10/16/2017

Refactoring of Battleship Game

Project 2 Report

Brady Brettin, Matthew Deremer, Luke Pace

Table of Contents

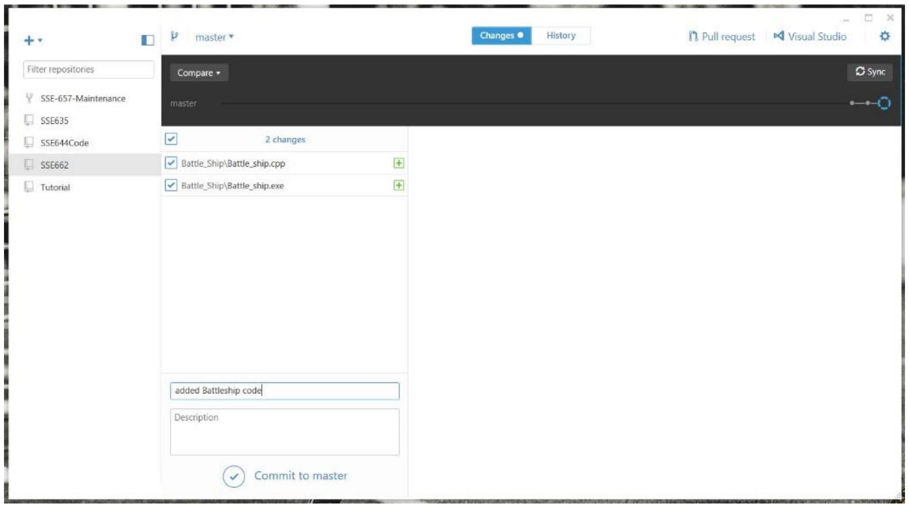
Table of Figures

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.



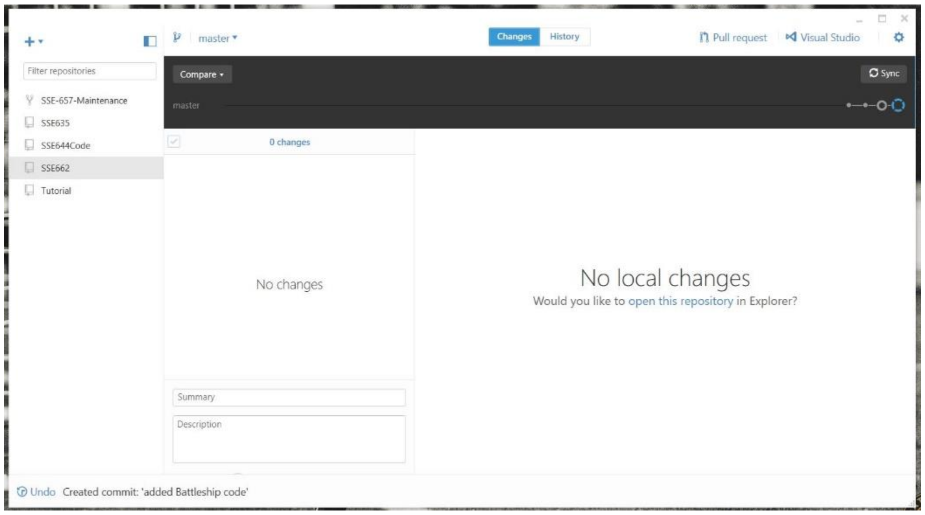
2 changes

Battle_Ship\Battle_ship.cpp

Battle_Ship\Battle_ship.exe

added Battleship code

Figure 1: GitHub



0 changes

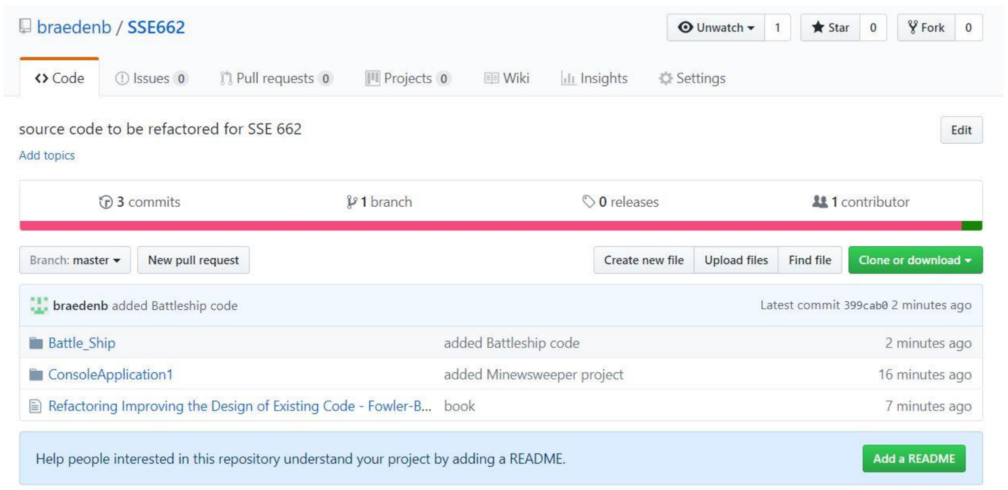
No changes

No local changes

Would you like to open this repository in Explorer?

Created commit: 'added Battleship code'

Figure 2: GitHub


braedenb / SSE662

Unwatch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

source code to be refactored for SSE 662

Add topics

3 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

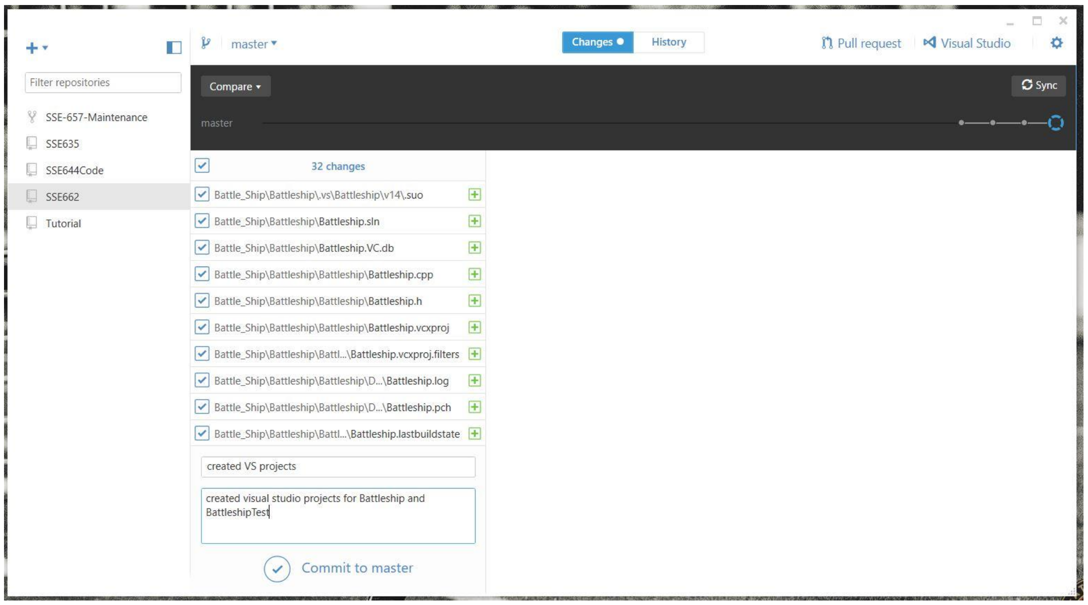
braedenb added Battleship code — Latest commit 399cab0 2 minutes ago

Battle_Ship	added Battleship code	2 minutes ago
ConsoleApplication1	added Minewsweeper project	16 minutes ago
Refactoring Improving the Design of Existing Code - Fowler-B...	book	7 minutes ago

Help people interested in this repository understand your project by adding a README. Add a README

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.


master | Changes | History | Pull request | Visual Studio

Filter repositories

- SSE-657-Maintenance
- SSE635
- SSE644Code
- SSE662
- Tutorial

Compare | Sync

master

32 changes

- Battle_Ship\Battleship\.vs\Battleship\v14\.suo
- Battle_Ship\Battleship\Battleship.sln
- Battle_Ship\Battleship\Battleship.VC.db
- Battle_Ship\Battleship\Battleship\Battleship.cpp
- Battle_Ship\Battleship\Battleship\Battleship.h
- Battle_Ship\Battleship\Battleship\Battleship.vcxproj
- Battle_Ship\Battleship\Battl...\Battleship.vcxproj.filters
- Battle_Ship\Battleship\Battleship\D...\Battleship.log
- Battle_Ship\Battleship\Battleship\D...\Battleship.pch
- Battle_Ship\Battleship\Battl...\Battleship.lastbuildstate

created VS projects

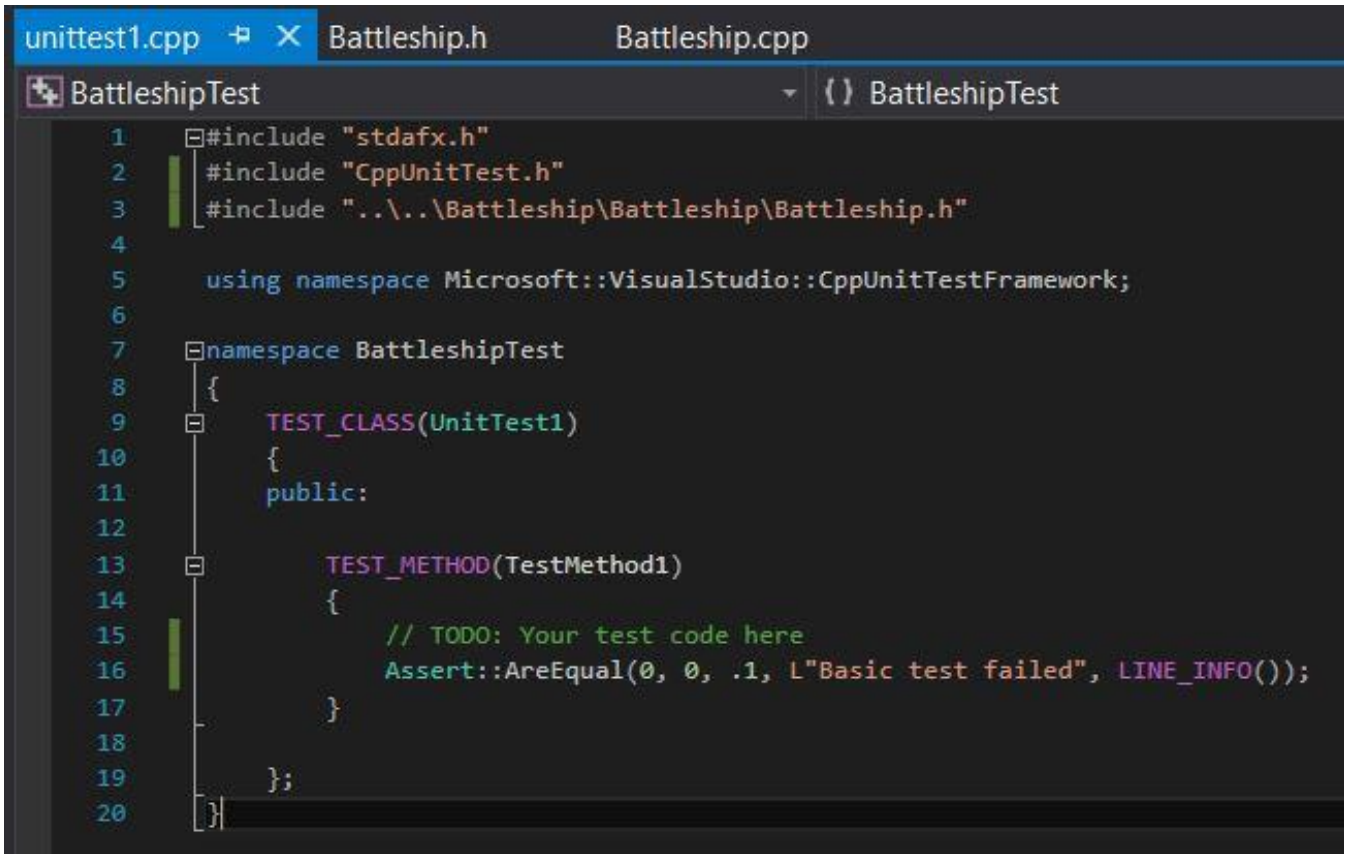
created visual studio projects for Battleship and BattleshipTest

Commit to master

Figure 4: Visual Studio Project

Construction of Test Suite

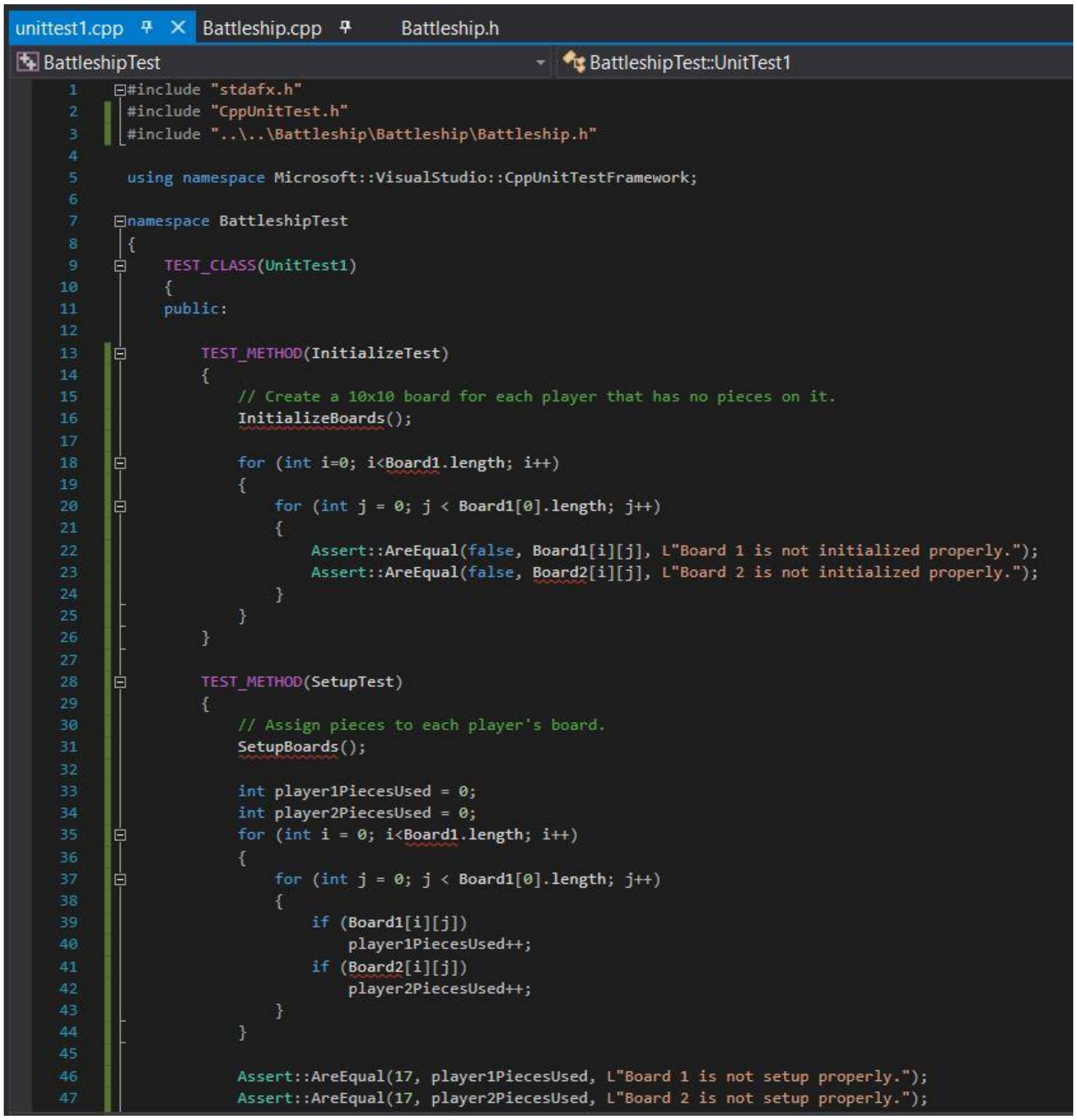
Before we could begin writing unit test, a test file had to be created and connected to the Battleship code as shown in Figure 5, below.



```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "..\..\Battleship\Battleship\Battleship.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace BattleshipTest
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestMethod1)
		{
			// TODO: Your test code here
			Assert::AreEqual(0, 0, .1, L"Basic test failed", LINE_INFO());
		}

	};
}
```

