10/16/2017

Brady Brettin, Matthew Deremer, Luke Pace

Refactoring of Battleship Game

Project 2 Report

Table of Contents

[Table of Figures 2](#_Toc495922160)

[Overview 5](#_Toc495922161)

[Project Setup 6](#_Toc495922162)

[Construction of Test Suite 8](#_Toc495922163)

[Initial Setup of Methods for Test Suite 11](#_Toc495922164)

[Creation of Classes and Methods 16](#_Toc495922165)

[Battle Ship Class 17](#_Toc495922166)

[Placing Ships and Targeting 19](#_Toc495922167)

[Fire at Opponent Ship 21](#_Toc495922168)

[Count and Print Hits 24](#_Toc495922169)

[Playing the Game 26](#_Toc495922170)

[References 36](#_Toc495922171)

# Table of Figures

[Figure 1: GitHub 6](#_Toc495922181)

[Figure 2: GitHub 6](#_Toc495922182)

[Figure 3: GitHub Online 7](#_Toc495922183)

[Figure 4: Visual Studio Project 7](#_Toc495922184)

[Figure 5: Battleship Test Setup 8](#_Toc495922185)

[Figure 6: Battleship Test 9](#_Toc495922186)

[Figure 7: Battleship Test 10](#_Toc495922187)

[Figure 8: Battleship Test 10](#_Toc495922188)

[Figure 9: InitializeBoards() and SetupBoards() 11](#_Toc495922189)

[Figure 10: Fire() and CheckWin() Methods 12](#_Toc495922190)

[Figure 11: Boards and Message 12](#_Toc495922191)

[Figure 12: PrintBoards() Method 13](#_Toc495922192)

[Figure 13: Failed Test 14](#_Toc495922193)

[Figure 14: Failed Test 15](#_Toc495922194)

[Figure 15: Failed Test 16](#_Toc495922195)

[Figure 16: Import Libraries and Initialize Methods 17](#_Toc495922196)

[Figure 17: Battleship Class 18](#_Toc495922197)

[Figure 18: setShip() Method 19](#_Toc495922198)

[Figure 19: validShipPlacement() Method 20](#_Toc495922199)

[Figure 20: validTarget() Method 20](#_Toc495922200)

[Figure 21: Original Fire Code 22](#_Toc495922201)

[Figure 22: Fire() Method 23](#_Toc495922202)

[Figure 23: Fire Method() 24](#_Toc495922203)

[Figure 24: countHits() Method 25](#_Toc495922204)

[Figure 25: printHits() Method 26](#_Toc495922205)

[Figure 26: playGame() Method 27](#_Toc495922206)

[Figure 27: Original Main() Method 28](#_Toc495922207)

[Figure 28: Original Main() Method 29](#_Toc495922208)

[Figure 29: Main() Method 29](#_Toc495922209)

[Figure 30: InitializeBoard() and PrintBoard() in action 30](#_Toc495922210)

[Figure 31: Set and Validate Ships for Player 1 31](#_Toc495922211)

[Figure 32: Player 1 Board with Ships Placed 31](#_Toc495922212)

[Figure 33: Set and Validate Ships for Player 2 32](#_Toc495922213)

[Figure 34: Player 2 Board with Ships Placed 32](#_Toc495922214)

[Figure 35: Sample Gameplay 33](#_Toc495922215)

[Figure 36: Sample of Winning 35](#_Toc495922216)

# Overview

When constructing a software project or piece of code, design and quality is just as important as functionality. The is especially true when working on large scale projects. In the first iteration of a project, it is easy to ignore the design of code and focus on functionality. While the code may function as intended, it may be long, complex, and hard to read. To fix these issues, the code must be refactored. Refactoring is the process of restructuring working code to reduce the number of lines, decrease complexity, and improve readability. Some examples that hint at the need for code to be refactored are: long methods, duplicate code, switch statements, and data clumps. These issues can be resolved in many ways including: composing methods, organizing data, and simplifying expressions. To demonstrate some of these refactoring process, our team found some source code for a Battleship game that was in desperate need of refactoring. The original code was approximately 28,000 lines filled with duplicated code, poor naming conventions, few methods, and no test. In the report below, we detail the construction of a test suit for the application and the refactoring done to reduce the code to 233 lines.

# Project Setup

Our team found the source code used for this project from an open source site. After the code had been downloaded, we pushed it to our team repository on Github, shown in Figures 1 through 3 below, so all team members could collaborate on the project.

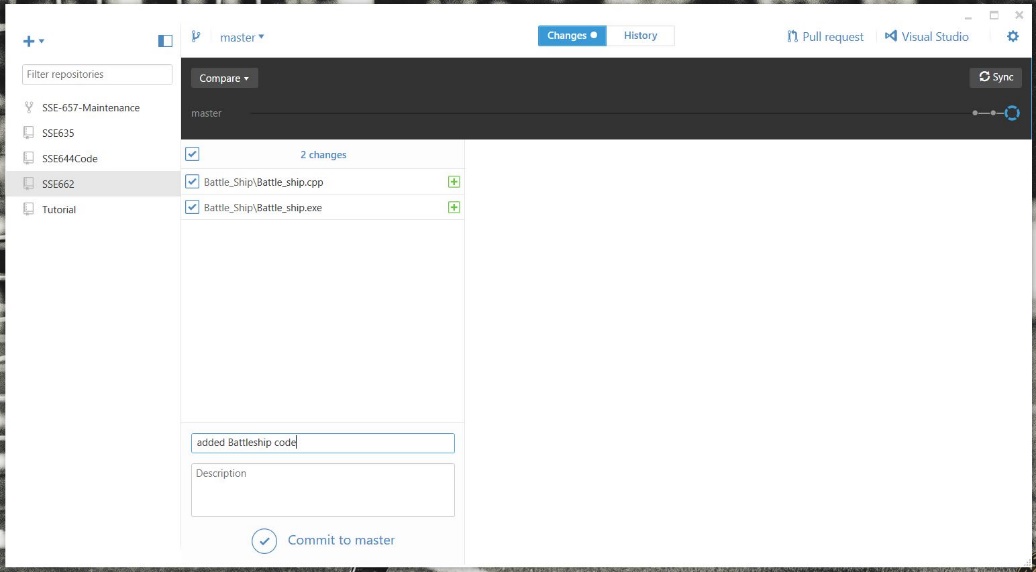


Figure 1: GitHub

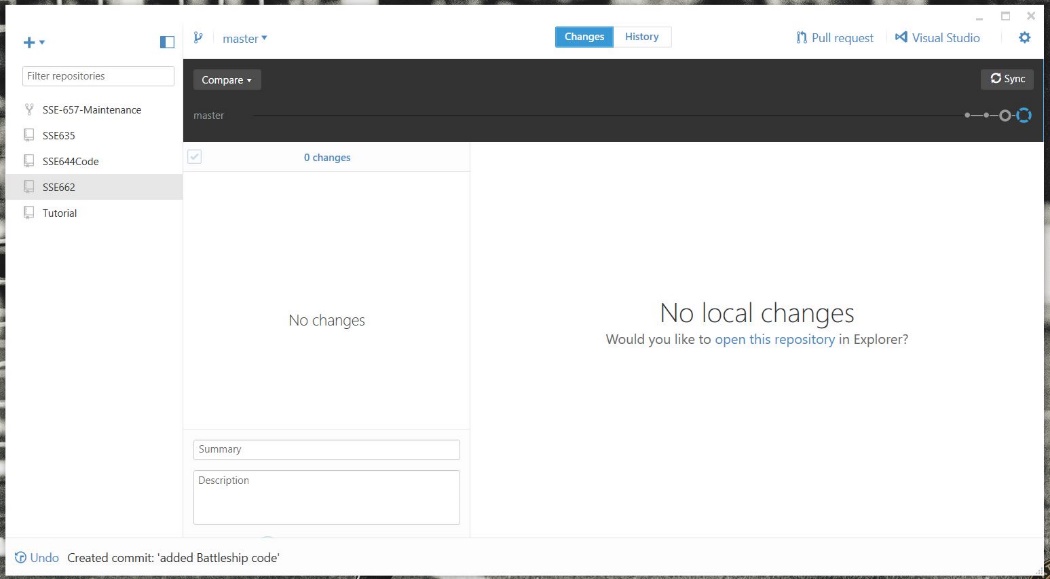


Figure 2: GitHub

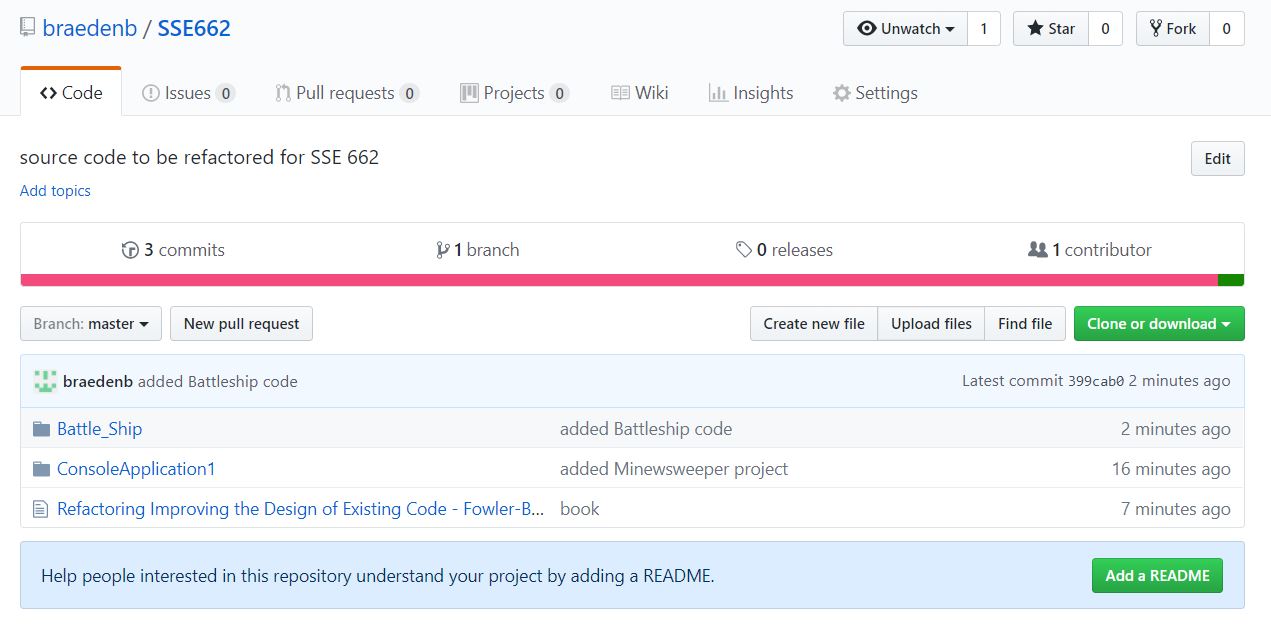


Figure 3: GitHub Online

Once the source code had been pushed to GitHub, we created a project in Visual Studio, shown in Figure 4 below, to connect the Battleship application to the test suite which is explained in the next section.

