**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 27, 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 ([source](https://www.ibm.com/topics/software-testing)). 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 ([source](https://www.ibm.com/topics/software-testing)).

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. Throughout this paper, we will be reviewing the way 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 test 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 ([source](https://www.ibm.com/topics/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 ([source](https://www.ibm.com/topics/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 ([source](https://www.geeksforgeeks.org/software-engineering-software-quality-assurance/)).

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 ([source](https://devops.com/continuous-testing-complementary-to-agile-and-devops/)).

**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.

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 software development life cycle (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” ([source](https://smartbear.com/blog/software-bug-cost/)). When the development process leaves ample room aside for testing, high-quality, reliable software applications can be delivered with few errors.

A picture containing diagram

Description automatically generated**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 ([source](https://hackr.io/blog/types-of-software-testing)).

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 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

**Figure 1**

*Types of Software Testing Diagram (*[*source*](https://hackr.io/blog/types-of-software-testing)*)*

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 whole system works as intended. Functional tests can be performed either manually or with the use of automated tools ([source](https://hackr.io/blog/types-of-software-testing)).

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 handle 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 ([source](https://hackr.io/blog/types-of-software-testing)).

**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, the game is working towards its second release. As features are added or modified, both unit test scripts and integration test scripts have been developed to ensure 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. Whie 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 and integration tests as well!

**Test Scripting 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 assertion methods from the **unittest.TestCase** class to complete the scripts. Both **nose**(**2**) and **pytest** are external testing libraries designed to support the execution of **unittest** test cases.

Our program’s testing focuses mainly on the use of Python’s built-in **unittest** module for both writing and running our test scripts. The following sections will review 10 separate test scripts that were written to test class methods within our **TicTacToe** class. This is in no way considered a comprehensive testing of our game, but instead a simple introduction to how test cases are written and how a test runner works.

**Creating and Running unittest Scripts**

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

**Create the Gameboard:**

The first step in creating our unit tests was to create 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 through our test cases (e.g., line 17 in Figure 2: **Tic.create\_board()**). Then, we created a second variable that held our expected output, **test\_board**, and initialized it to hold values 1-9, as strings, shaped into a 3x3 grid.

Lastly, **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 gameboard’s data type is actually a **numpy** ndarray and produced an error that suggested the use of **numpy**’s built-in array testing methods instead. Figure 2 below displays the code used to create our first test script.

**Figure 2**

**Text

Description automatically generated***Test Script for TicTacToe’s create\_board() method*

**Figure 3**

*Test Script for TicTacToe’s display\_board() method*

**Text

Description automatically generated**

**Display the Gameboard:**

For our second test case, we wanted to test if our **display\_board()** method is functioning 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 automation in the future, 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 from our **unittest** module.

We then compared the captured output’s value to a newly created, long-hand version, of our gameboard display method (i.e., attempted to essentially draw the expected display line by line) as demonstrated in Figure 3 above using the same lines of code from our first unit test to create the test board.

**Figure 4**

*Test Script for TicTacToe’s reset\_board() method*

Text

Description automatically generated

**Reset the Gameboard:**

This third unit test is the easiest one so far, we simply test the effect of our **Tic.reset\_board()** on the **Tic.board** property against our **test\_board**, which has been redefined as an empty list. The full code for this test is displayed above in Figure 4.

**Randomly Choose First Player:**

Our fourth unit test focuses on the method we use to randomly assign which player goes first in our game called **coin\_flip()**. During gameplay, we use this method by assigning **first\_player** as the main player if the result is a 1, or as the opponent if the result is a 0. So, we assert that the returned value must be either 1 or 0 to pass our test. Figure 5 below displays our test code in full for this method.

**Figure 5**

*Test Script for TicTacToe’s coin\_flip() method*Text

Description automatically generated

**Figure 6**

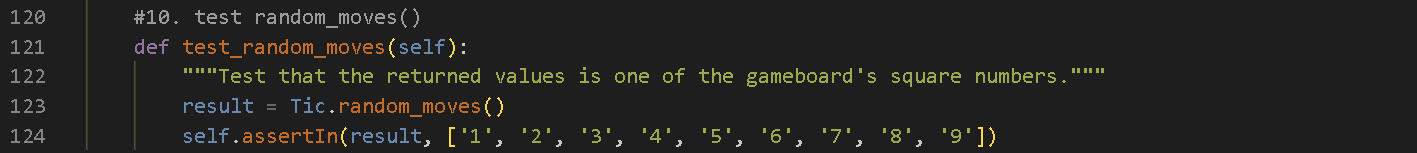
*Test Script for TicTacToe’s place\_marker() method*

**Text

Description automatically generated**

**Place Marker on Chosen Square:**

**Generate Random Moves for RandAI:**

****

**Player Chooses Game Markers (X or O):**

Text

Description automatically generated

**Assign Game Markers Properly:**

Text

Description automatically generated

**Get Gameboard Coordinates for Desired Marker Placement:**

Text

Description automatically generated

**Complete a Human Player’s Move:**

Text

Description automatically generated

**Conclusion**

There are many, many different aspects involved in the testing phases of software engineering, no one paper could ever do them justice.