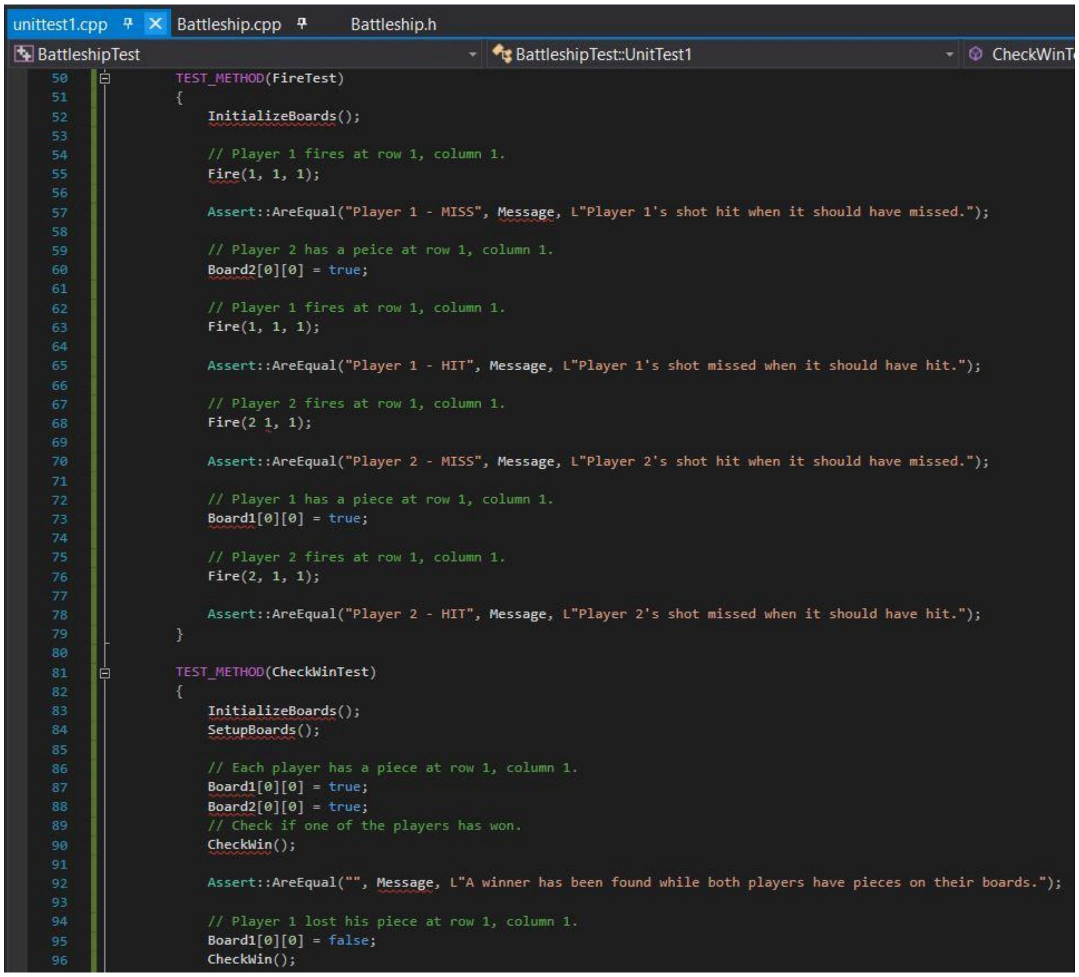
Figure 5: Battleship Test Setup

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.



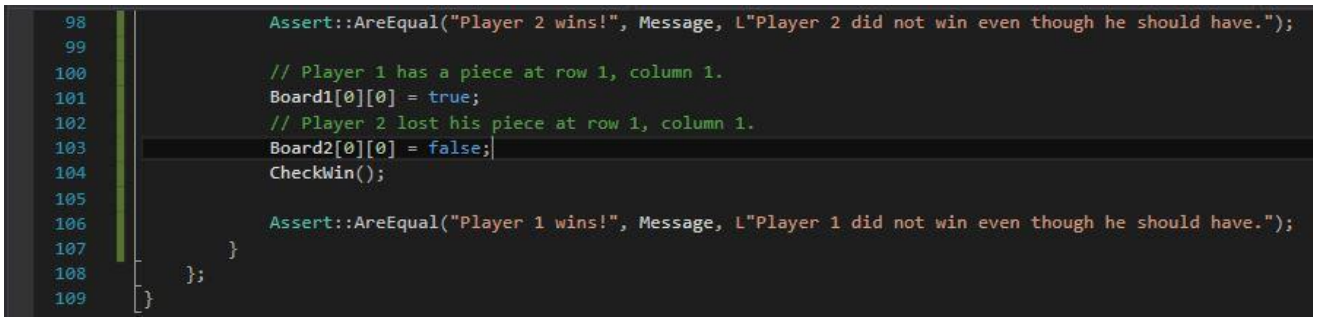
```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "..\..\Battleship\Battleship\Battleship.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace BattleshipTest
{
    TEST_CLASS(UnitTest1)
    {
    public:

        TEST_METHOD(InitializeTest)
        {
            // Create a 10x10 board for each player that has no pieces on it.
            InitializeBoards();

            for (int i=0; i<Board1.length; i++)
            {
                for (int j = 0; j < Board1[0].length; j++)
                {
                    Assert::AreEqual(false, Board1[i][j], L"Board 1 is not initialized properly.");
                    Assert::AreEqual(false, Board2[i][j], L"Board 2 is not initialized properly.");
                }
            }
        }

        TEST_METHOD(SetupTest)
        {
            // Assign pieces to each player's board.
            SetupBoards();

            int player1PiecesUsed = 0;
            int player2PiecesUsed = 0;
            for (int i = 0; i<Board1.length; i++)
            {
                for (int j = 0; j < Board1[0].length; j++)
                {
                    if (Board1[i][j])
                        player1PiecesUsed++;
                    if (Board2[i][j])
                        player2PiecesUsed++;
                }
            }

            Assert::AreEqual(17, player1PiecesUsed, L"Board 1 is not setup properly.");
            Assert::AreEqual(17, player2PiecesUsed, L"Board 2 is not setup properly.");
```

Figure 6: Battleship Test



```cpp
        TEST_METHOD(FireTest)
        {
            InitializeBoards();

            // Player 1 fires at row 1, column 1.
            Fire(1, 1, 1);

            Assert::AreEqual("Player 1 - MISS", Message, L"Player 1's shot hit when it should have missed.");

            // Player 2 has a peice at row 1, column 1.
            Board2[0][0] = true;

            // Player 1 fires at row 1, column 1.
            Fire(1, 1, 1);

            Assert::AreEqual("Player 1 - HIT", Message, L"Player 1's shot missed when it should have hit.");

            // Player 2 fires at row 1, column 1.
            Fire(2 1, 1);

            Assert::AreEqual("Player 2 - MISS", Message, L"Player 2's shot hit when it should have missed.");

            // Player 1 has a piece at row 1, column 1.
            Board1[0][0] = true;

            // Player 2 fires at row 1, column 1.
            Fire(2, 1, 1);

            Assert::AreEqual("Player 2 - HIT", Message, L"Player 2's shot missed when it should have hit.");
        }

        TEST_METHOD(CheckWinTest)
        {
            InitializeBoards();
            SetupBoards();

            // Each player has a piece at row 1, column 1.
            Board1[0][0] = true;
            Board2[0][0] = true;
            // Check if one of the players has won.
            CheckWin();

            Assert::AreEqual("", Message, L"A winner has been found while both players have pieces on their boards.");

            // Player 1 lost his piece at row 1, column 1.
            Board1[0][0] = false;
            CheckWin();
```

Figure 7: Battleship Test

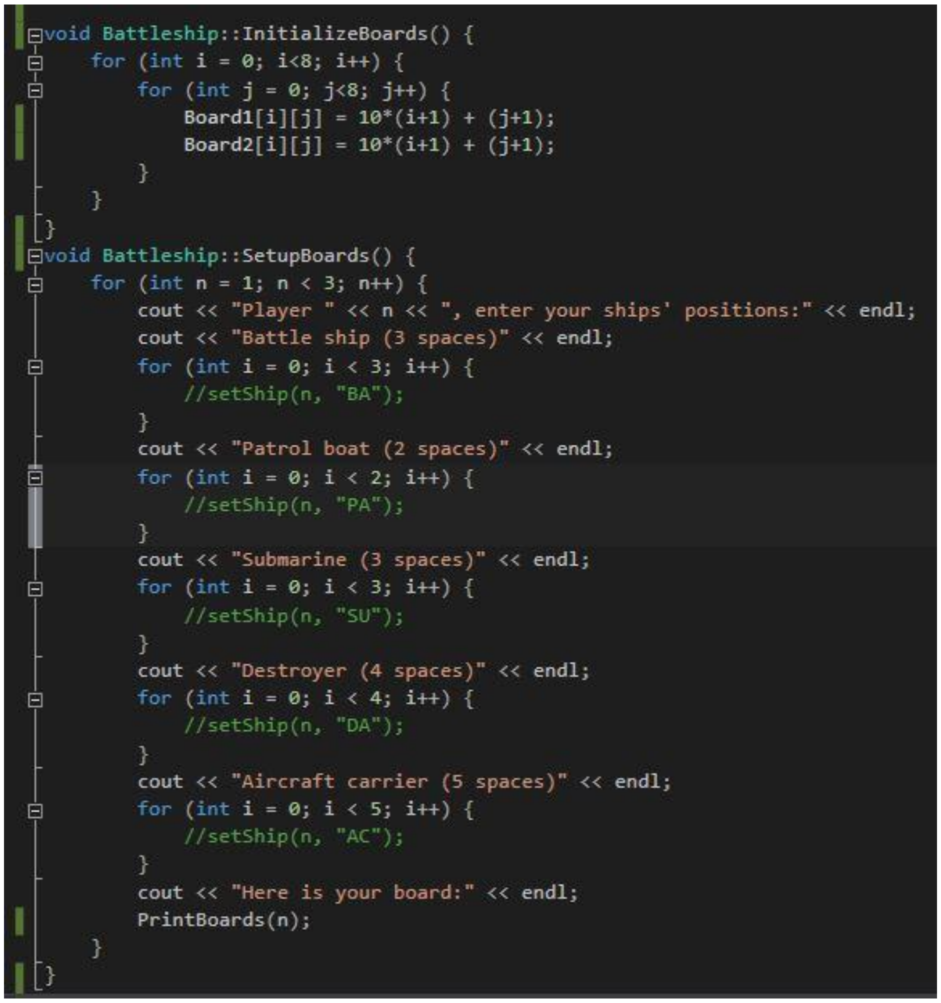


```cpp
            Assert::AreEqual("Player 2 wins!", Message, L"Player 2 did not win even though he should have.");

            // Player 1 has a piece at row 1, column 1.
            Board1[0][0] = true;
            // Player 2 lost his piece at row 1, column 1.
            Board2[0][0] = false;
            CheckWin();

            Assert::AreEqual("Player 1 wins!", Message, L"Player 1 did not win even though he should have.");
        }
    };
}
```

Figure 8: Battleship Test

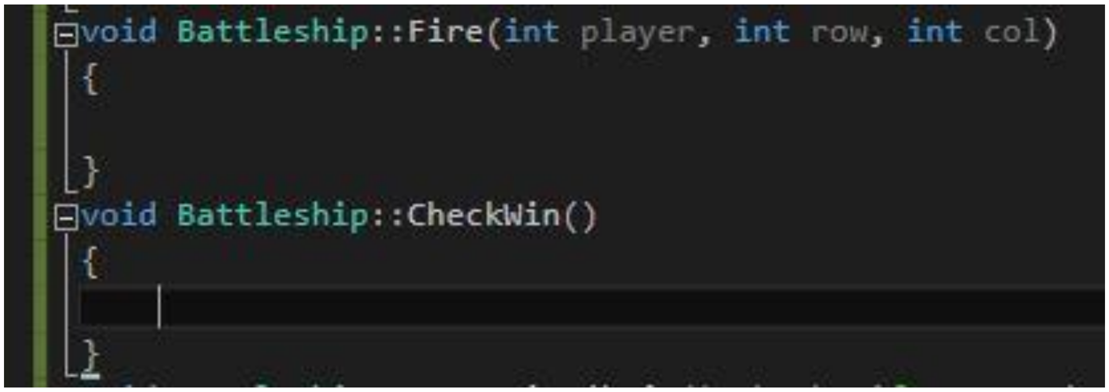
Initial Setup of Methods for Test Suite

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. This is an example of Extracting a Method. Extract Method is used when code is too complicated to read or an existing method is too long. The benefit of an Extract Method refactoring is increased readability and allows higher level methods to be more of a series of comments rather than complex expressions. 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. Also, with the creation of many of these methods, we demonstrated the refactoring techniques of renaming methods and adding/removing parameters. These are all examples making method calls simpler, as our text explains in Chapter 10. By renaming the methods and add/removing parameters to the original methods, we are able to make the code more readable and use objects rather than long list of parameters.



```cpp
void Battleship::InitializeBoards() {
    for (int i = 0; i<8; i++) {
        for (int j = 0; j<8; j++) {
            Board1[i][j] = 10*(i+1) + (j+1);
            Board2[i][j] = 10*(i+1) + (j+1);
        }
    }
}
void Battleship::SetupBoards() {
    for (int n = 1; n < 3; n++) {
        cout << "Player " << n << ", enter your ships' positions:" << endl;
        cout << "Battle ship (3 spaces)" << endl;
        for (int i = 0; i < 3; i++) {
            //setShip(n, "BA");
        }
        cout << "Patrol boat (2 spaces)" << endl;
        for (int i = 0; i < 2; i++) {
            //setShip(n, "PA");
        }
        cout << "Submarine (3 spaces)" << endl;
        for (int i = 0; i < 3; i++) {
            //setShip(n, "SU");
        }
        cout << "Destroyer (4 spaces)" << endl;
        for (int i = 0; i < 4; i++) {
            //setShip(n, "DA");
        }
        cout << "Aircraft carrier (5 spaces)" << endl;
        for (int i = 0; i < 5; i++) {
            //setShip(n, "AC");
        }
        cout << "Here is your board:" << endl;
        PrintBoards(n);
    }
}
```