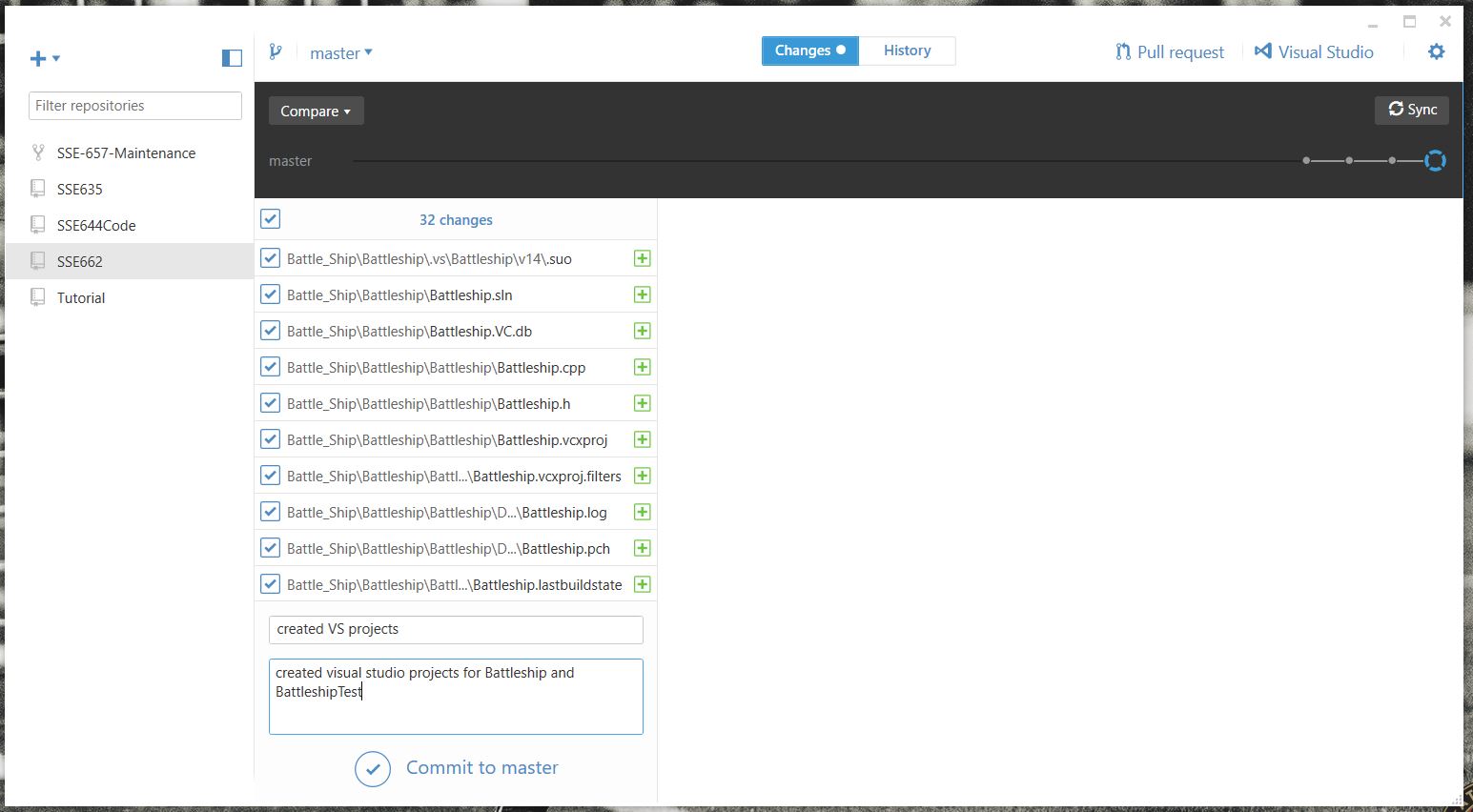


Figure 4: Visual Studio Project

# Construction of Test Suite

Before we could begin writing unit test, a test file had to be created and connected to the Battleship code as shown in Figure 5, below.

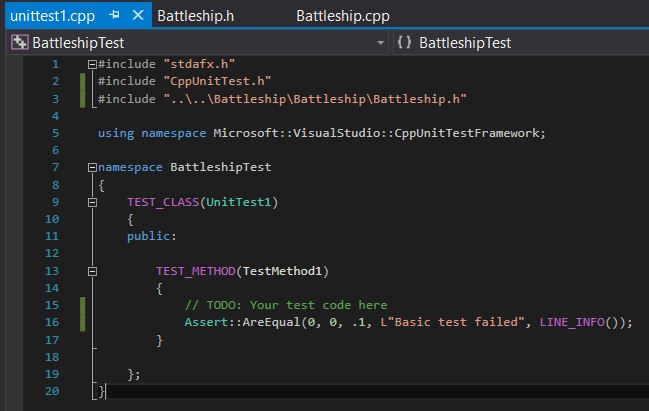


Figure 5: Battleship Test Setup

A cursory examination of the source code showed a major missing component for any refactoring: unit testing. Testing provides a means through which we can confirm that any refactoring done to the project does not affect the overall functionality and usability of the project. By providing a test suite to run against the various methods in the project, we can quickly and efficiently narrow down any errors in the code. The test suite is shown in Figures 6 through 8, below.

