test_display +=  ('\t|         |         |         |\n')
41          test_display += (f'\t|    {test_board[2, 0]}    |    {test_board[2, 1]}    |    {test_board[2, 2]}    |
            \n')
42          test_display +=  ('\t|         |         |         |\n')
43          test_display +=  ('\t-------------------------------\n')
44          # compare the value of our method's output and our expected output
45          self.assertEqual(output.getvalue(), test_display)
```

gameboard display method (i.e., attempted to essentially draw the expected display line by line).

The third unit test in this section is the simplest we have implemented thus far. When initializing our **tic_tac_toe** module, we create a property called **self.board** and initialize it as an empty list. So, we compare the effect of **Tic.reset_board()** on the **Tic.board** property against our expected output variable, **test_board**, which has been redefined as an empty list as well. Additionally, we do not have to use **numpy**'s assertation statement this time because both boards should end up equaling an empty list, which is a datatype that the **unittest** module is prepared to handle. Figure 5 below display the code for the final unit test in this section.

Figure 5

Testing the resetting of our gameboard to an empty list

```
47      # 03. test reset_board()
48      def test_reset_board(self):
49          """Test that the gameboard is reset to an empty list"""
50          Tic.reset_board() # call method we are testing
51          test_board = [] # construct expected result
52          # compare by calling in the class variable that was reset using Tic
53          self.assertEqual(Tic.board, test_board)
```

2. Choosing, Assigning, and Placing Game Markers (X or O)

In order to isolate the behaviors of our fourth **TicTacToe** method, **choose_marker()**, we first created a mocked input to pass as a test method parameter in order to remove the side-effect of requesting user input via the command line. In our test, we pass both **'x'** and **'o'**, call the method twice, and assert what the results should be for each mocked input. Despite running two assertation statements, the test runner recognizes the code displayed in Figure 6 below, as a single unit test.

We used the same mockup input for our fifth test case as well. The method **assign_marker()** calls our previous method **choose_marker()** as part of its

Figure 6

Testing the output of proper user inputs when choosing a player marker (X or O)

```
55      @unittest.mock.patch('tic_tac_toe.input', create = True)
56      # 04. test choose_marker()
57      def test_choose_marker(self, mocked_input):
58          """Test that user input is properly reflected in the list order of the returned characters."""
59          mocked_input.side_effect = ['x'] # create a mock input for 'x'
60          result = Tic.choose_marker() # capture results of the method we are testing
61          self.assertEqual(result, ['X', 'O']) # compare with expected results based on mock input
62
63          mocked_input.side_effect = ['o'] # create a mock input for 'o'
64          result = Tic.choose_marker() # capture results of the method we are testing
65          self.assertEqual(result, ['O', 'X']) # compare with expected results based on mock input
```

Figure 7

Testing that the chosen player markers are properly assigned

```
67      @unittest.mock.patch('tic_tac_toe.input', create = True)
68      # 05. test assign_marker()
69      def test_assign_marker_x(self, mocked_input):
70          """Test that the game markers are properly assigned to player and opponent."""
71          mocked_input.side_effect = ['x'] # create a mock input of 'x'
72          resulting_player, resulting_opponent = Tic.assign_markers() # capture return values
73          results = [resulting_player, resulting_opponent] # save results into single variable
74          self.assertEqual(results, ['X', 'O']) # compare with expected results
75
76          mocked_input.side_effect = ['O'] # create a mock input of 'o'
77          resulting_player, resulting_opponent = Tic.assign_markers() # caputre return values
78          results = [resulting_player, resulting_opponent] # save results into a single variables
79          self.assertEqual(results, ['O', 'X']) # compare with expected results
```

design, making the test cases appear to be essentially testing the same activities, however, one is designed to simply return the markers in their correct order (based on user input) while the other is designed to use those returned values to assign the appropriate markers to the main player and their opponent. Figure 7 demonstrates this in code above.

Display in Figure 8 below, our sixth and final test case for this section involves placing a player's game marker on their chosen square on the gameboard in its most simplistic form. In order to test the functionality of this method, we do not need to call any other functions or create a mock input as this method is use well after all of those other activities. Instead, we simply provided an example

Figure 8

Testing the replacement of our 1-9 strings with a player marker of X or O

```
112     # 09. test place_marker()
113     def test_place_marker(self):
114         """Test that the value of player become the value of the indexed board location."""
115         player = 'X'
116         row, col = 1, 1
117         Tic.place_marker(row, col, player)
118         np.testing.assert_equal(Tic.board[1][1], 'X')
```

Figure 9

Testing that our coin flip method will return one of two results (0 or 1)

```
81      # 06. test coin_flip()
82      def test_coin_flip(self):
83          "Test that either 1 or 0 is returned as a result of our coin flip."
84          heads_or_tails = Tic.coin_flip() # capture results
85          potential_results = [0, 1] # construct expected results list
86          self.assertIn(heads_or_tails, potential_results) # captured results is in expected results list
```

player variable (**'X'**) and example coordinates (**row, col = 1, 1**) in order to call **Tic.place_marker(row, col, player)** with all of its required parameters. We then asserted that the resulting **Tic.board[1][1]** would be equal to the value of **'X'**. During our gameplay loops, we use the returned values from our **get_coords()** method to initialize the **row** and **col** arguments.

3. Who goes first?

One requirement for our Tic Tac Toe game was for the first player to be randomly assigned. In order to meet this requirement, we designed a method called **coin_flip()** that has a randomized chance of returning either a **1** or a **0**. Within our gameplay loops, we then apply the returned value to an if/else statement that assigns first player to our main player (the one who got to choose their game marker) if the returned value is one or to our opponent if the returned value is zero. Our test method for the coin flip can be seen in Figure 9 above.

4. Obtaining Human & AI Moves and Their Coordinates

This final category of unit tests for our game is concerned with obtaining the desire move from a human player, obtaining the generated move for our AI player, and obtaining the coordinates that correlate with each of those activities. Figure 10 on the next page displays the code used to assess our random move generator.

Figure 10

Evaluating our random move generator