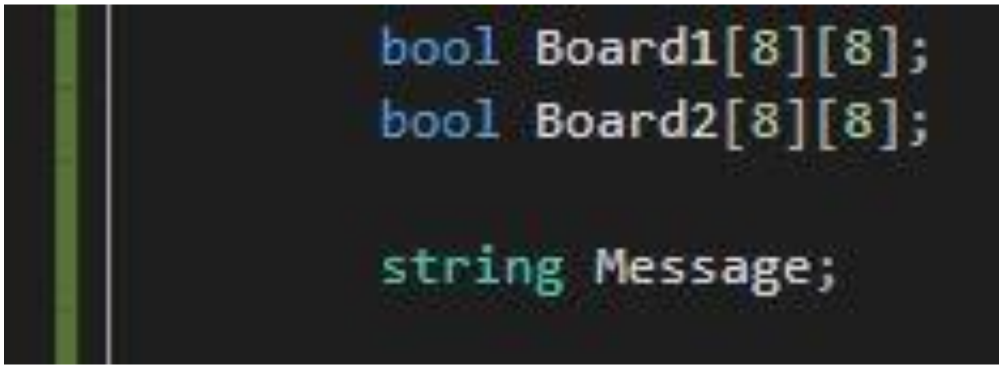
Figure 9: InitializeBoards() and SetupBoards()



```cpp
void Battleship::Fire(int player, int row, int col)
{


}
void Battleship::CheckWin()
{

}
```

Figure 10: Fire() and CheckWin() Methods

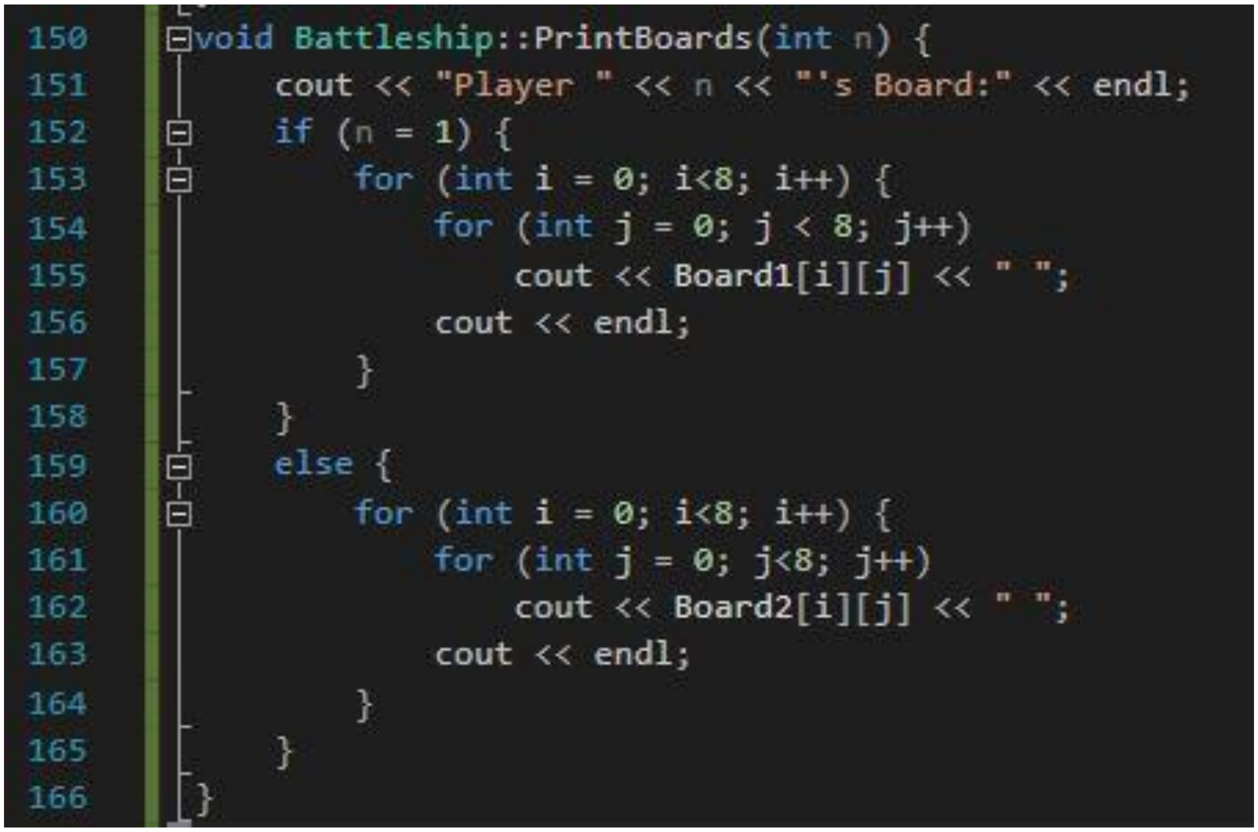
Next, we refactored the code to use a multi-dimensional 8x8 array of Booleans, rather than the array of integers currently being used. This refactoring improves the performance and execution time of the code and better models the real-world example of a Battleship game. A Message string, which would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



```
bool Board1[8][8];
bool Board2[8][8];

string Message;
```

Figure 11: Boards and Message

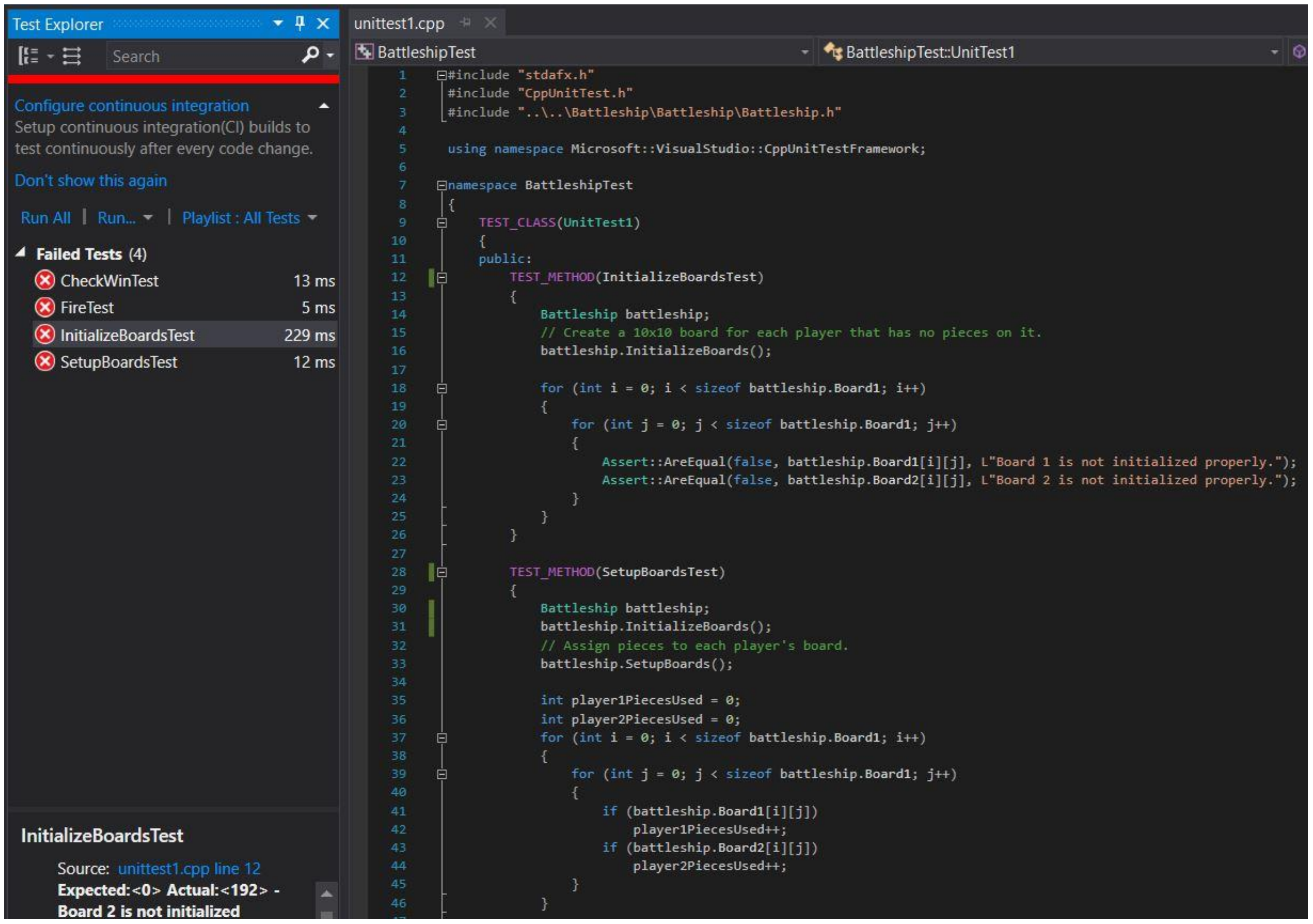
In the process of creating our tests, we also decided that it would be beneficial to have a method that would print the contents of the boards. Such a method would allow the user to easily view the contents of either board at the request of the user. As such, we created a PrintBoards() method shown in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



```
void Battleship::PrintBoards(int n) {
    cout << "Player " << n << "'s Board:" << endl;
    if (n = 1) {
        for (int i = 0; i<8; i++) {
            for (int j = 0; j < 8; j++)
                cout << Board1[i][j] << " ";
            cout << endl;
        }
    }
    else {
        for (int i = 0; i<8; i++) {
            for (int j = 0; j<8; j++)
                cout << Board2[i][j] << " ";
            cout << endl;
        }
    }
}
```

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.



Test Explorer

Search

Configure continuous integration

Setup continuous integration(CI) builds to test continuously after every code change.

Don't show this again

Run All | Run... | Playlist : All Tests

```
Failed Tests (4)
  CheckWinTest            13 ms
  FireTest                 5 ms
  InitializeBoardsTest   229 ms
  SetupBoardsTest         12 ms
```

InitializeBoardsTest

Source: unittest1.cpp line 12

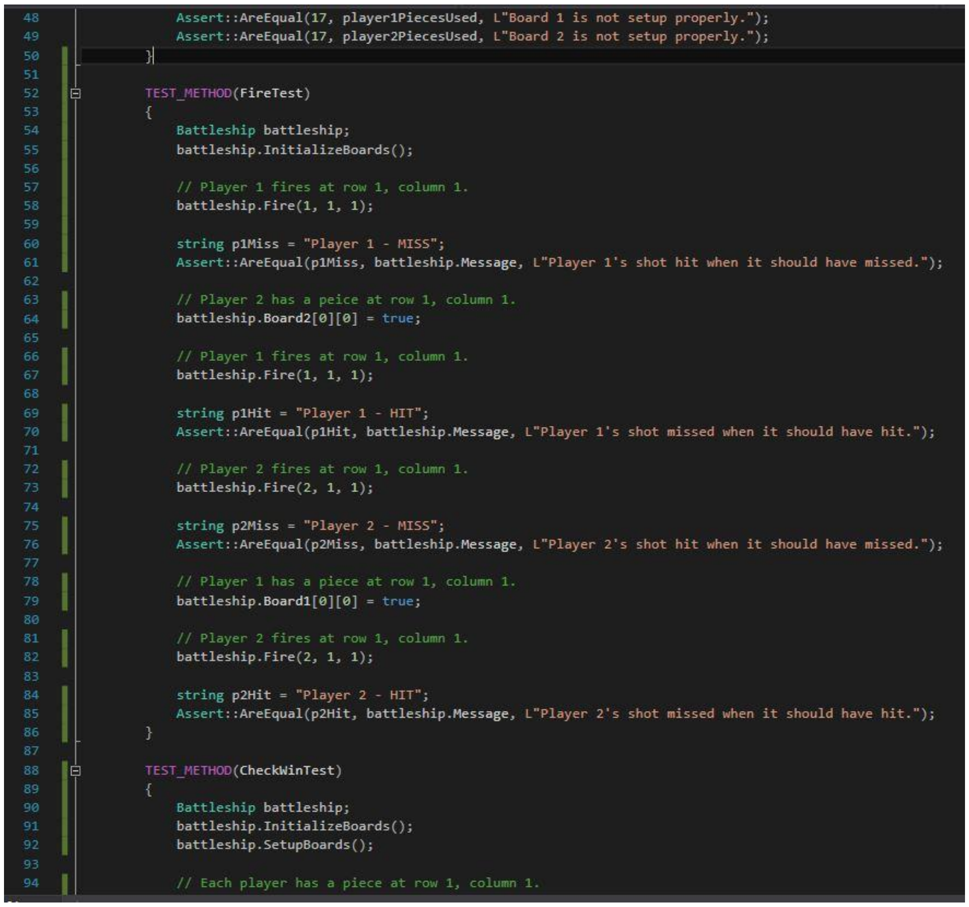
Expected:<0> Actual:<192> - Board 2 is not initialized