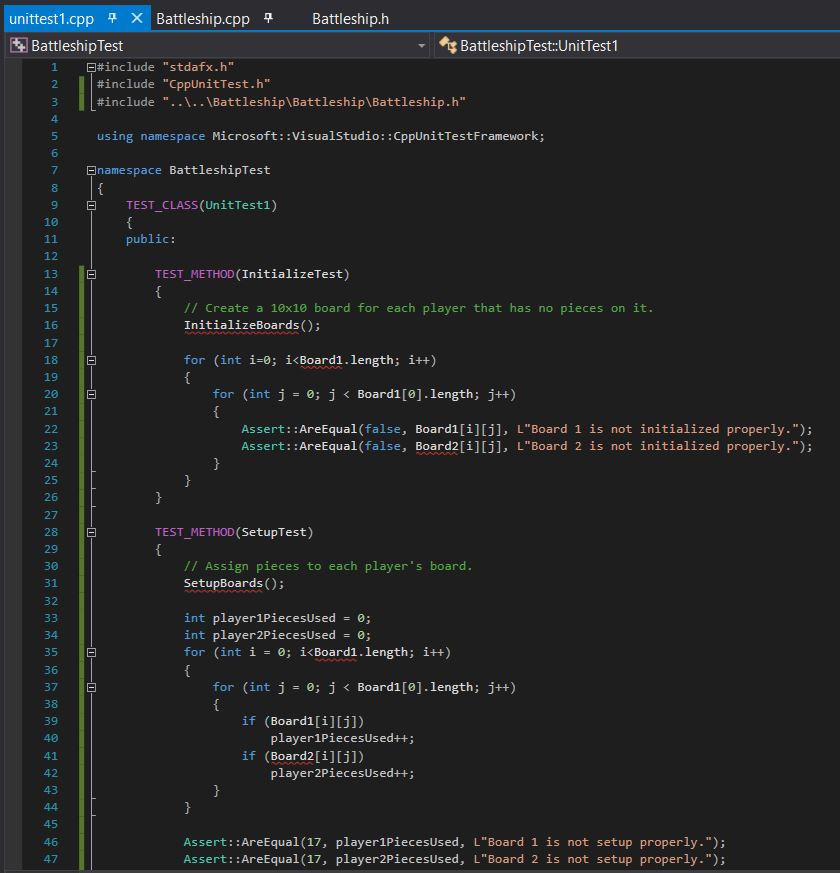


Figure 6: Battleship Test

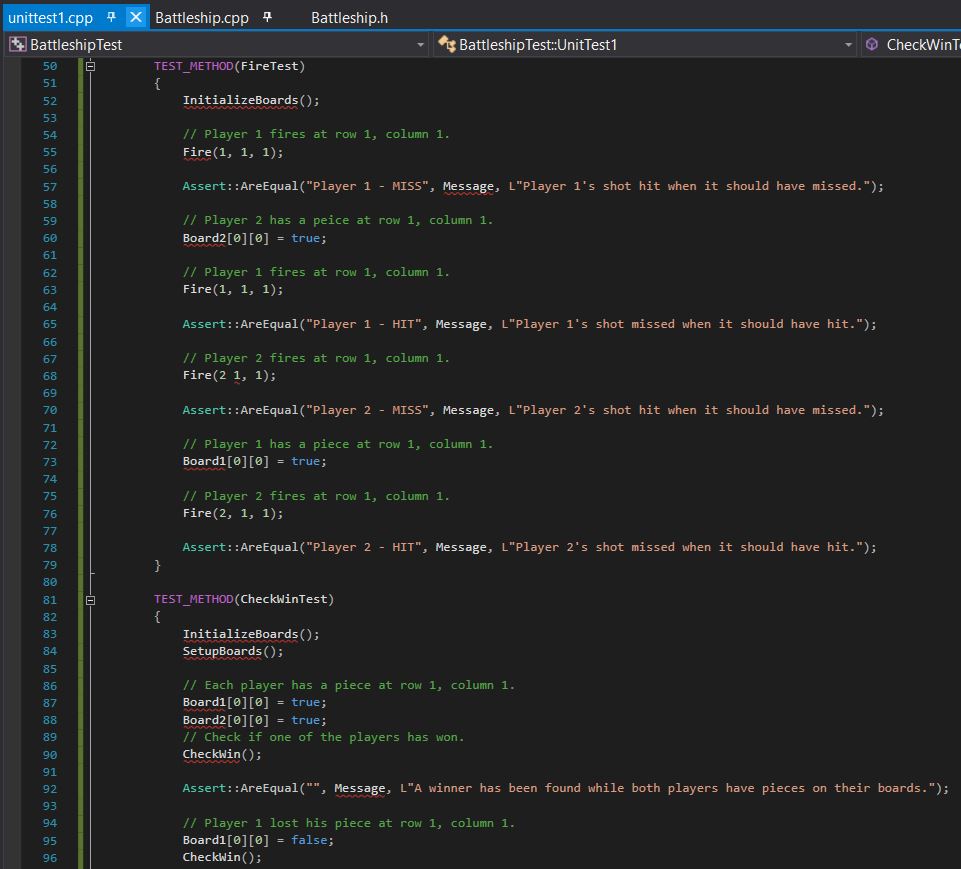


Figure 7: Battleship Test

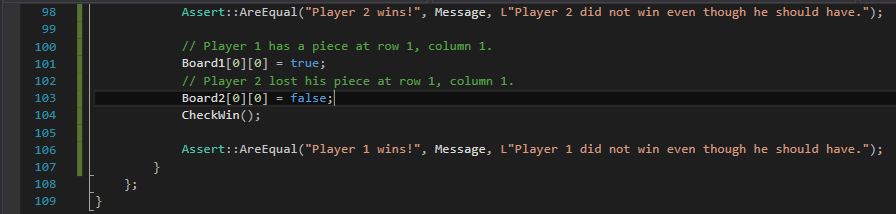


Figure 8: Battleship Test

# Initial Setup of Methods for Test Suite

Given the expansive, naïve nature of the source code, we created the tests in the previous section with visions of how we would like the code to be organized, rather than tests that would work with the current source code. As a result, we created several new objects and methods that we would like to see in the refactored version of the code. As a result, however, this test suite cannot be compiled until these new objects and methods are written. To begin, we created methods for InitializeBoards(), SetupBoards(), Fire(), and CheckWin() shown in Figures 9 and 10, below.

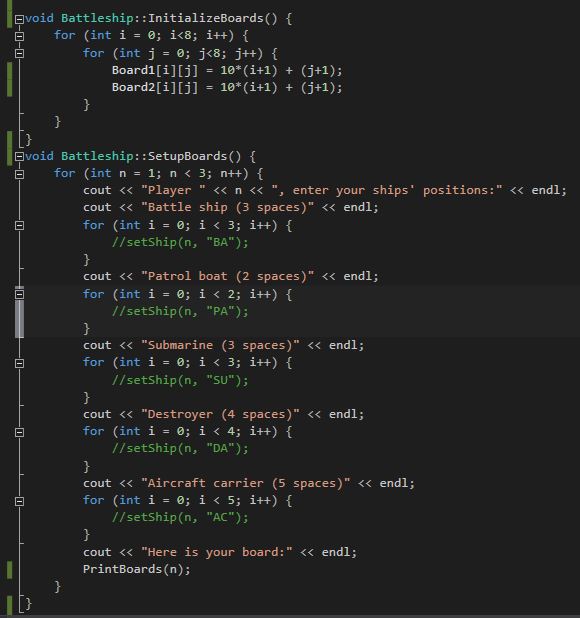


Figure 9: InitializeBoards() and SetupBoards()

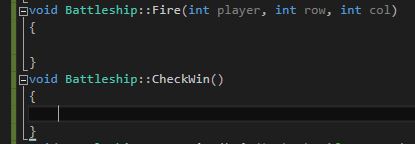


Figure 10: Fire() and CheckWin() Methods

Next, we refactored the code to use a multi-dimensional 8x8 array of Booleans, rather than the array of integers currently being used. This refactoring improves the performance and execution time of the code and better models the real-world example of a Battleship game. A Message string, which would hold output to the console, was also created as shown in Figure 11, below.

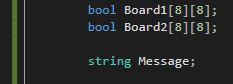


Figure 11: Boards and Message

In the process of creating our tests, we also decided that it would be beneficial to have a method that would print the contents of the boards. Such a method would allow the user to easily view the contents of either board at the request of the user. As such, we created a PrintBoards() method shown in Figure 12, below.

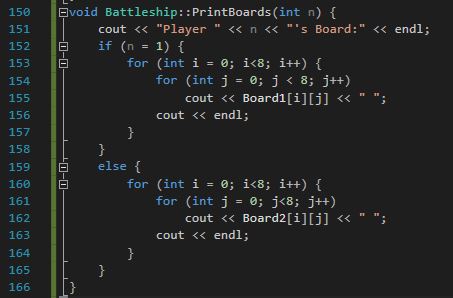


Figure 12: PrintBoards() Method

With these three major changes, the test suite can now compile. All that remained was to add the ‘Battleship.cpp’ file to the test project. Running the tests should produce the failed outputs shown below in Figures 13 through 15. We purposefully fail these tests to model the “red, green, refactor” method of refactoring and test writing.