```
120     #10. test random_moves()
121     def test_random_moves(self):
122         """Test that the returned values is one of the gameboard's square numbers."""
123         result = Tic.random_moves()
124         self.assertIn(result, ['1', '2', '3', '4', '5', '6', '7', '8', '9'])
```

This test case, however, is not exactly comprehensive. This test script merely ensures that a valid move is returned based on an empty gameboard, not a mid-game board.

Figure 11 displays our penultimate test case, which assesses the function's ability to obtain a specified move from a human player that correlates with an open square on the gameboard. For this case we had to create a mock input as well as catch the expected printouts in order to isolate all of the method's behaviors – a statement which also applies to our **get_coords()** method test case, displayed in Figure 12 at the top of the next page. The mock input used for both of these test methods was **'5'** – which is equivalent to the middle square of our gameboard, so our expected coordinates are **1**, **1**.

Figure 11

*Evaluating the function of our **human_moves()** method*

```
88      @unittest.mock.patch('tic_tac_toe.input', create = True)
89      # 07. test human_moves()
90      def test_human_moves(self, mocked_input):
91          """Test that input equals returned value and is printed to console."""
92          with patch('sys.stdout', new=StringIO()) as output:
93              mocked_input.side_effect = ['5'] # create mock input of 5
94              player = 'X' # initilize method parameter
95              result = Tic.human_moves(player) # captures printed output and returned value
96          test_output = f"You chose square {result}!" # construct expected output
97          self.assertEqual(result, '5') # tests returned value
98          self.assertEqual(output.getvalue().strip(), test_output) # tests printed output
```

Figure 12

Testing obtaining the coordinates of a specified move

```
100     @unittest.mock.patch('tic_tac_toe.input', create = True)
101     # 08. test get_coords()
102     def test_get_coords(self, mocked_input):
103         """Tests that input is converted to proper board coordinates and captures console output."""
104         with patch('sys.stdout', new=StringIO()) as output:
105             mocked_input.side_effect = ['5'] # create mock input of 5
106             player = 'X' # initilize method parameter
107             results = Tic.get_coords(player) # captures printed output and returned values
108         test_output = "You chose square 5!" # construct expected output
109         np.testing.assert_equal(results, [1, 1]) # tests returned values
110         self.assertEqual(output.getvalue().strip(), test_output) # tests printed output
```

Results

The results from running our **test** module are displayed at the bottom of this page in Figure 13. All ten of our tests successfully met the conditions of their assertation statements. These test cases were created *after* the code was written for our program. In fact, our tests cases were created by using trial and error. We only tested methods that we already knew functioned properly due to our constant exploratory testing (i.e., playing the game repeatedly as it was developed to test for errors and exceptions). The unittest test runner failed many cases as we learned about creating mock inputs, capturing system outputs, the difference between NumPy assertation statements and Unittest statements, etc. Additionally, these unit tests did not cover the entire Tic Tac Toe program's classes or even the full set of **TicTacToe** class methods.

Figure 13

Unittest test runner results

```
PS C:\Users\Saman\Desktop\CS 504 (software engineering)\05. Independent Project\CS504-Project-Samantha-Hipple\Submission 3\TicTacToe v2.0> python test.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.019s

OK
```

Conclusion

There is much information left that can be explored concerning the topic of testing in software engineering. For example, one topic this report failed to explore is the concept of Test-driven development (TDD). TDD methods are becoming more common as time progresses. TDD methods aim to produce test scripts that meet project requirements *before* beginning the actual programming. As we continue to learn more about software testing and the creation of test scripts, we can move towards using TDD methods in the future more readily. Additionally, future research on this topic would be heavily focused on tools available to create automated testing processes for unit testing, integration testing, and regression testing as well as nonfunctional system testing tools and methods. The full code and development journey for our Tic Tac Toe game can be found on GitHub at https://github.com/Hipples/Tic-Tac-Toe.

References

Choudhary, A. (2020, September 24). *Continuous Testing: Complementary to Agile and DevOps*. Continuous Testing: Complementary to Agile and DevOps - DevOps.com. Retrieved May 30, 2022, from https://devops.com/continuous-testing-complementary-to-agile-and-devops/

IBM. (n.d.). *What is software testing?* What is software testing and how does it work? Retrieved May 30, 2022, from https://www.ibm.com/topics/software-testing

McPeak, A. (2017, August 8). What's the True Cost of a Software Bug? Retrieved May 30, 2022, from https://smartbear.com/blog/software-bug-cost/

Medewar, S. (2022, February 23). *Types of Software Testing.* Types of Software Testing: Different Testing Types with Details. Retrieved May 30, 2022, from https://hackr.io/blog/types-of-software-testing

MKS075. (2021, September 22). *Software Engineering | Software Quality Assurance.* Software Engineering | Software Quality Assurance - GeeksforGeeks. Retrieved May 30, 2022, from https://www.geeksforgeeks.org/software-engineering-software-quality-assurance/

Shaw, A. (2018, October 22). *Getting Started With Testing in Python*. Getting Started With Testing in Python – Real Python. Retrieved May 30, 2022, from https://realpython.com/python-testing/