unittest1.cpp

BattleshipTest — BattleshipTest::UnitTest1

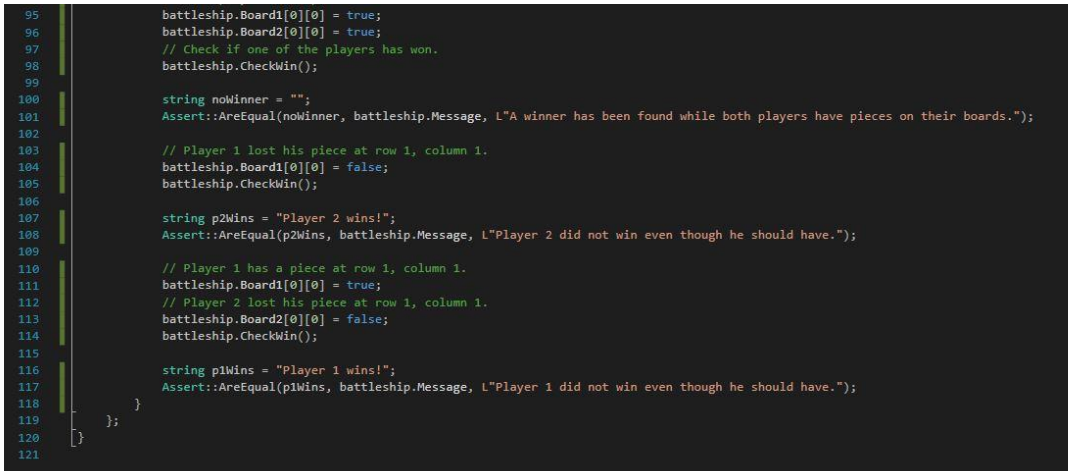
```cpp
#include "stdafx.h"
#include "CppUnitTest.h"
#include "..\..\Battleship\Battleship\Battleship.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace BattleshipTest
{
    TEST_CLASS(UnitTest1)
    {
    public:
        TEST_METHOD(InitializeBoardsTest)
        {
            Battleship battleship;
            // Create a 10x10 board for each player that has no pieces on it.
            battleship.InitializeBoards();

            for (int i = 0; i < sizeof battleship.Board1; i++)
            {
                for (int j = 0; j < sizeof battleship.Board1; j++)
                {
                    Assert::AreEqual(false, battleship.Board1[i][j], L"Board 1 is not initialized properly.");
                    Assert::AreEqual(false, battleship.Board2[i][j], L"Board 2 is not initialized properly.");
                }
            }
        }

        TEST_METHOD(SetupBoardsTest)
        {
            Battleship battleship;
            battleship.InitializeBoards();
            // Assign pieces to each player's board.
            battleship.SetupBoards();

            int player1PiecesUsed = 0;
            int player2PiecesUsed = 0;
            for (int i = 0; i < sizeof battleship.Board1; i++)
            {
                for (int j = 0; j < sizeof battleship.Board1; j++)
                {
                    if (battleship.Board1[i][j])
                        player1PiecesUsed++;
                    if (battleship.Board2[i][j])
                        player2PiecesUsed++;
                }
            }
```

Figure 13: Failed Test



```
            Assert::AreEqual(17, player1PiecesUsed, L"Board 1 is not setup properly.");
            Assert::AreEqual(17, player2PiecesUsed, L"Board 2 is not setup properly.");
        }

        TEST_METHOD(FireTest)
        {
            Battleship battleship;
            battleship.InitializeBoards();

            // Player 1 fires at row 1, column 1.
            battleship.Fire(1, 1, 1);

            string p1Miss = "Player 1 - MISS";
            Assert::AreEqual(p1Miss, battleship.Message, L"Player 1's shot hit when it should have missed.");

            // Player 2 has a peice at row 1, column 1.
            battleship.Board2[0][0] = true;

            // Player 1 fires at row 1, column 1.
            battleship.Fire(1, 1, 1);

            string p1Hit = "Player 1 - HIT";
            Assert::AreEqual(p1Hit, battleship.Message, L"Player 1's shot missed when it should have hit.");

            // Player 2 fires at row 1, column 1.
            battleship.Fire(2, 1, 1);

            string p2Miss = "Player 2 - MISS";
            Assert::AreEqual(p2Miss, battleship.Message, L"Player 2's shot hit when it should have missed.");

            // Player 1 has a piece at row 1, column 1.
            battleship.Board1[0][0] = true;

            // Player 2 fires at row 1, column 1.
            battleship.Fire(2, 1, 1);

            string p2Hit = "Player 2 - HIT";
            Assert::AreEqual(p2Hit, battleship.Message, L"Player 2's shot missed when it should have hit.");
        }

        TEST_METHOD(CheckWinTest)
        {
            Battleship battleship;
            battleship.InitializeBoards();
            battleship.SetupBoards();

            // Each player has a piece at row 1, column 1.
```

Figure 14: Failed Test

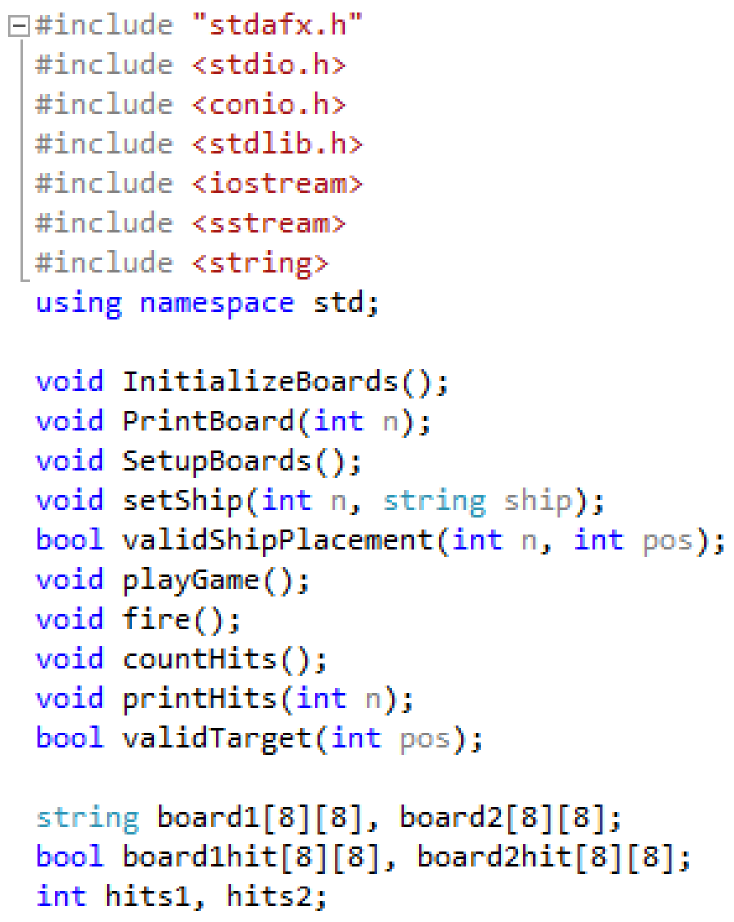


```
            battleship.Board1[0][0] = true;
            battleship.Board2[0][0] = true;
            // Check if one of the players has won.
            battleship.CheckWin();

            string noWinner = "";
            Assert::AreEqual(noWinner, battleship.Message, L"A winner has been found while both players have pieces on their boards.");

            // Player 1 lost his piece at row 1, column 1.
            battleship.Board1[0][0] = false;
            battleship.CheckWin();

            string p2Wins = "Player 2 wins!";
            Assert::AreEqual(p2Wins, battleship.Message, L"Player 2 did not win even though he should have.");

            // Player 1 has a piece at row 1, column 1.
            battleship.Board1[0][0] = true;
            // Player 2 lost his piece at row 1, column 1.
            battleship.Board2[0][0] = false;
            battleship.CheckWin();

            string p1Wins = "Player 1 wins!";
            Assert::AreEqual(p1Wins, battleship.Message, L"Player 1 did not win even though he should have.");
        }
    };
}
```

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.

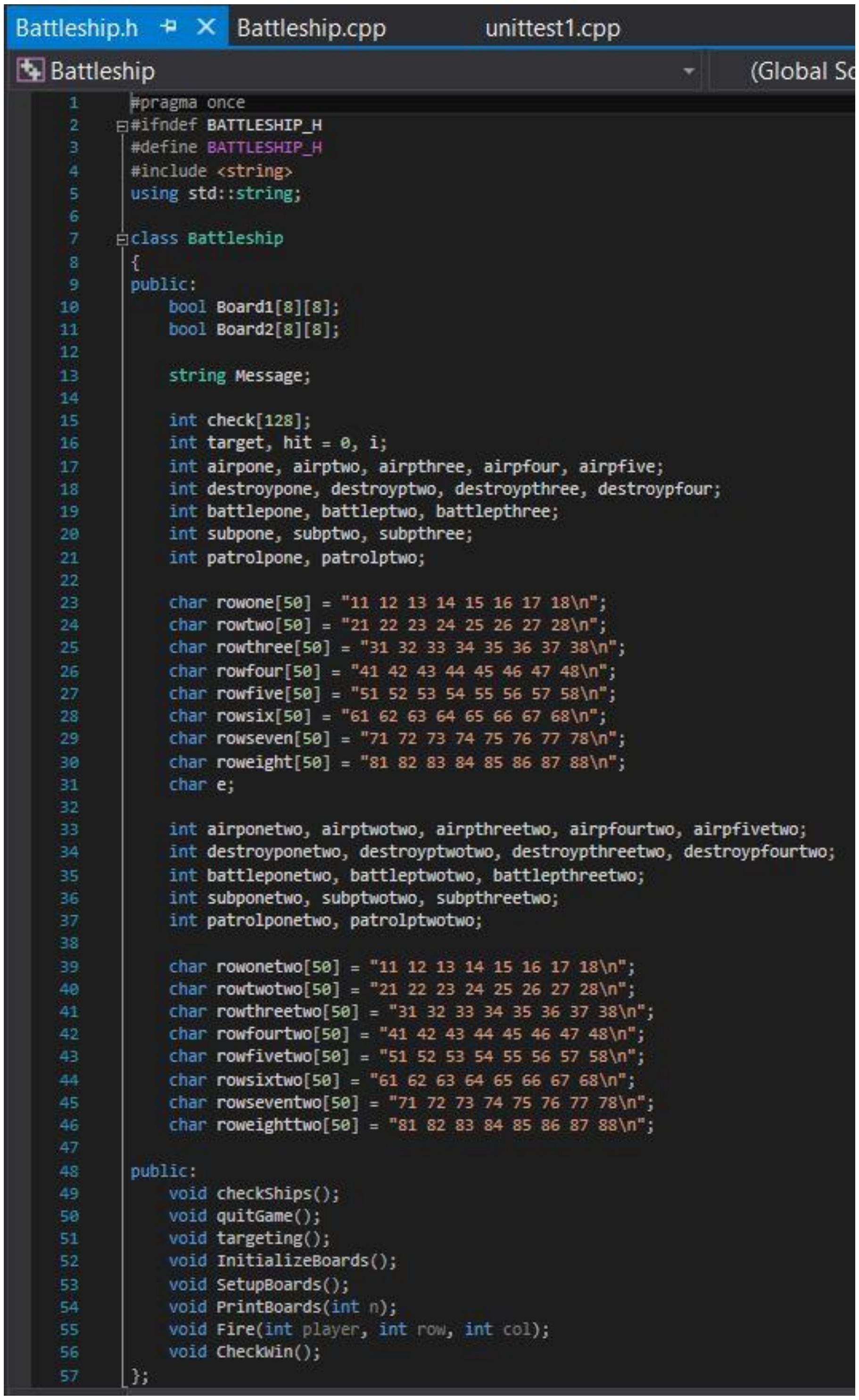


```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

void InitializeBoards();
void PrintBoard(int n);
void SetupBoards();
void setShip(int n, string ship);
bool validShipPlacement(int n, int pos);
void playGame();
void fire();
void countHits();
void printHits(int n);
bool validTarget(int pos);

string board1[8][8], board2[8][8];
bool board1hit[8][8], board2hit[8][8];
int hits1, hits2;
```

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. This refactoring is an example of Extract Class. The original code had many lines at the beginning of the project making it hard to see where the main method started. By implementing the Extract Class refactoring, we were able to define a clear class with well defined methods that is easy to understand.

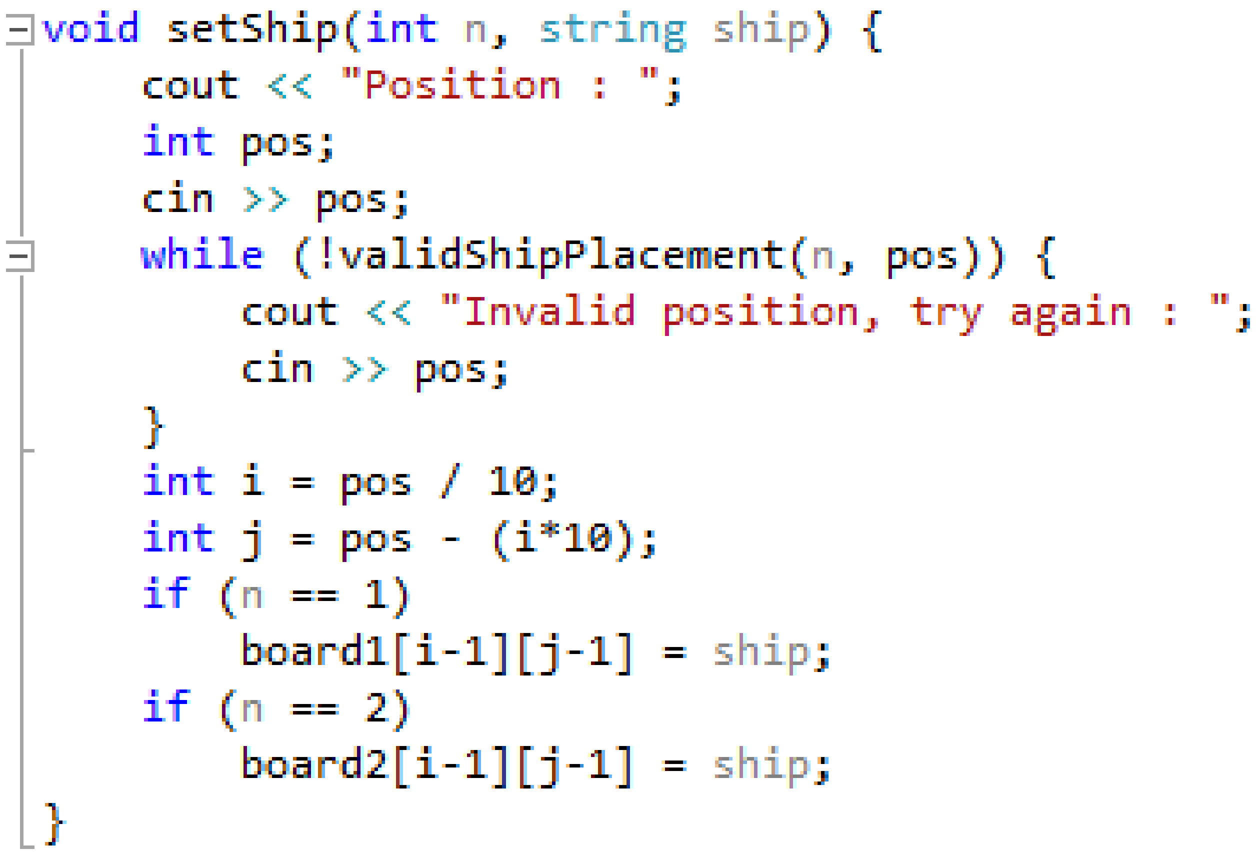


```cpp
#pragma once
#ifndef BATTLESHIP_H
#define BATTLESHIP_H
#include <string>
using std::string;

class Battleship
{
public:
    bool Board1[8][8];
    bool Board2[8][8];

    string Message;

    int check[128];
    int target, hit = 0, i;
    int airpone, airptwo, airpthree, airpfour, airpfive;
    int destroypone, destroyptwo, destroypthree, destroypfour;
    int battlepone, battleptwo, battlepthree;
    int subpone, subptwo, subpthree;
    int patrolpone, patrolptwo;

    char rowone[50] = "11 12 13 14 15 16 17 18\n";
    char rowtwo[50] = "21 22 23 24 25 26 27 28\n";
    char rowthree[50] = "31 32 33 34 35 36 37 38\n";
    char rowfour[50] = "41 42 43 44 45 46 47 48\n";
    char rowfive[50] = "51 52 53 54 55 56 57 58\n";
    char rowsix[50] = "61 62 63 64 65 66 67 68\n";
    char rowseven[50] = "71 72 73 74 75 76 77 78\n";
    char roweight[50] = "81 82 83 84 85 86 87 88\n";
    char e;

    int airponetwo, airptwotwo, airpthreetwo, airpfourtwo, airpfivetwo;
    int destroyponetwo, destroyptwotwo, destroypthreetwo, destroypfourtwo;
    int battleponetwo, battleptwotwo, battlepthreetwo;
    int subponetwo, subptwotwo, subpthreetwo;
    int patrolponetwo, patrolptwotwo;

    char rowonetwo[50] = "11 12 13 14 15 16 17 18\n";
    char rowtwotwo[50] = "21 22 23 24 25 26 27 28\n";
    char rowthreetwo[50] = "31 32 33 34 35 36 37 38\n";
    char rowfourtwo[50] = "41 42 43 44 45 46 47 48\n";
    char rowfivetwo[50] = "51 52 53 54 55 56 57 58\n";
    char rowsixtwo[50] = "61 62 63 64 65 66 67 68\n";
    char rowseventwo[50] = "71 72 73 74 75 76 77 78\n";
    char roweighttwo[50] = "81 82 83 84 85 86 87 88\n";

public:
    void checkShips();
    void quitGame();
    void targeting();
    void InitializeBoards();
    void SetupBoards();
    void PrintBoards(int n);
    void Fire(int player, int row, int col);
    void CheckWin();
};
```