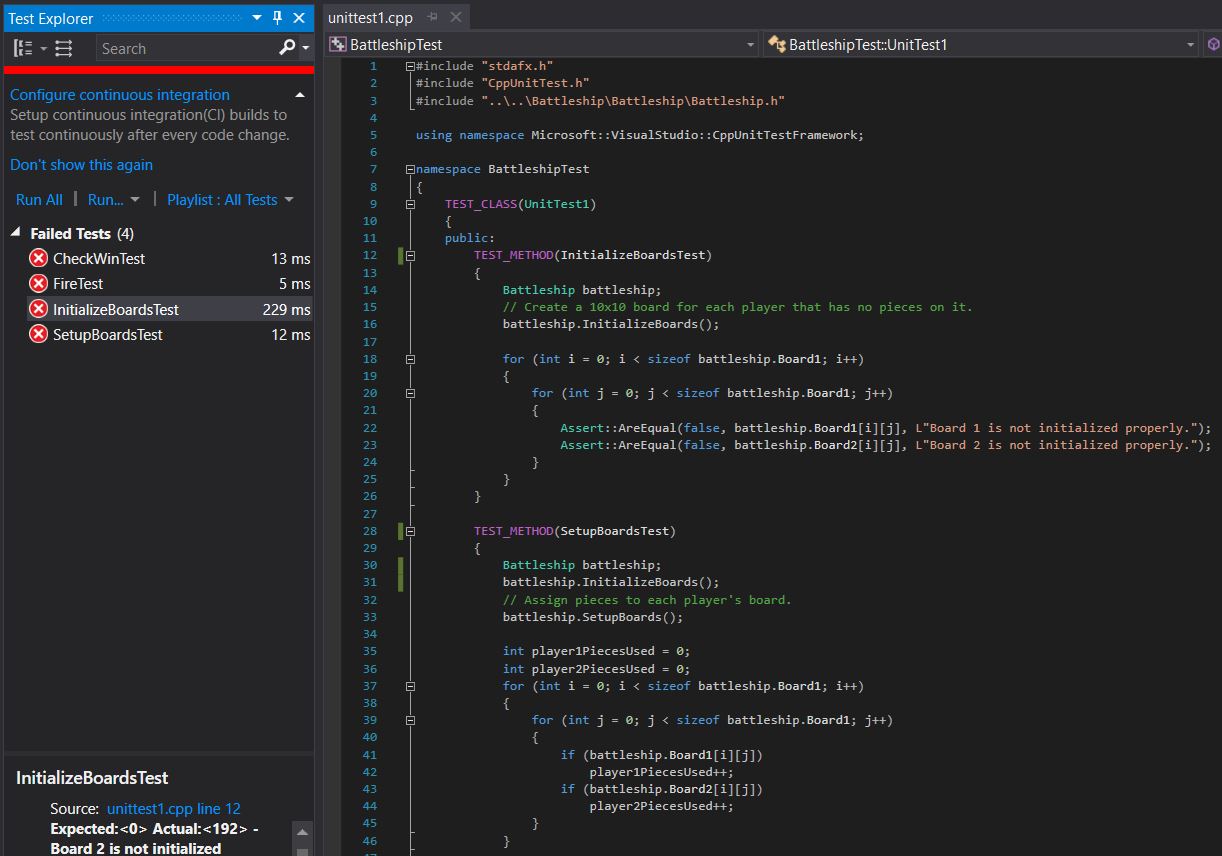


Figure 13: Failed Test

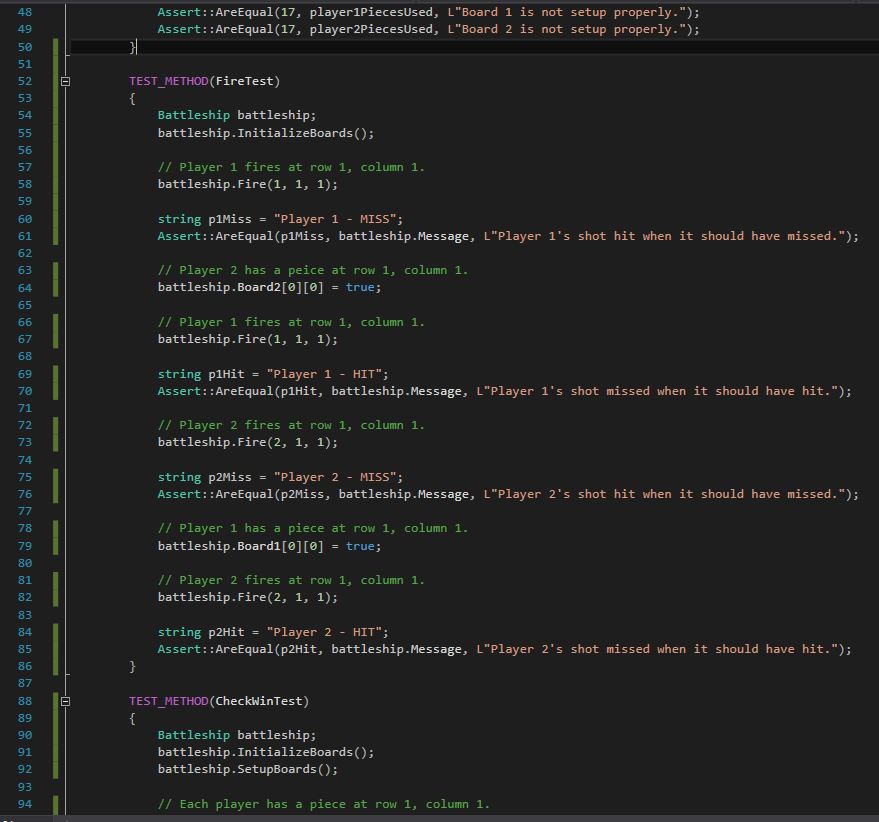


Figure 14: Failed Test

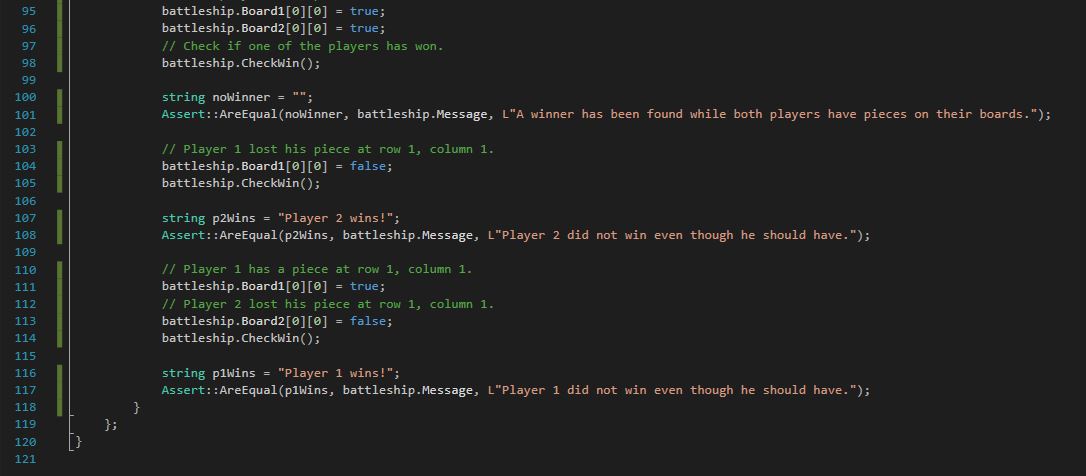


Figure 15: Failed Test

# Creation of Classes and Methods

Before we began any more refactoring, we needed to edit the header to include all the required libraries and layout the methods we thought would be needed to simplify the code. The newly designed header is shown in Figure 16, below.

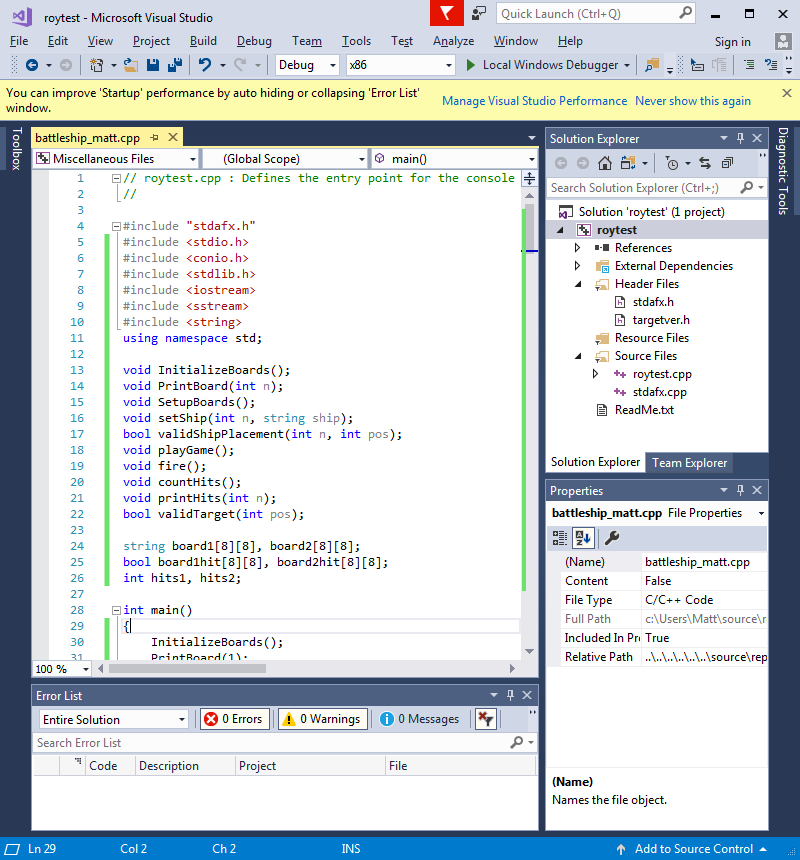


Figure 16: Import Libraries and Initialize Methods

## Battle Ship Class

C++ allows a user to include a header file to be used in testing, which holds a class that includes all its variables and methods. Unfortunately, the source code had not created a class for the application, so we created a Battleship class to hold all the variables and methods shown in Figure 17, below.

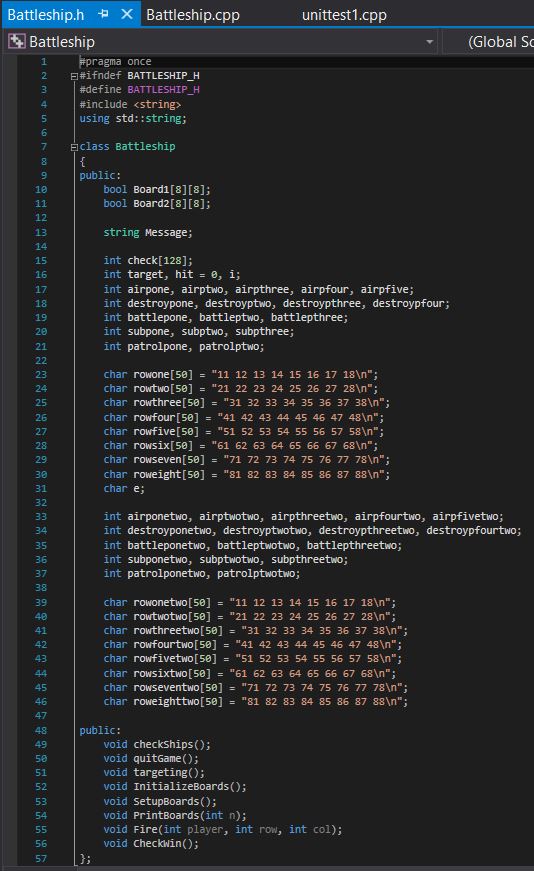


Figure 17: Battleship Class

## Placing Ships and Targeting

We then needed to create a method to place a ship on the board. Originally, this was done by many repeating lines of code in the main method. While constructing this method, we also realized we needed a method to check if a ship was being placed in a valid position. A valid position is one that is on the board and not occupied by another ship. These methods are shown in Figures 18 and 19, below.

