**SE463 Assignment #5 Report**

Building An Intelligent Tic-Tac-Toe Game By Incrementally Testing Its Full-Stack Integration

By: Elizabeth Dixon

**Table of Contents**

* Pg. 3-4: Project Overview & Design
* Pg. 5-6: Test Cases & Outcome Analysis

* Pg. 7-8: Findings & Testing Concepts Reflection

**Project Overview & Design**

The first thing I thought of when planning how to go about building this project with its limitations in mind was that I will have to write each functionality bit by bit to be successful. With something like game logic (in this case tic-tac-toe) that, more often than not, contains multitudes of conditional statements and lots of functions that are fairly similar in nature, it’s important to recognize the small differences that make the game tic as intended. In order to do that, each new function of the game must be tested as it is written, something that is good practice anyways when writing code anyways but feels especially applicable in a situation like this. With that being said, in terms of Integration types of testing, to develop this tic-tac-toe game, I landed on using the Incremental Integration Testing method and, by association, would find great use in applying what I have learned about Unit Testing as well. Without being aware, Incremental Integration Testing is a concept that I have become increasingly familiar with as I have gone through a class such as SE361 where Incremental Integration Testing often comes with the territory of the Agile and Scrum Methodologies (not to mention the crossover between Front-end and Back-end development). All that to say, I thought it was great practice to utilize my prior knowledge from a perspective that is new to me - that being the concept of Integration Testing.

Another incremental way of thinking was implemented in my work in CS341, wherein there was heavy focus on classes and objects the best ways that they should be organized. So, I began by thinking about all of the different functions that might be needed *apart* from some of the overarching variables such as an enumeration for the type of player being manipulated and constant characters for said players’ game pieces (X and O). I began by writing a void Game::printBoard() function and the Game::Game() object initialization because I knew that there wouldn’t be any crossover between other functions for their bases. This was essentially the same idea for the implementation of Game::checkWin(Player player), but many more variables had to be considered afterward - although, checkWin soon became one of the most important functions of the entire program and was implemented in nearly each function thereafter. The intelligence of the computer player (CPU), using a simple minimax algorithm, was definitely difficult to figure out at first, but became easier when I realized that the miniSearch and the maxSearch (after the minimax Move reference was defined) were essentially the same thing but opposites. getHumanMove() and play() had more to do with the UI - or how the user interacted with the terminal rather than the other more Back-end heavy functions (save printBoard()). Overall, the Integration aspect of this project was not as daunting as I initially thought, and using Incremental Integration Testing definitely helped with that ease.

**Test Cases & Outcome Analysis**

As far as test cases go, as mentioned before, I essentially made them up as I was incrementally going along, with a pretty Unit Testing mindset. I was quickly reminded of the first BVA assignment, where I had to keep remembering that there were outcomes or inputs that I had to remember to not include, which was important here - especially in the many conditionals that I had to implement. For example, when I was first building printBoard(), or anything that included that characters ‘-’ or ‘|’ to build the board, I had to make sure that the board was even, that the board array(s) I was using truly made the amount of squares necessary as deemed by the program requirements, and so on. I had to think of things like “What if the loop index for printBoard() was < 4 - would that change the outcome undesirably?”. Obviously this simple base UI aspect was one of the less complicated aspects I had to test whereas the checkWin functionality was definitely the most difficult. This was because it had as much to do with the board as it did with how the players interacted with each other - making the Integration aspect more important than ever. As further examples, I had to make sure that each player was doing what they were supposed to do and not what the other player was supposed to do, as well as giving the functionality of one player to the other to see what it would do - highlighting the frustration of writing a test case that I knew was going to fail. So, as a bit of a failsafe (but not really), I made sure to set the player instances of the Player object as the enumeration of AI and HUMAN as well as indicated a playerType so that the game pieces could be properly referenced. Then came the “checkers” for the sequences of moves made by each player wherein I checked for horizontal, vertical, and diagonal sequence behavior and returned the boolean checkWin as necessary. This aspect was definitely the most Unit Testing-based way of working that I had to employ as there are so many different outcomes and possibilities in tic-tac-toe for each player.

Additionally, I also had to make sure that the minimax algorithm actually worked in an intelligent way and not just in one way (i.e. always going in the same space when it should be following what the player does).

\* There is a more comprehensive list of test cases I could exhibit here, but this is the most significant overview.

**Findings & Testing Concepts Reflection**

In general, I found this project to be a very good final code-writing project, as it has employed many of the ways of thinking that has been taught to us this semester. Additionally, it seems to be the most applicable in general - whether out in industry or in the Butler CSSE classroom. I found that, among other testing methods, Incremental Integration testing has to keep the developer very honest, which is true for any testing method, but this testing method makes shortcomings exponentially more obvious which has its pros and cons. On one hand, it’s easier to see one’s shortcomings when developing a program, but on the other you have to worry about remedying said numerous shortcomings. Although it was difficult to carry this project out, as I had to be the oracle in some ways, it has helped me to become a better software tester.

Although through possibly unnecessary repetition (like BVA) and sometimes not-so-good coding practices, I have a more nuanced perspective and understanding of software testing after building this tic-tac-toe program - more so than in previous assignments for this class. As mentioned before, Integration Testing keeps the developer honest - and it just goes to show how much the Front-end and Back-end can come to rely on each other when writing code in general, but especially user-friendly code. Additionally, a project like building a tic-tac-toe game requires a lot more focus on code semantics as opposed to previous projects where the “rules”, for lack of a better word, were far more cut and dry (like in path testing - how the path is simply a representation of a program). Yes, tic-tac-toe has rules too but there is no room for overstepping or error when it comes to its conditions - particularly when the user is not fully in control of one of the players and has to work as intended essentially on its own. Similarly to many thoughts presented before, this is applicable to so many situations concerning building full-stack programs using continuous testing, but tic-tac-toe is just a simple and very applicable example. Another important lesson I learned was that it’s useful to use multiple testing methods in tandem. For example, Unit Testing’s scale is a lot less daunting when it is used in increments, so I can definitely use that concept for future reference and comprehension of software testing methods.