Figure 17: Battleship Class

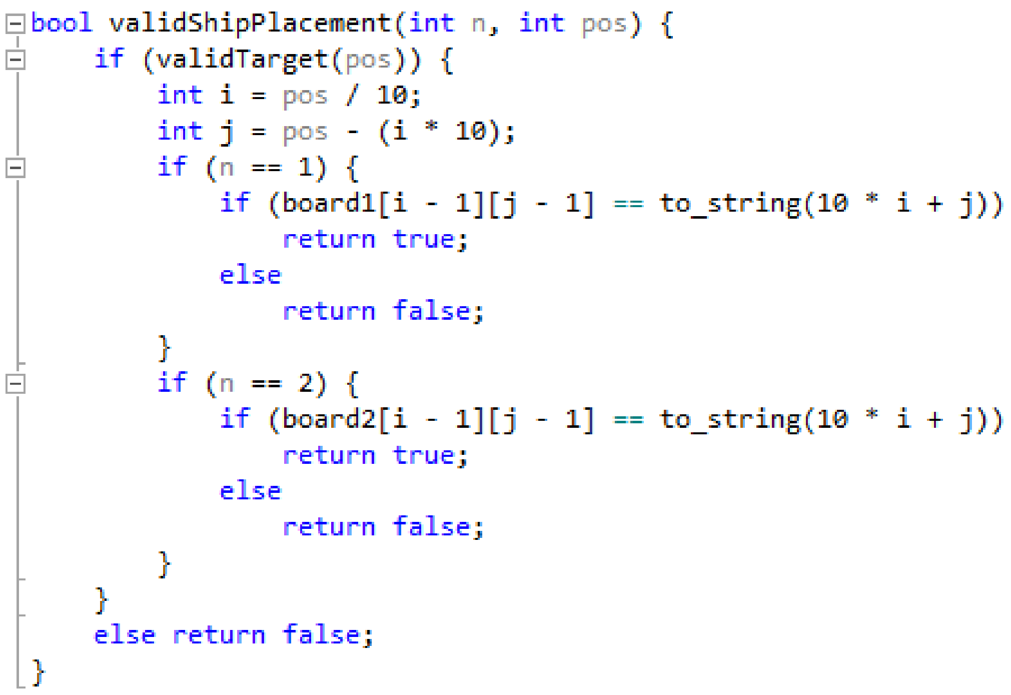
Placing Ships and Targeting

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.



```
void setShip(int n, string ship) {
    cout << "Position : ";
    int pos;
    cin >> pos;
    while (!validShipPlacement(n, pos)) {
        cout << "Invalid position, try again : ";
        cin >> pos;
    }
    int i = pos / 10;
    int j = pos - (i*10);
    if (n == 1)
        board1[i-1][j-1] = ship;
    if (n == 2)
        board2[i-1][j-1] = ship;
}
```

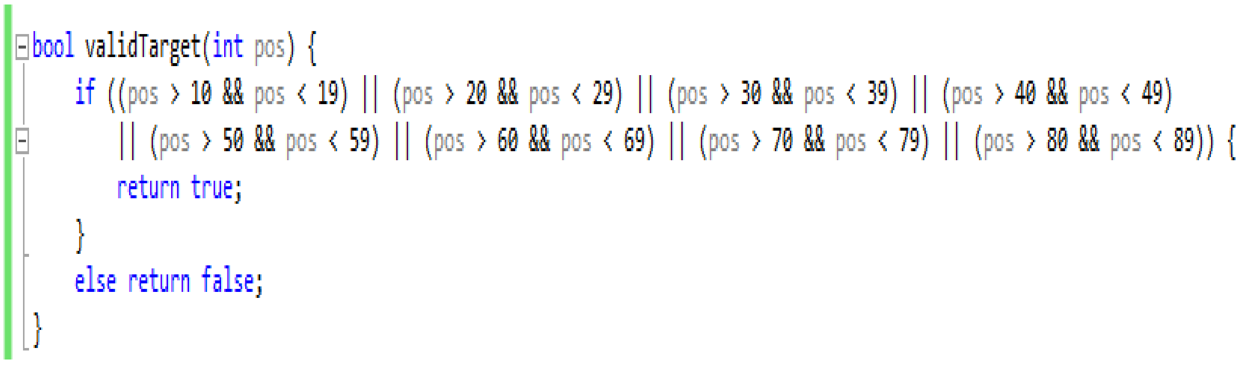
Figure 18: setShip() Method



```
bool validShipPlacement(int n, int pos) {
    if (validTarget(pos)) {
        int i = pos / 10;
        int j = pos - (i * 10);
        if (n == 1) {
            if (board1[i - 1][j - 1] == to_string(10 * i + j))
                return true;
            else
                return false;
        }
        if (n == 2) {
            if (board2[i - 1][j - 1] == to_string(10 * i + j))
                return true;
            else
                return false;
        }
    }
    else return false;
}
```

Figure 19: validShipPlacement() Method

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

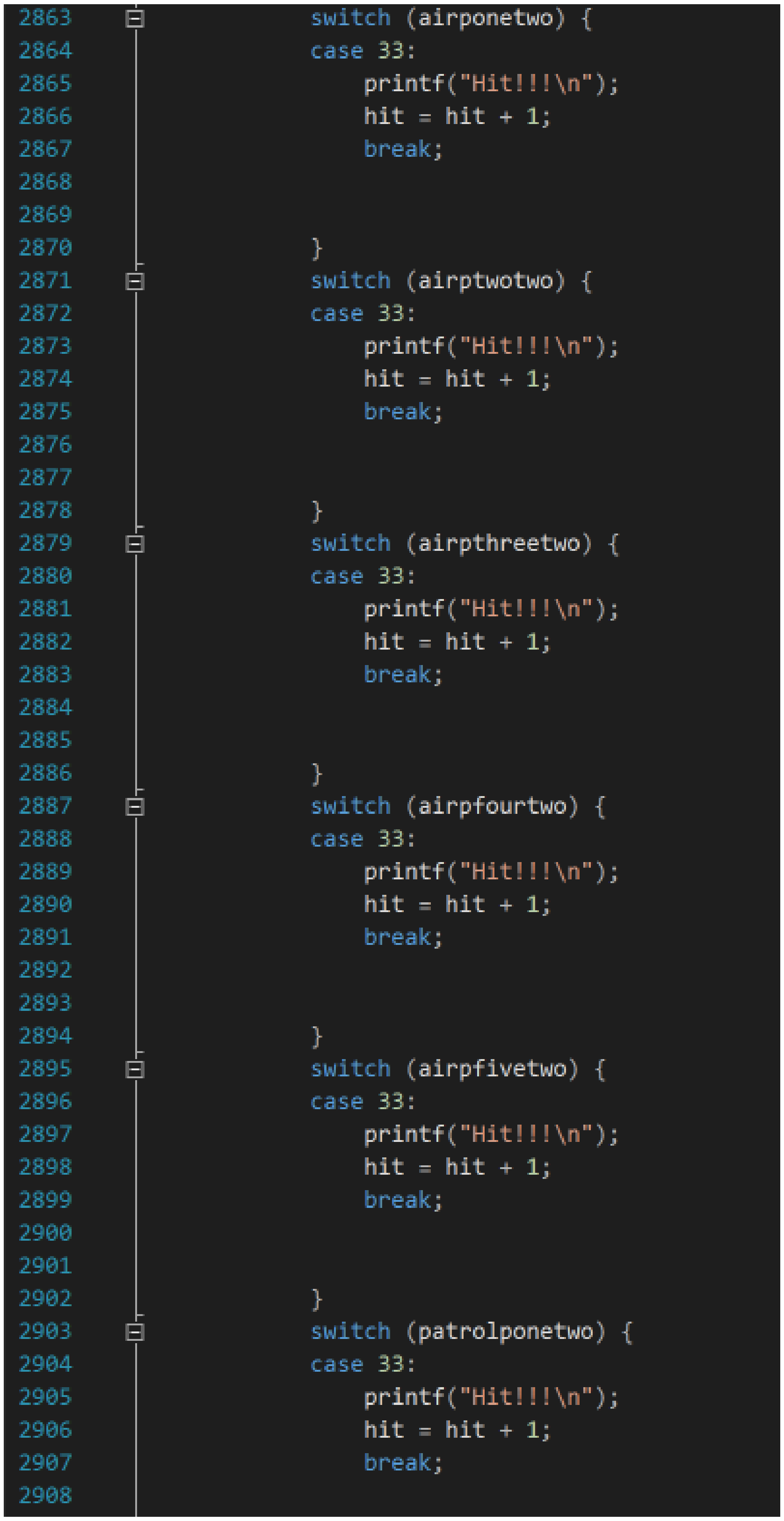


```
bool validTarget(int pos) {
    if ((pos > 10 && pos < 19) || (pos > 20 && pos < 29) || (pos > 30 && pos < 39) || (pos > 40 && pos < 49)
        || (pos > 50 && pos < 59) || (pos > 60 && pos < 69) || (pos > 70 && pos < 79) || (pos > 80 && pos < 89)) {
        return true;
    }
    else return false;
}
```

Figure 20: validTarget() Method

Fire at Opponent Ship

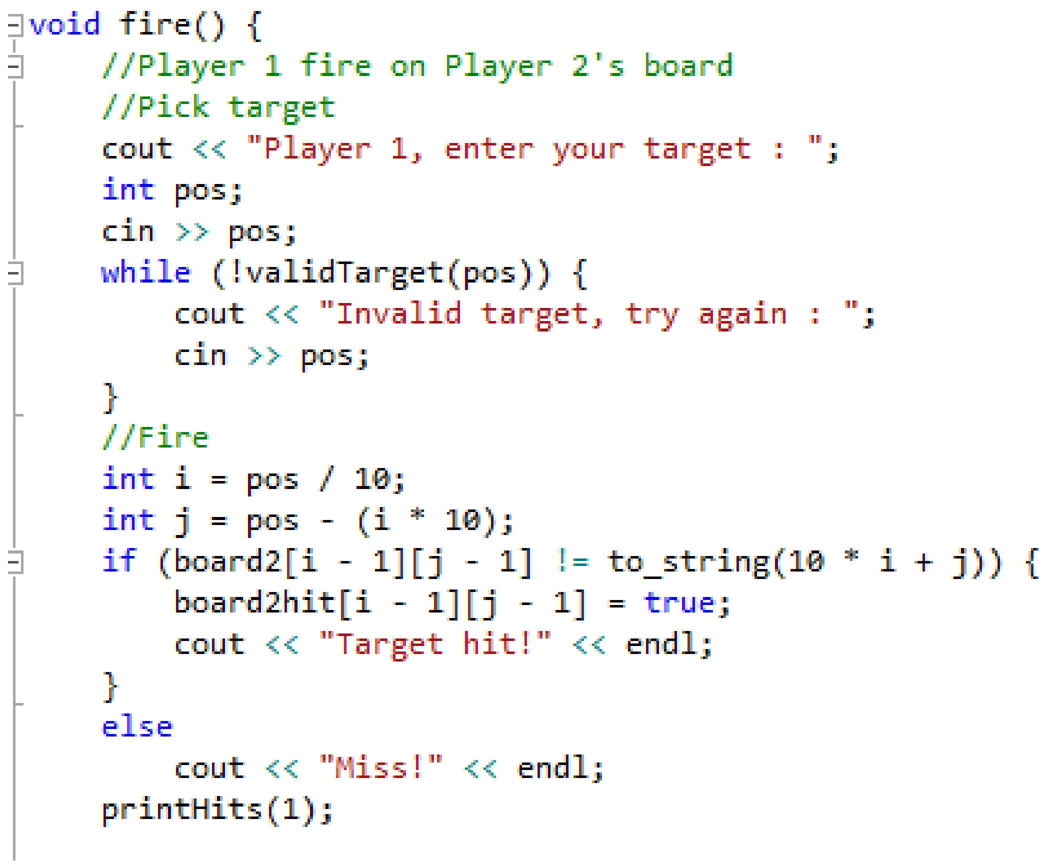
We then moved on to the largest refactoring done for this project. At this point, the boards for each player had been created and printed, ships could be placed on valid positions, and ships could be targeted. Now we needed to create a method to allow a player to fire at a position on the opponent's board. The original code, a sample is shown in Figure 21 below, was thousands of lines of switch statements and duplicated code. In our text, the author refers to the techniques used in Figure 21 as "Bad Smells." A "Bad Smell" is an indication that code needs to be refactored. The "Bad Smells" found in this project were: duplicated code, long methods, large class, and switch statements. This is just a small sample of the many "Bad Smells" in the original code.



```
        switch (airponetwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;


        }
        switch (airptwotwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;


        }
        switch (airpthreetwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;


        }
        switch (airpfourtwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;


        }
        switch (airpfivetwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;


        }
        switch (patrolponetwo) {
        case 33:
            printf("Hit!!!\n");
            hit = hit + 1;
            break;