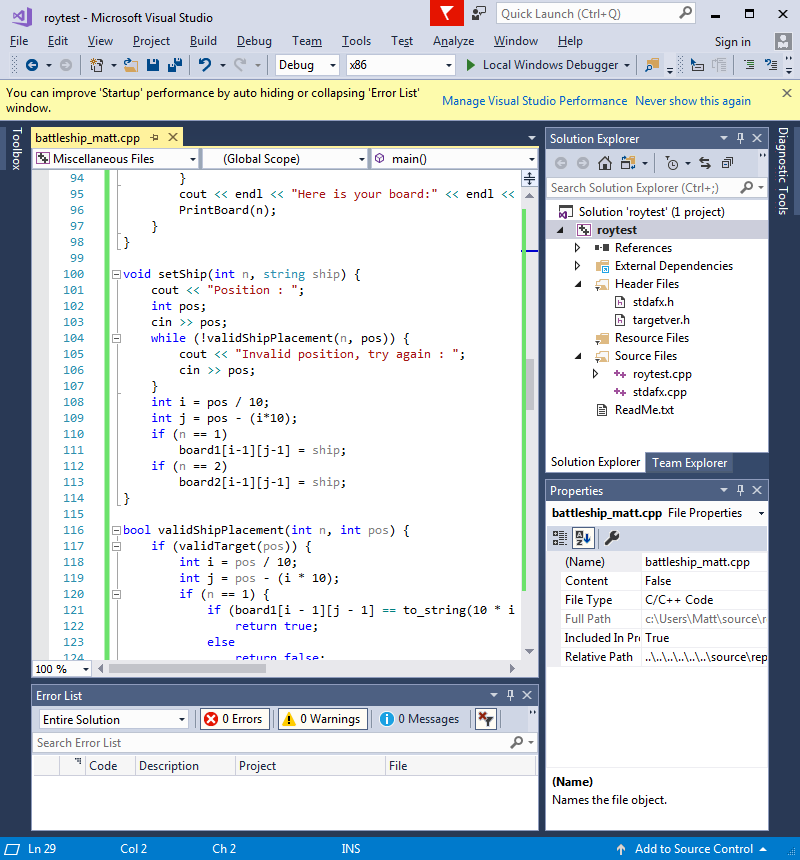


Figure 18: setShip() Method

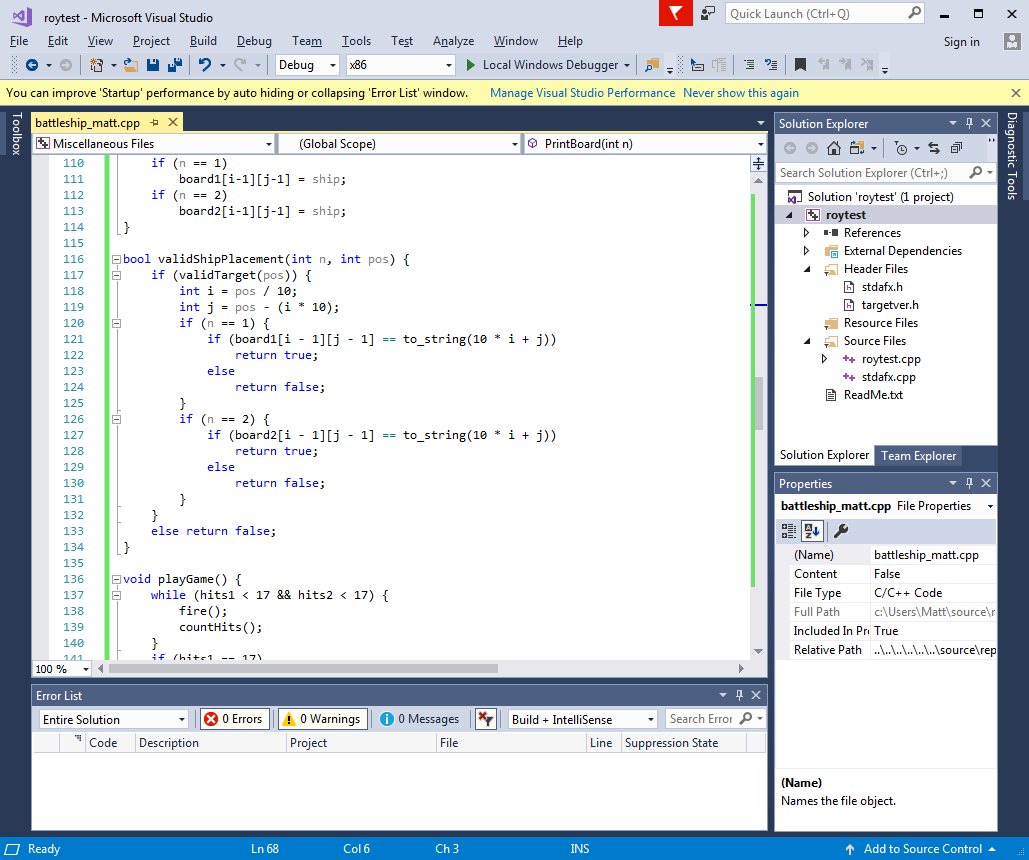


Figure 19: validShipPlacement() Method

The next method that needed to be created was validTaget(). This method, shown in Figure 20, would check that the position a player entered to fire at was a valid position on the board.

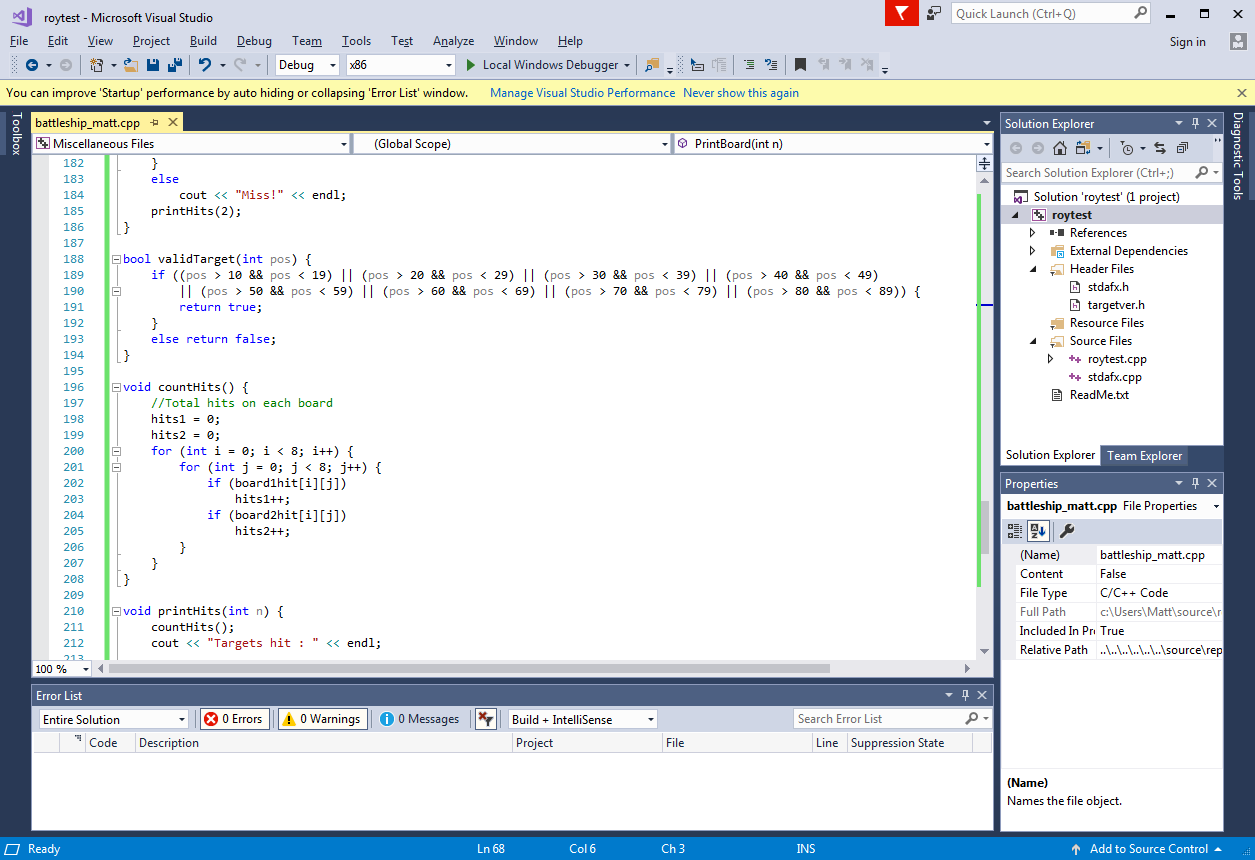


Figure 20: validTarget() Method

## Fire at Opponent Ship

We then moved on to the largest refactoring done for this project. At this point, the boards for each player had been created and printed, ships could be placed on valid positions, and ships could be targeted. Now we needed to create a method to allow a player to fire at a position on the opponent’s board. The original code, a sample is shown in Figure 21 below, was thousands of lines of switch statements. This is highly inefficient, hard to read, and slow.

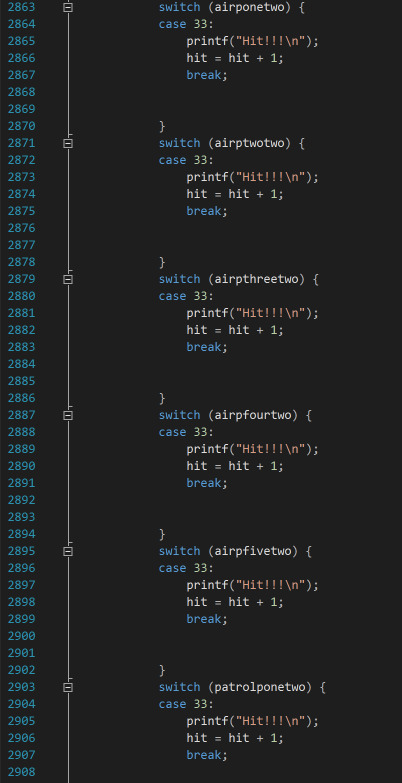


Figure 21: Original Fire Code

To improve this code, we created a Fire() method, shown in Figures 22 and 23 below. By creating this method, we were able to vastly reduce the number of lines of code and improve the readability.

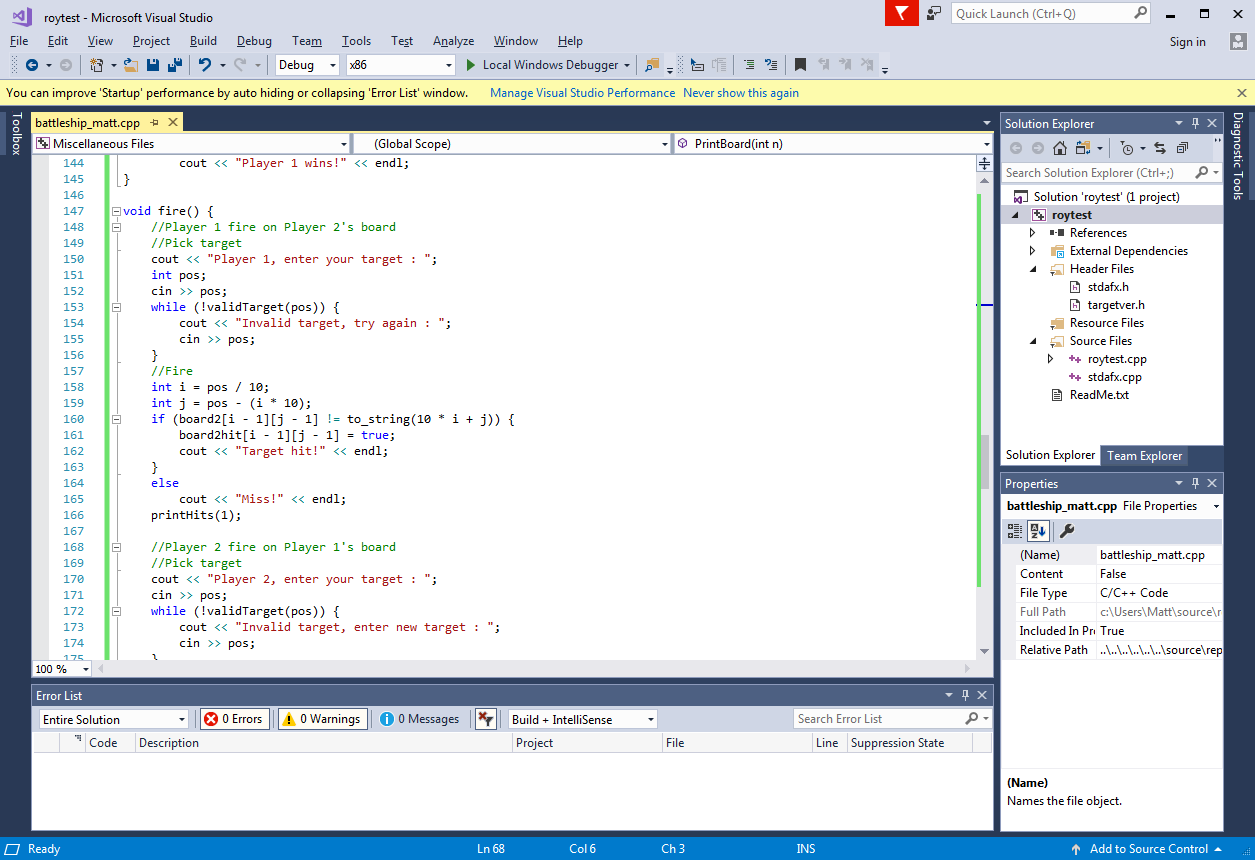


Figure 22: Fire() Method

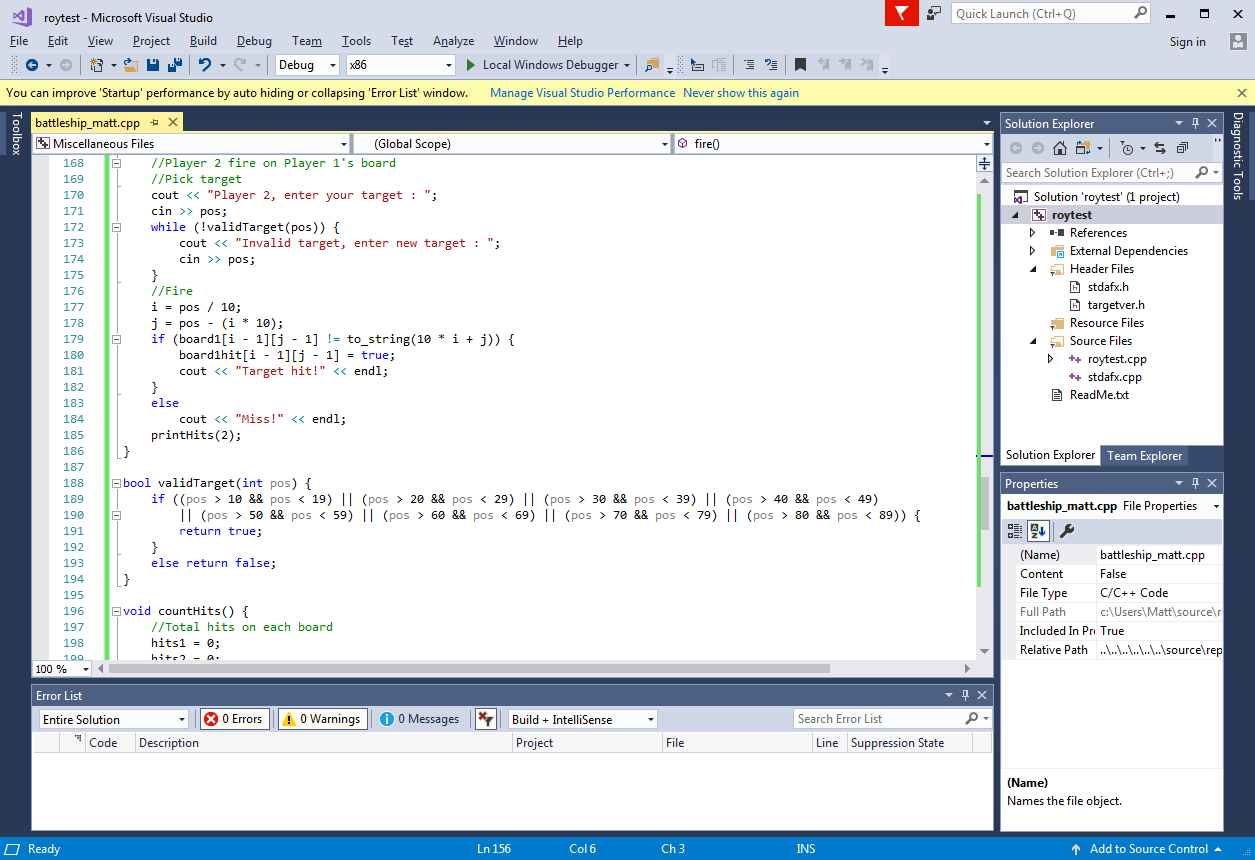


Figure 23: Fire Method()

## Count and Print Hits

After writing the method to allow a player to fire at the opponent’s ships, we needed to construct methods to count how many times a player’s ship had been hit, Figure 24, and to print the number of hits made by each player, Figure 25. The countHits() method is a vital part of the game because it is used to determine who wins. A player wins by sinking all the opponent’s ships.