```

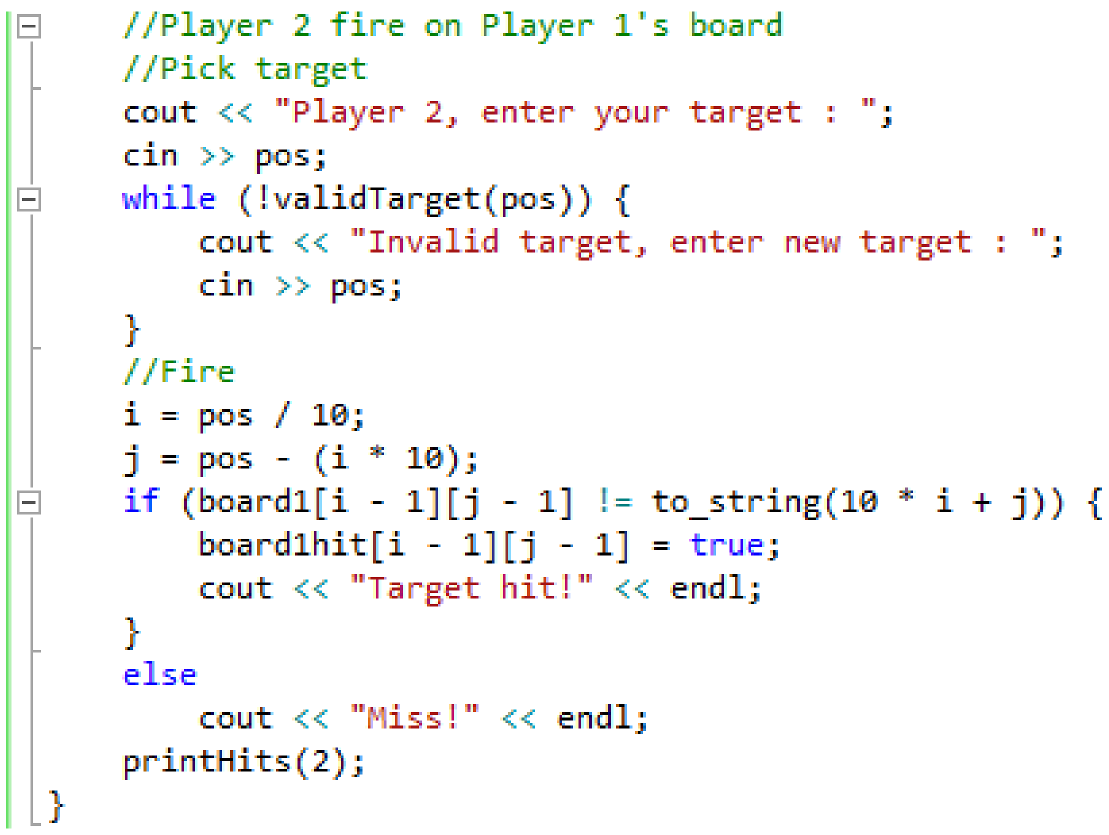
Figure 21: Original Fire Code

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. This method demonstrates both the Extract Method and Substitute Algorithm refactoring techniques (see *Initial Setup of Methods for Test Suite* section for explanation of Extract Method). Substitute Algorithm is done when the existing algorithm works but is difficult to understand. As Figure 21 showed, the original algorithm for firing at an opponent's ship worked but was hard to read due to its length and repetition. By replacing the code with the new Fire() method, it is much easier to analyze the methods functionality.



```
void fire() {
    //Player 1 fire on Player 2's board
    //Pick target
    cout << "Player 1, enter your target : ";
    int pos;
    cin >> pos;
    while (!validTarget(pos)) {
        cout << "Invalid target, try again : ";
        cin >> pos;
    }
    //Fire
    int i = pos / 10;
    int j = pos - (i * 10);
    if (board2[i - 1][j - 1] != to_string(10 * i + j)) {
        board2hit[i - 1][j - 1] = true;
        cout << "Target hit!" << endl;
    }
    else
        cout << "Miss!" << endl;
    printHits(1);
```

Figure 22: Fire() Method

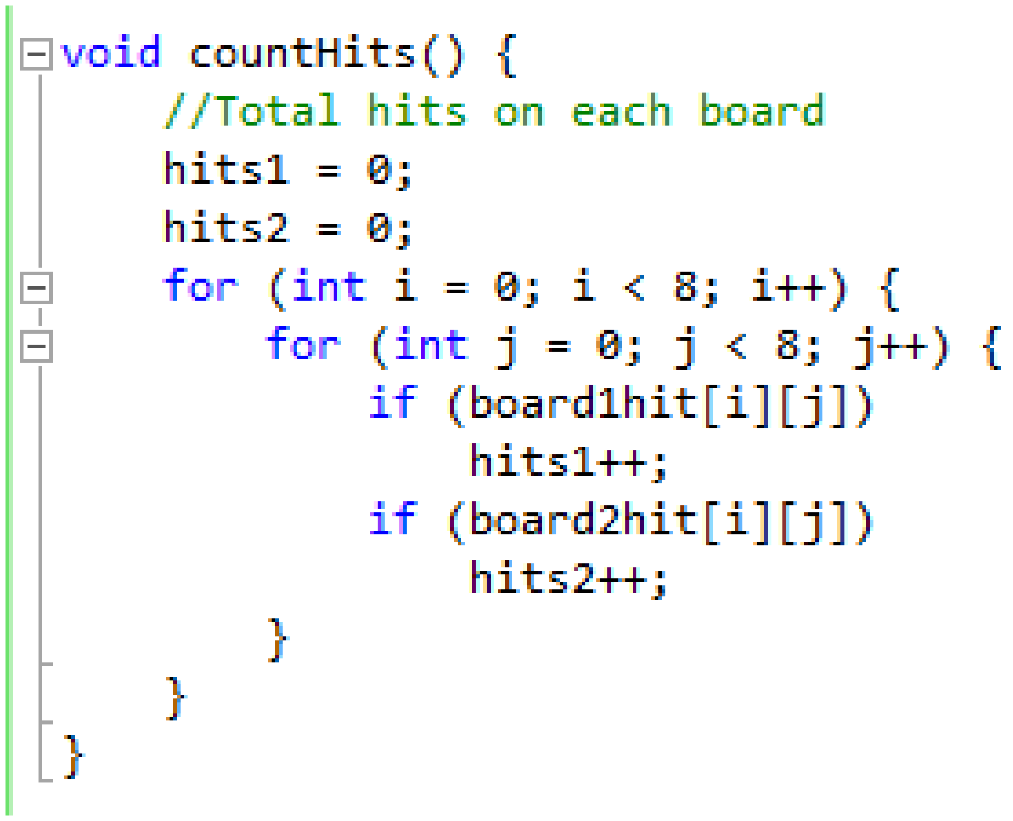


```
        //Player 2 fire on Player 1's board
        //Pick target
        cout << "Player 2, enter your target : ";
        cin >> pos;
        while (!validTarget(pos)) {
            cout << "Invalid target, enter new target : ";
            cin >> pos;
        }
        //Fire
        i = pos / 10;
        j = pos - (i * 10);
        if (board1[i - 1][j - 1] != to_string(10 * i + j)) {
            board1hit[i - 1][j - 1] = true;
            cout << "Target hit!" << endl;
        }
        else
            cout << "Miss!" << endl;
        printHits(2);
    }
```

Figure 23: Fire Method()

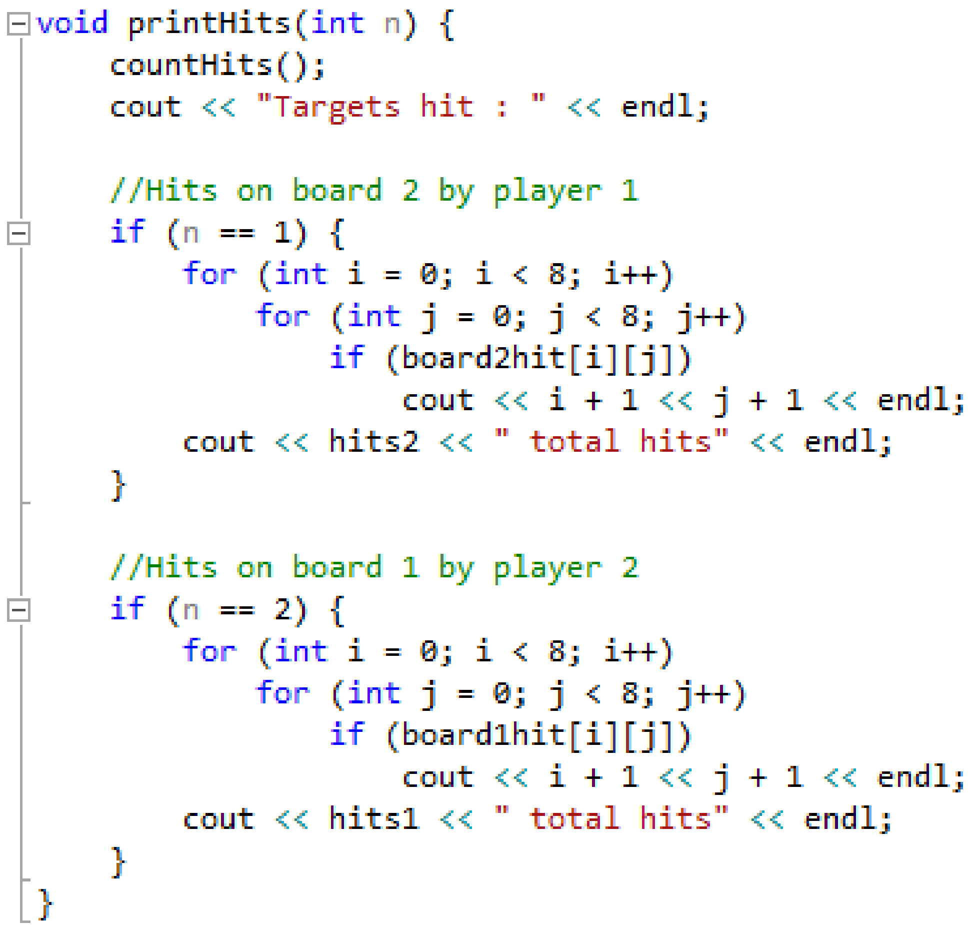
Count and Print Hits

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.



```
void countHits() {
    //Total hits on each board
    hits1 = 0;
    hits2 = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (board1hit[i][j])
                hits1++;
            if (board2hit[i][j])
                hits2++;
        }
    }
}
```

Figure 24: countHits() Method

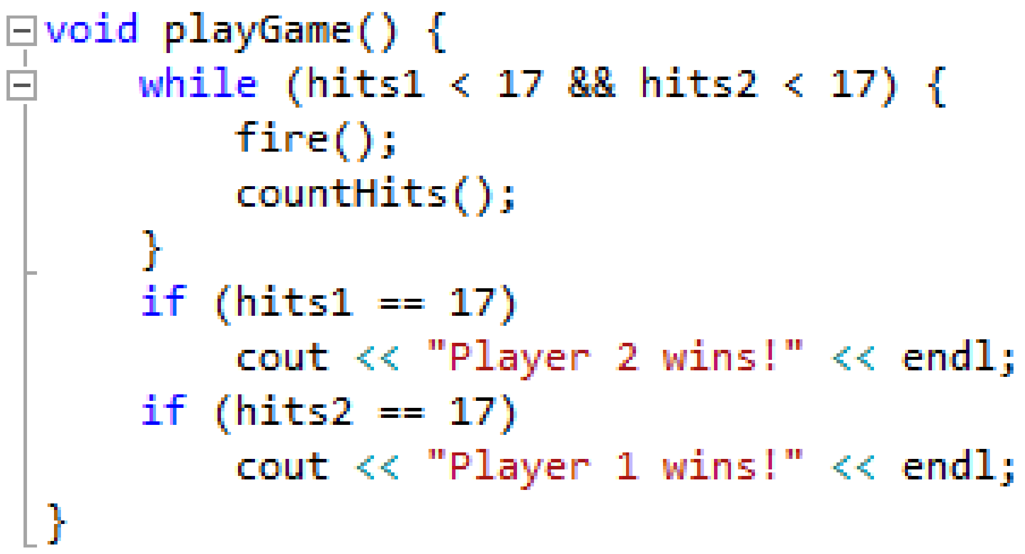


```cpp
void printHits(int n) {
    countHits();
    cout << "Targets hit : " << endl;

    //Hits on board 2 by player 1
    if (n == 1) {
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < 8; j++)
                if (board2hit[i][j])
                    cout << i + 1 << j + 1 << endl;
        cout << hits2 << " total hits" << endl;
    }

    //Hits on board 1 by player 2
    if (n == 2) {
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < 8; j++)
                if (board1hit[i][j])
                    cout << i + 1 << j + 1 << endl;
        cout << hits1 << " total hits" << endl;
    }
}
```