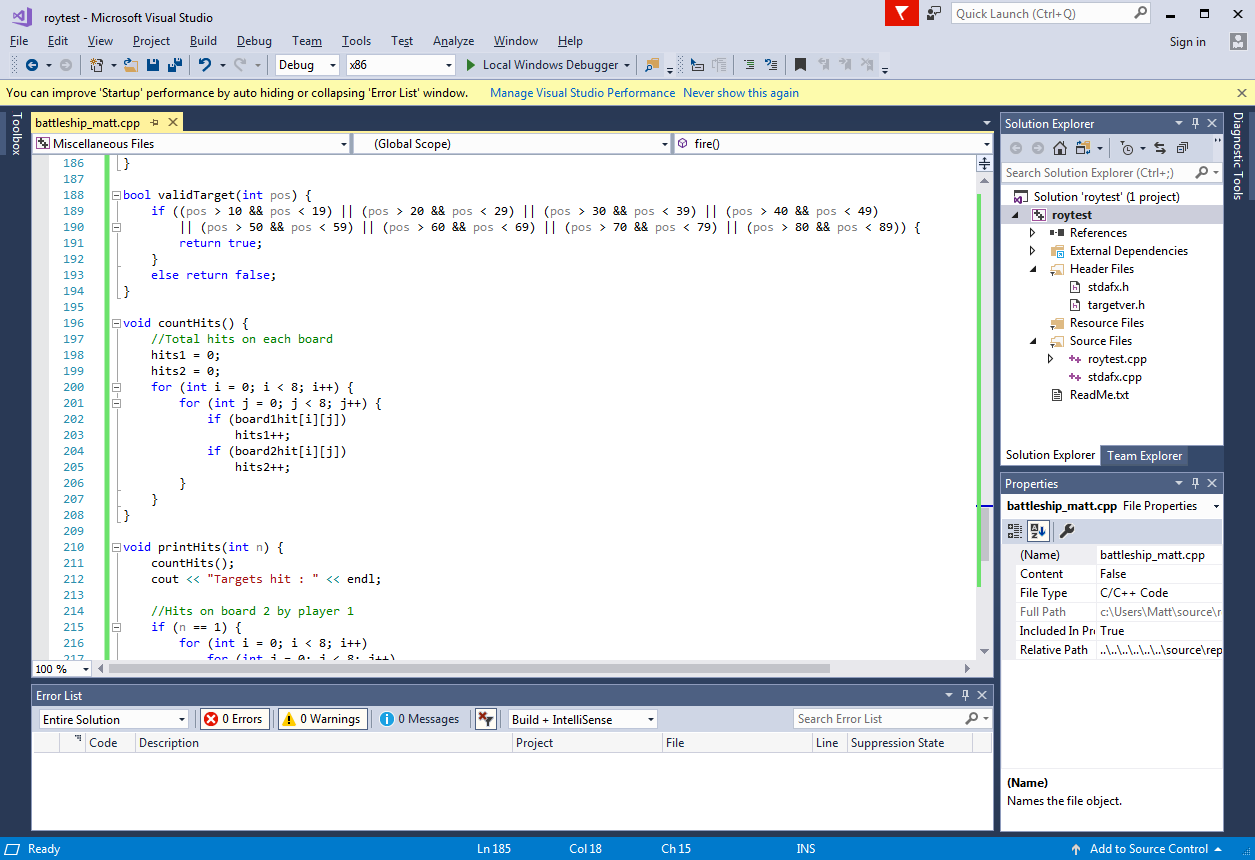


Figure 24: countHits() Method

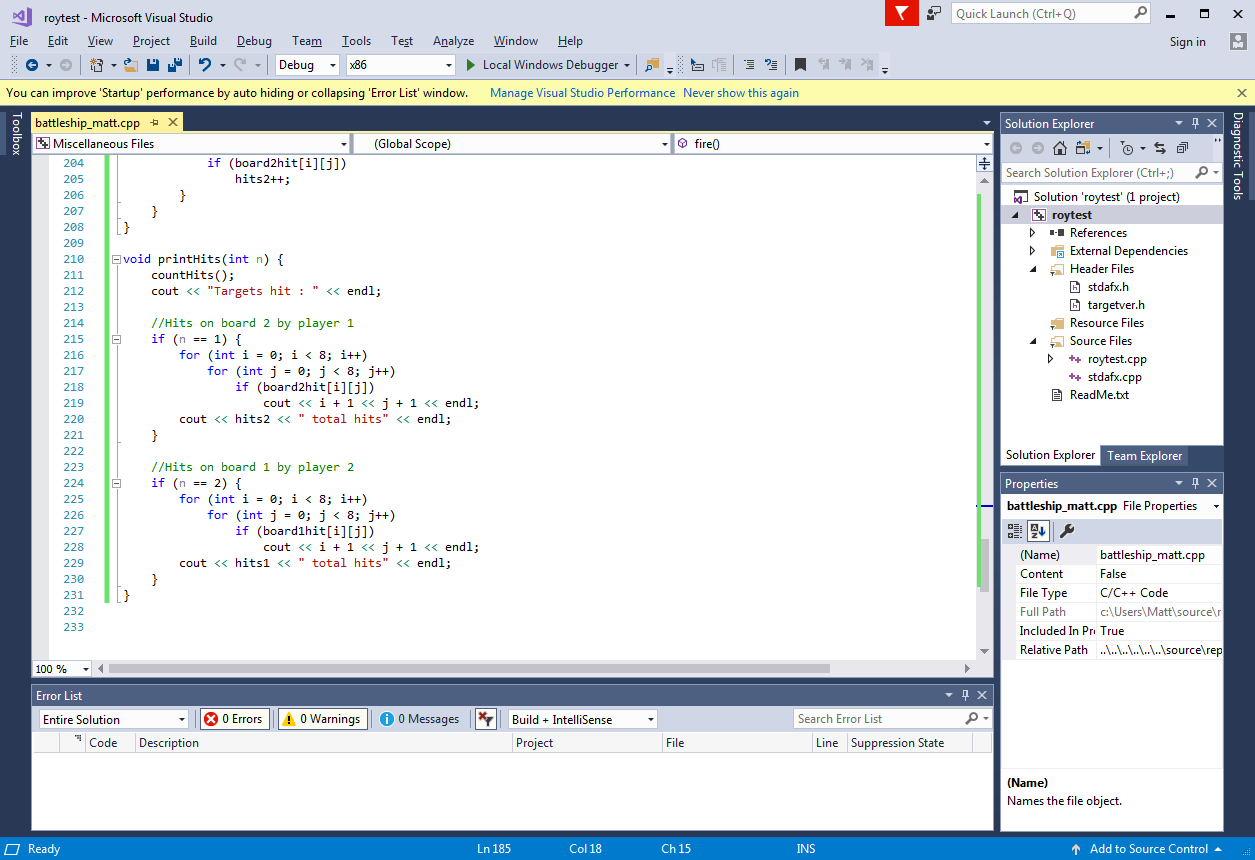


Figure 25: printHits() Method

## Playing the Game

Now that all the functionality of the game had been refactored into short, easy to read methods, we need to construct a method, playGame(), to utilize the other methods. The playGame() method can be seen in Figure 26, below.

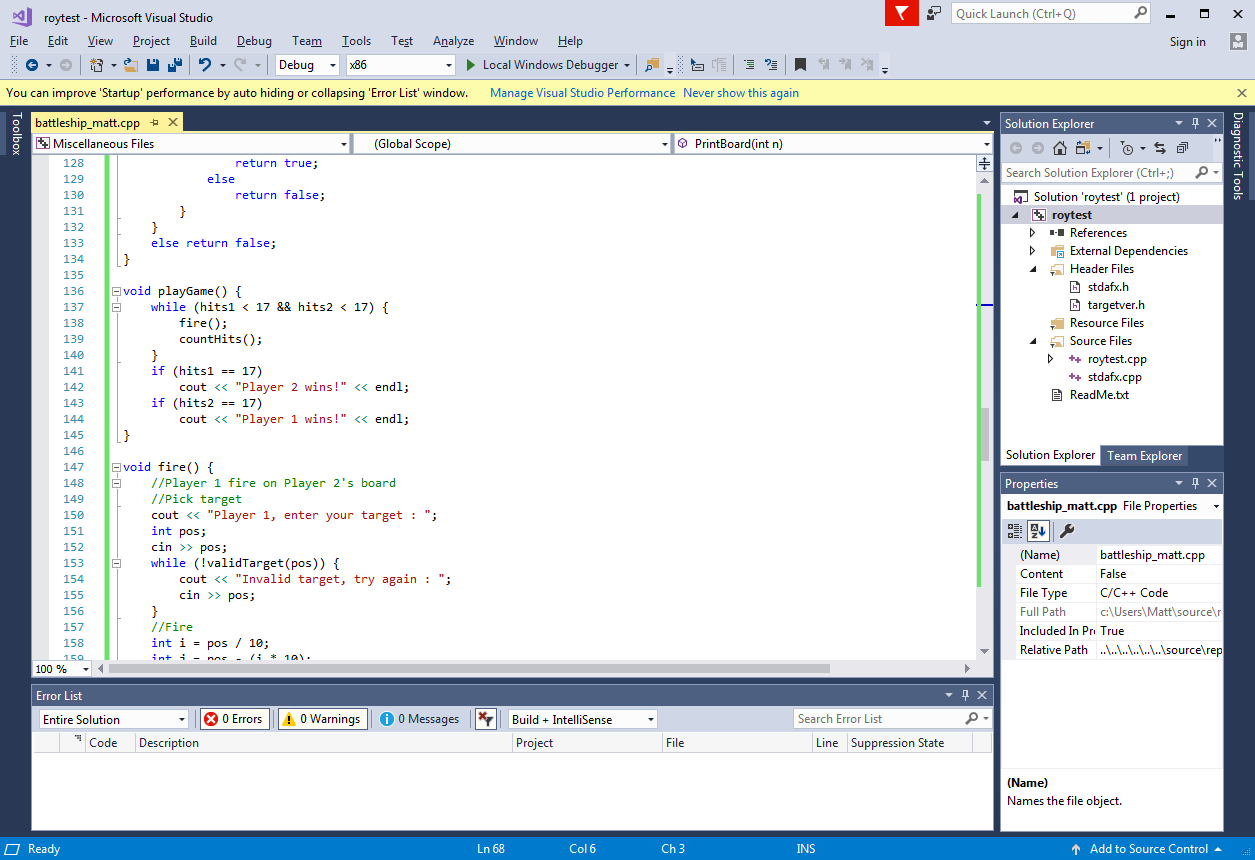


Figure 26: playGame() Method

In the original main method, shown in Figures 27 and 28 below, there was repetition of code and it was hard to read.

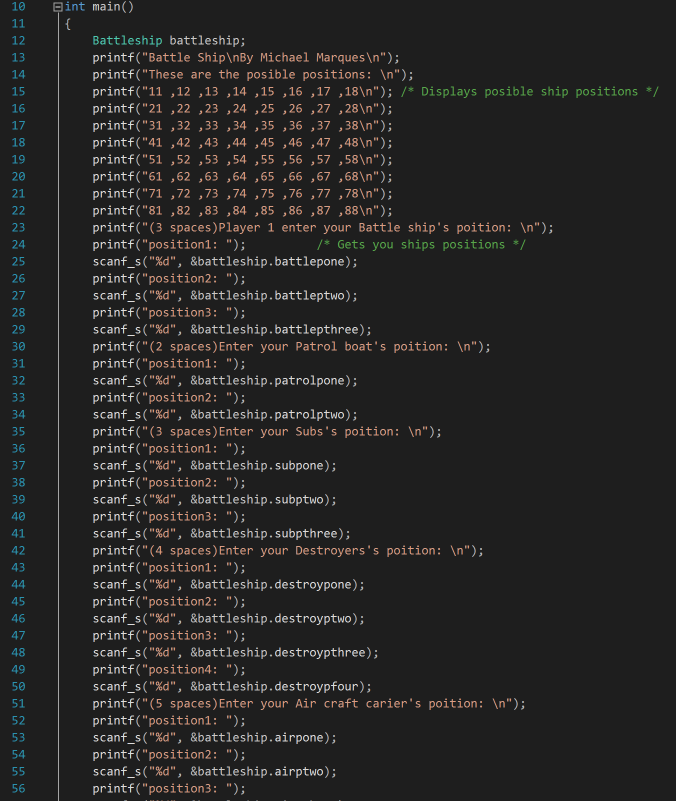


Figure 27: Original Main() Method

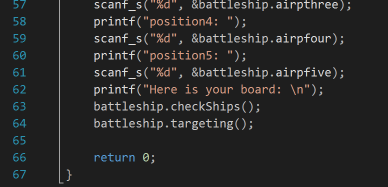


Figure 28: Original Main() Method

After all the refactoring done by creating methods, classes, renaming objects, and eliminating switch statements, we are able to write a clean, easy to read main method shown in Figure 29, below.