Figure 25: printHits() Method

Playing the Game

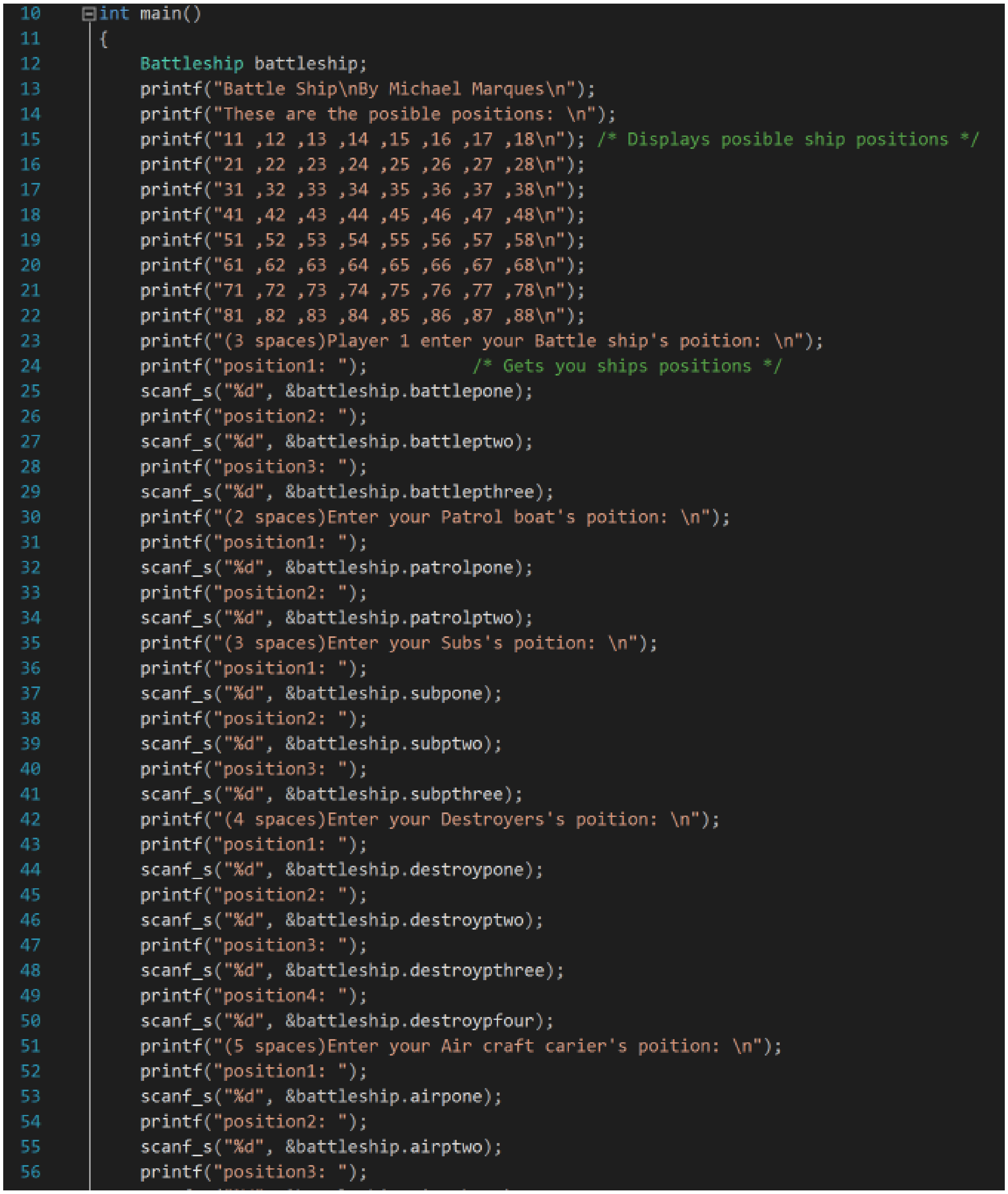
Now that all the functionality of the game had been refactored into short, easy to read methods, we needed to construct a method, playGame(), to utilize the other methods. The playGame() method can be seen in Figure 26, below.



```
void playGame() {
    while (hits1 < 17 && hits2 < 17) {
        fire();
        countHits();
    }
    if (hits1 == 17)
        cout << "Player 2 wins!" << endl;
    if (hits2 == 17)
        cout << "Player 1 wins!" << endl;
}
```

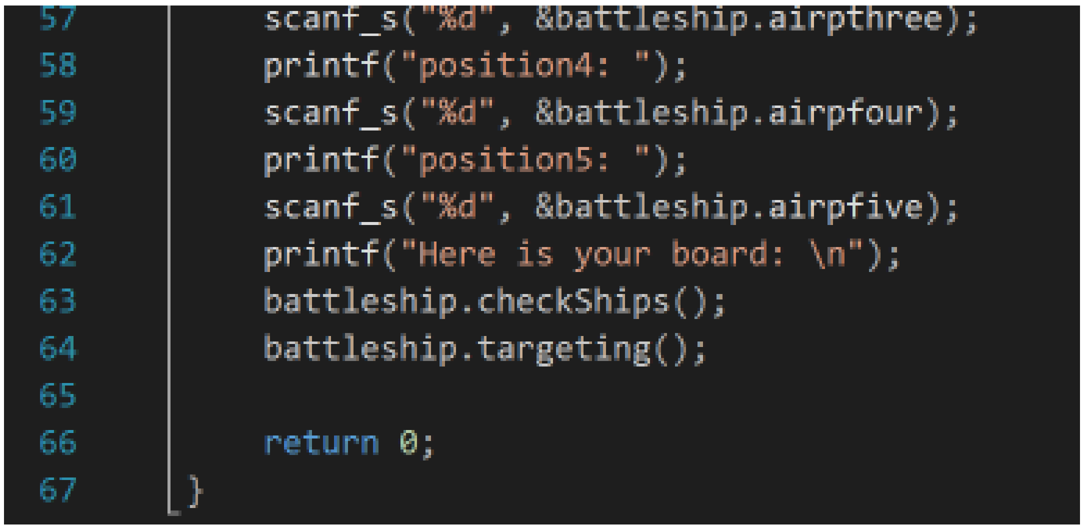
Figure 26: playGame() Method

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



```
int main()
{
    Battleship battleship;
    printf("Battle Ship\nBy Michael Marques\n");
    printf("These are the posible positions: \n");
    printf("11 ,12 ,13 ,14 ,15 ,16 ,17 ,18\n"); /* Displays posible ship positions */
    printf("21 ,22 ,23 ,24 ,25 ,26 ,27 ,28\n");
    printf("31 ,32 ,33 ,34 ,35 ,36 ,37 ,38\n");
    printf("41 ,42 ,43 ,44 ,45 ,46 ,47 ,48\n");
    printf("51 ,52 ,53 ,54 ,55 ,56 ,57 ,58\n");
    printf("61 ,62 ,63 ,64 ,65 ,66 ,67 ,68\n");
    printf("71 ,72 ,73 ,74 ,75 ,76 ,77 ,78\n");
    printf("81 ,82 ,83 ,84 ,85 ,86 ,87 ,88\n");
    printf("(3 spaces)Player 1 enter your Battle ship's poition: \n");
    printf("position1: ");          /* Gets you ships positions */
    scanf_s("%d", &battleship.battlepone);
    printf("position2: ");
    scanf_s("%d", &battleship.battleptwo);
    printf("position3: ");
    scanf_s("%d", &battleship.battlepthree);
    printf("(2 spaces)Enter your Patrol boat's poition: \n");
    printf("position1: ");
    scanf_s("%d", &battleship.patrolpone);
    printf("position2: ");
    scanf_s("%d", &battleship.patrolptwo);
    printf("(3 spaces)Enter your Subs's poition: \n");
    printf("position1: ");
    scanf_s("%d", &battleship.subpone);
    printf("position2: ");
    scanf_s("%d", &battleship.subptwo);
    printf("position3: ");
    scanf_s("%d", &battleship.subpthree);
    printf("(4 spaces)Enter your Destroyers's poition: \n");
    printf("position1: ");
    scanf_s("%d", &battleship.destroypone);
    printf("position2: ");
    scanf_s("%d", &battleship.destroyptwo);
    printf("position3: ");
    scanf_s("%d", &battleship.destroypthree);
    printf("position4: ");
    scanf_s("%d", &battleship.destroypfour);
    printf("(5 spaces)Enter your Air craft carier's poition: \n");
    printf("position1: ");
    scanf_s("%d", &battleship.airpone);
    printf("position2: ");
    scanf_s("%d", &battleship.airptwo);
    printf("position3: ");
```

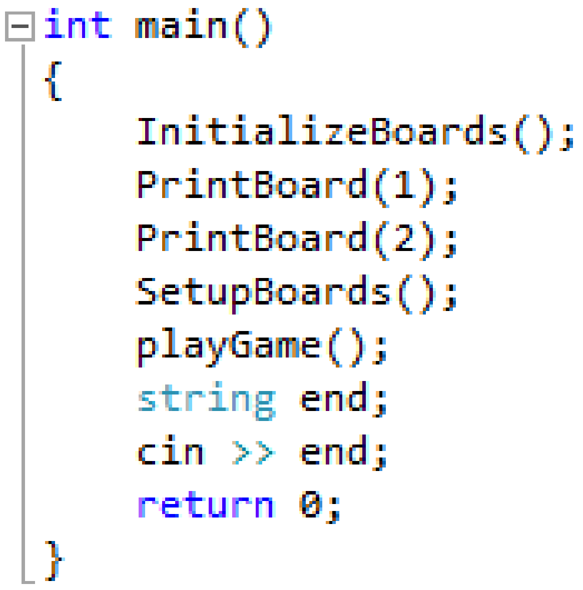
Figure 27: Original Main() Method



```
        scanf_s("%d", &battleship.airpthree);
        printf("position4: ");
        scanf_s("%d", &battleship.airpfour);
        printf("position5: ");
        scanf_s("%d", &battleship.airpfive);
        printf("Here is your board: \n");
        battleship.checkShips();
        battleship.targeting();

        return 0;
}
```

Figure 28: Original Main() Method

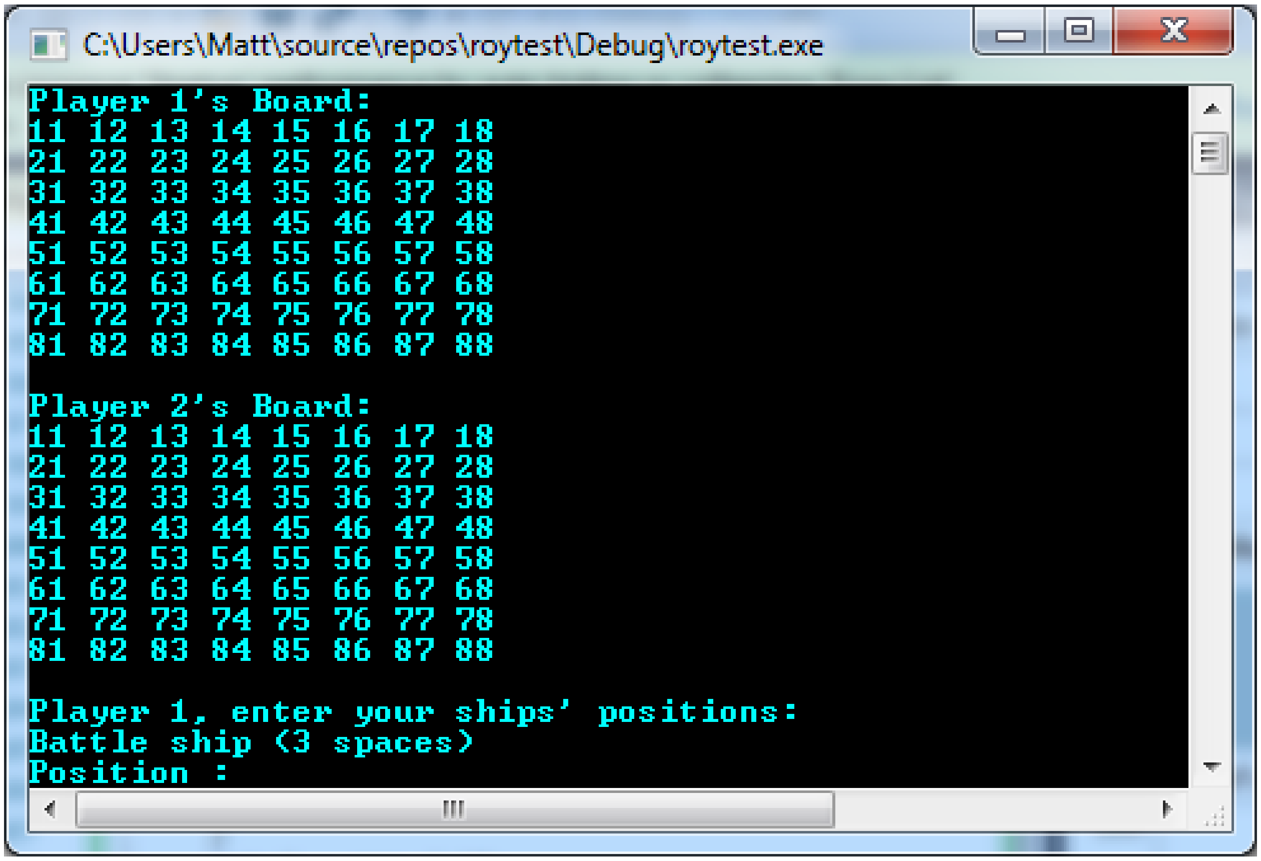
After all the refactoring done by creating methods, classes, renaming objects, and eliminating switch statements, we are able to write a clean, easy to read main method shown in Figure 29, below.



```
int main()
{
    InitializeBoards();
    PrintBoard(1);
    PrintBoard(2);
    SetupBoards();
    playGame();
    string end;
    cin >> end;
    return 0;
}
```

Figure 29: Main() Method

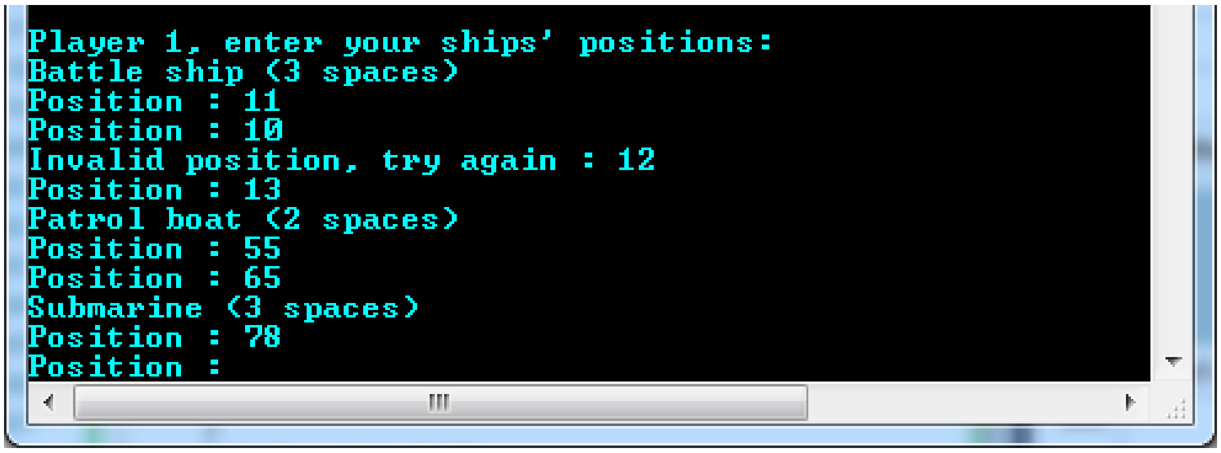
The Battleship game is now ready to be played. Figure 30, below, shows the InitializeBoard() and PrintBoard() methods being used.



```
Player 1's Board:
11 12 13 14 15 16 17 18
21 22 23 24 25 26 27 28
31 32 33 34 35 36 37 38
41 42 43 44 45 46 47 48
51 52 53 54 55 56 57 58
61 62 63 64 65 66 67 68
71 72 73 74 75 76 77 78
81 82 83 84 85 86 87 88

Player 2's Board:
11 12 13 14 15 16 17 18
21 22 23 24 25 26 27 28
31 32 33 34 35 36 37 38
41 42 43 44 45 46 47 48
51 52 53 54 55 56 57 58
61 62 63 64 65 66 67 68
71 72 73 74 75 76 77 78
81 82 83 84 85 86 87 88

Player 1, enter your ships' positions:
Battle ship (3 spaces)
Position :
```

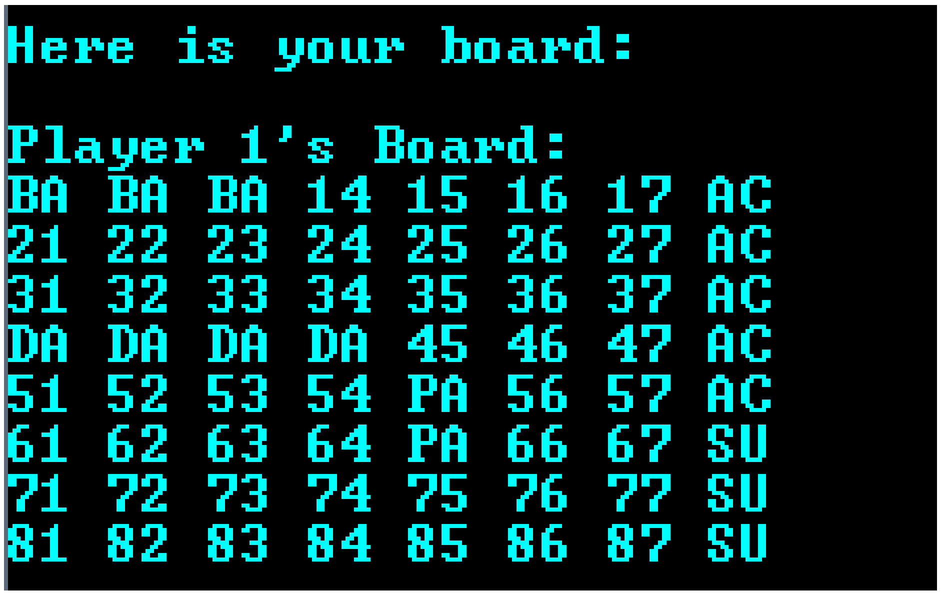
Figure 30: InitializeBoard() and PrintBoard() in action

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.



```
Player 1, enter your ships' positions:
Battle ship (3 spaces)
Position : 11
Position : 10
Invalid position, try again : 12
Position : 13
Patrol boat (2 spaces)
Position : 55
Position : 65
Submarine (3 spaces)
Position : 78
Position :
```

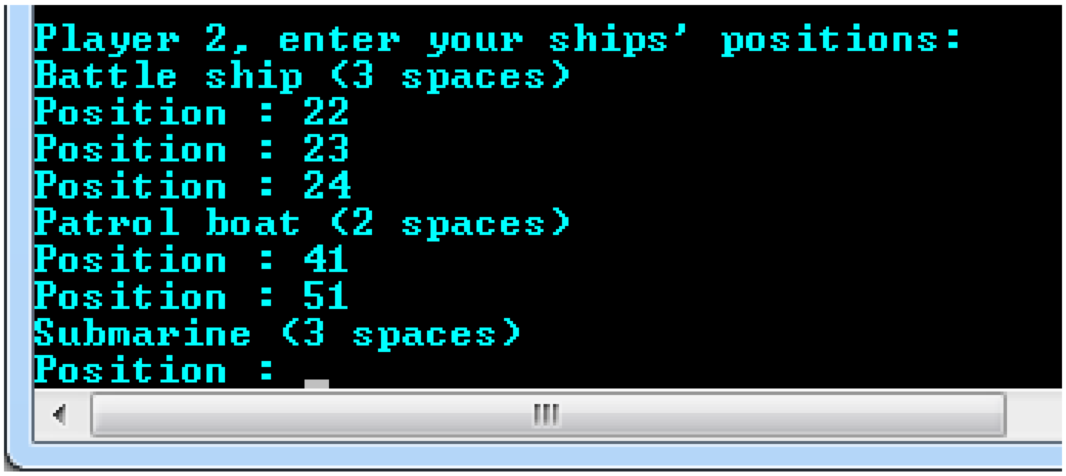
Figure 31: Set and Validate Ships for Player 1



```
Here is your board:

Player 1's Board:
BA BA BA 14 15 16 17 AC
21 22 23 24 25 26 27 AC
31 32 33 34 35 36 37 AC
DA DA DA DA 45 46 47 AC
51 52 53 54 PA 56 57 AC
61 62 63 64 PA 66 67 SU
71 72 73 74 75 76 77 SU
81 82 83 84 85 86 87 SU
```

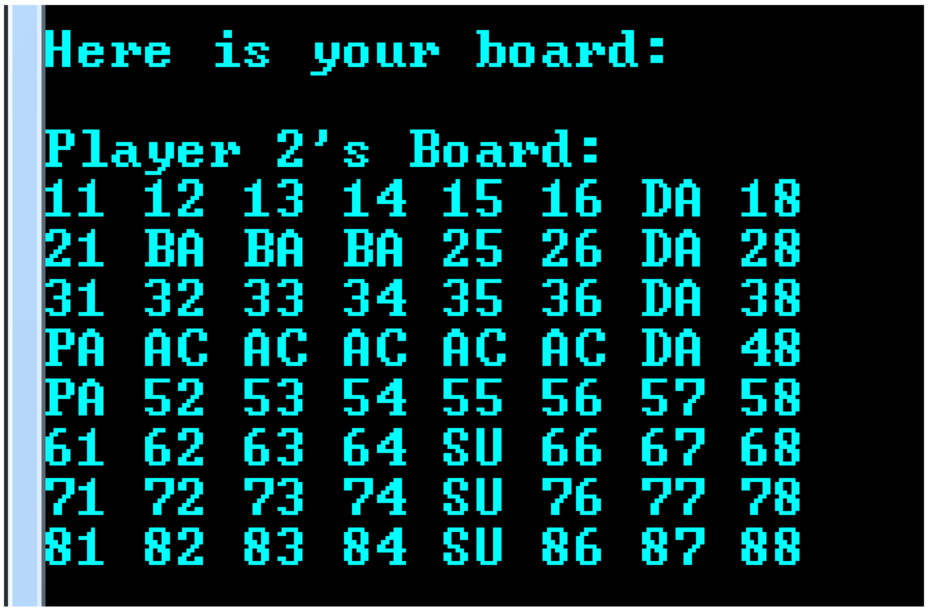
Figure 32: Player 1 Board with Ships Placed

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



```
Player 2, enter your ships' positions:
Battle ship (3 spaces)
Position : 22
Position : 23
Position : 24
Patrol boat (2 spaces)
Position : 41
Position : 51
Submarine (3 spaces)
Position : _
```

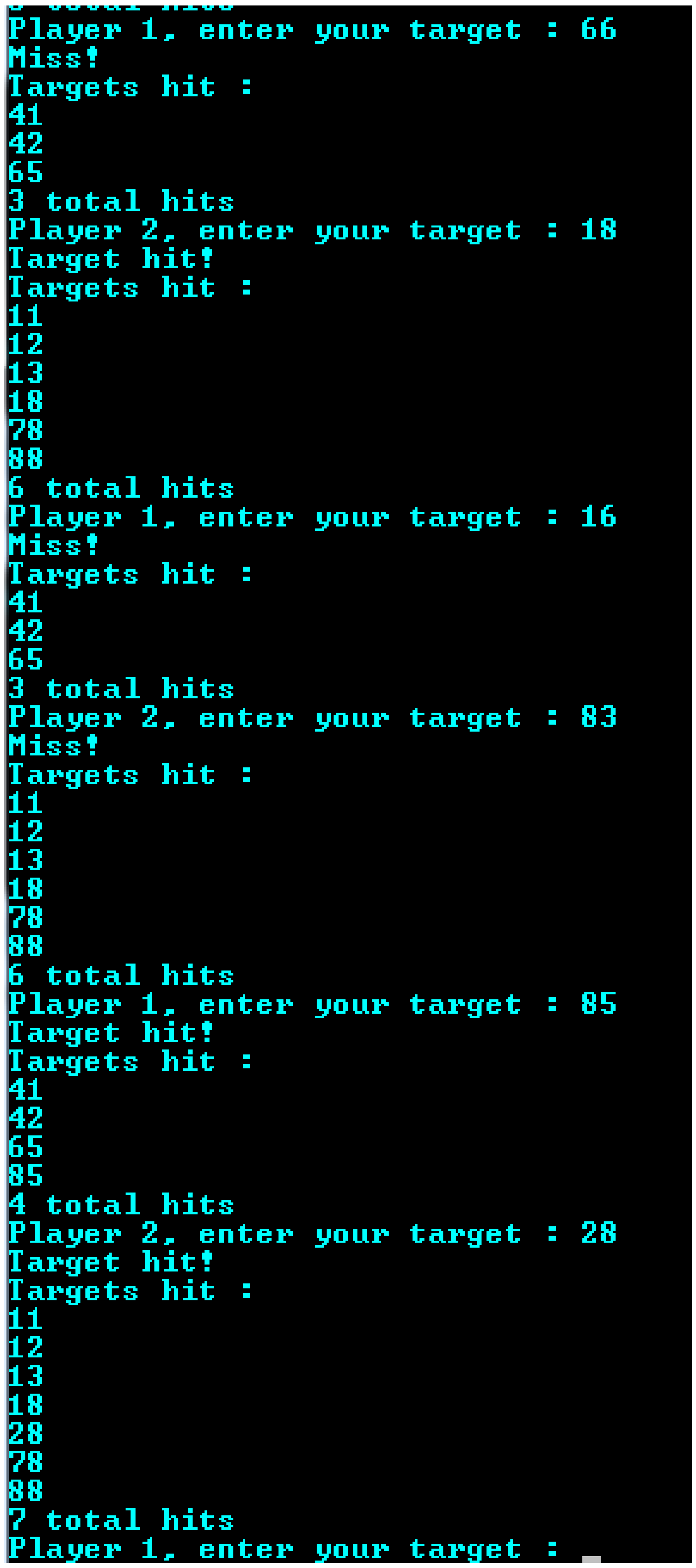
Figure 33: Set and Validate Ships for Player 2



```
Here is your board:

Player 2's Board:
11 12 13 14 15 16 DA 18
21 BA BA BA 25 26 DA 28
31 32 33 34 35 36 DA 38
PA AC AC AC AC AC DA 48
PA 52 53 54 55 56 57 58
61 62 63 64 SU 66 67 68
71 72 73 74 SU 76 77 78
81 82 83 84 SU 86 87 88
```

Figure 34: Player 2 Board with Ships Placed

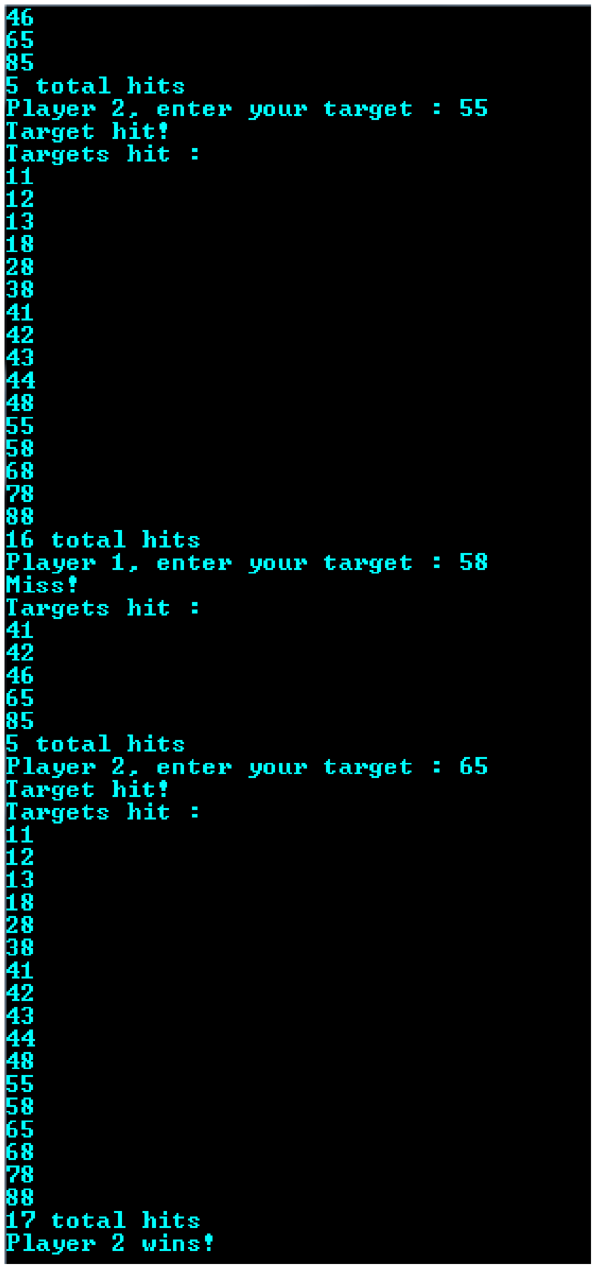
After both players have set up their boards, the playGame() method is called, which also calls the validateTraget(), fire(), countHits(), and printHits() methods. A sample of the game play can be seen in Figure 35, below.



```
Player 1, enter your target : 66
Miss!
Targets hit :
41
42
65
3 total hits
Player 2, enter your target : 18
Target hit!
Targets hit :
11
12
13
18
78
88
6 total hits
Player 1, enter your target : 16
Miss!
Targets hit :
41
42
65
3 total hits
Player 2, enter your target : 83
Miss!
Targets hit :
11
12
13
18
78
88
6 total hits
Player 1, enter your target : 85
Target hit!
Targets hit :
41
42
65
85
4 total hits
Player 2, enter your target : 28
Target hit!
Targets hit :
11
12
13
18
28
78
88
7 total hits
Player 1, enter your target : _
```

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.



```
46
65
85
5 total hits
Player 2, enter your target : 55
Target hit!
Targets hit :
11
12
13
18
28
38
41
42
43
44
48
55
58
68
78
88
16 total hits
Player 1, enter your target : 58
Miss!
Targets hit :
41
42
46
65
85
5 total hits
Player 2, enter your target : 65
Target hit!
Targets hit :
11
12
13
18
28
38
41
42
43
44
48
55
58
65
68
78
88
17 total hits
Player 2 wins!
```

Figure 36: Sample of Winning

References

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/

(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/

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