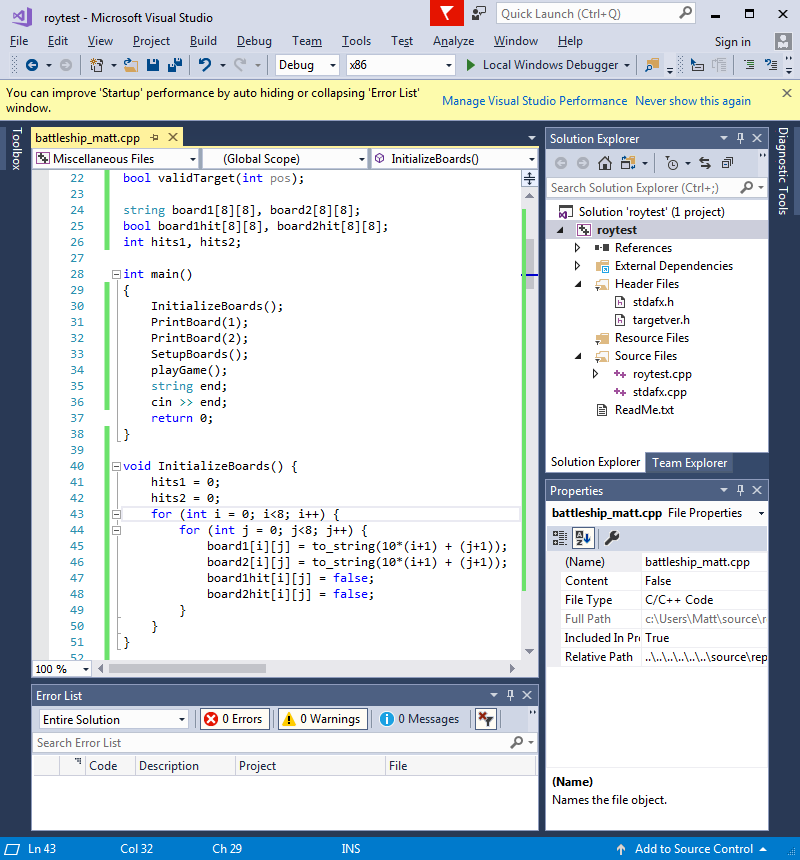


Figure 29: Main() Method

The Battleship game is now ready to be played. Figure 30, below, shows the InitializeBoard() and PrintBoard() methods being used.

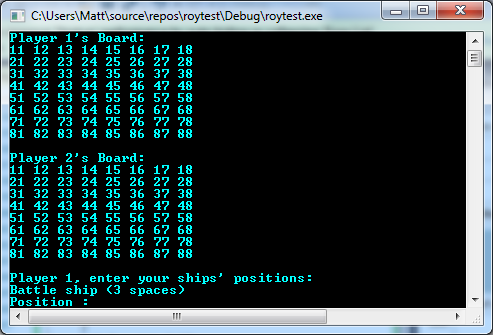


Figure 30: InitializeBoard() and PrintBoard() in action

After the boards have been printed, Player 1 must place their ships on the board which uses the setShip() and validateShip() methods, shown in Figure 31. Once all of Player 1’s ships have been placed, the board is printed again to showing each ship, Figure 32.

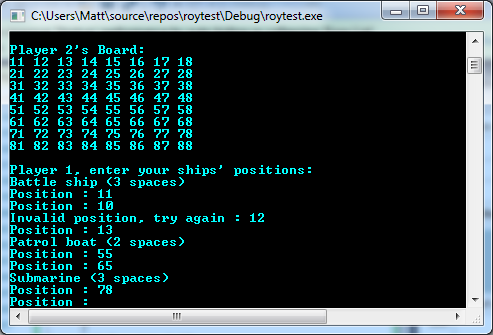


Figure 31: Set and Validate Ships for Player 1

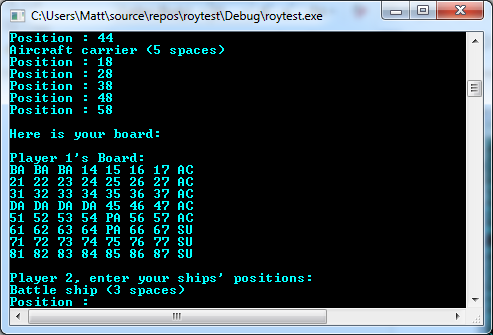


Figure 32: Player 1 Board with Ships Placed

The same process is repeated for Player 2 as shown in Figures 33 and 34, below.

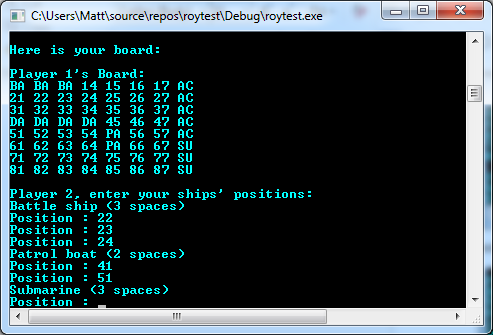


Figure 33: Set and Validate Ships for Player 2

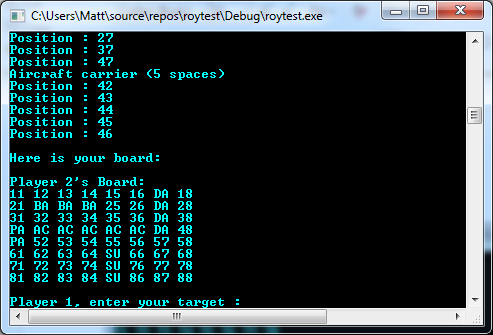


Figure 34: Player 2 Board with Ships Placed

After both players have set up their boards, the playGame() method is called which also calls the validateTraget(), fire(), countHits(), and printHits() methods. A sample of the game play can be seen in Figure 35, below.

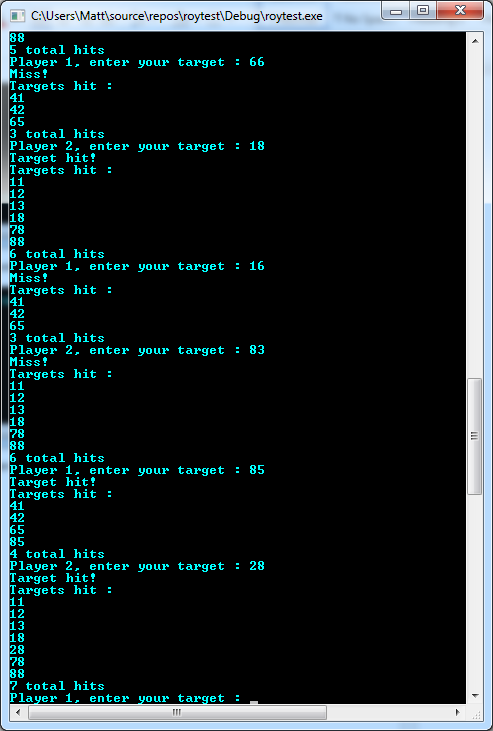


Figure 35: Sample Gameplay

As the players continue the game, they progress closer to sinking all the opponent’s ships. Once a player has achieved 17 hits, sunk all opponent’s ships, the condition in the playGame() method is executed indicating which player has won. A sample of a player winning the game is shown in Figure 36, below.

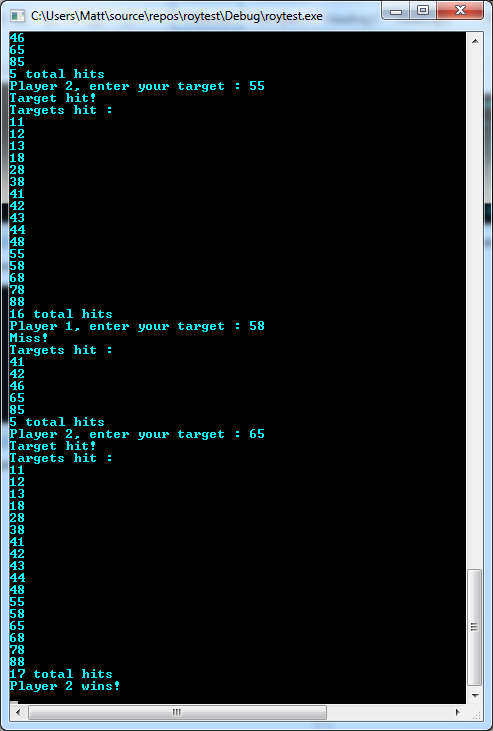


Figure 36: Sample of Winning

# References

(n.d.). Retrieved October 16, 2017, from <https://msdn.microsoft.com/en-us/library/jj620919.aspx>

How to find 2d array size in c. (n.d.). Retrieved October 16, 2017, from <https://stackoverflow.com/questions/10274162/how-to-find-2d-array-size-in-c>

How to solve the error LNK2019: unresolved external symbol - function? (n.d.). Retrieved October 16, 2017, from https://stackoverflow.com/questions/19886397/how-to-solve-the-error-lnk2019-unresolved-external-symbol-function

Multidimensional Arrays in C / C. (2017, May 30). Retrieved October 16, 2017, from <http://www.geeksforgeeks.org/multidimensional-arrays-c-cpp/>

Popa, A. (n.d.). C Unit Testing in Visual Studio. Retrieved October 16, 2017, from <https://blogs.msdn.microsoft.com/vcblog/2017/04/19/cpp-testing-in-visual-studi/>

Source code - Games. (n.d.). Retrieved October 16, 2017, from <https://www.cprogramming.com/cgi-bin/source/source.cgi?action=Category&CID=2>

What size is a Battleship game board? (n.d.). Retrieved October 16, 2017, from <https://answers.yahoo.com/question/index?qid=20110315131034AA08bPe>