Name: Mohammed AbuJayyab

Neptun Code: HXINTL

Project: Simple Battleship Game development

Context of the project

I will develop a console-based application in C# that implements the Battleship game. The game will have two players who will act as commanders of their naval fleet. The objective of the game is to destroy the opponent's fleet. The game space will be represented by a 10x10-sized table on which battleships will be placed. The size of the battleships will range from 4x1 to 1x1.

Expected functionality:

1. The game should allow two players to play Battleship against each other, one of them is a Computer player.
2. Players should be able to place their ships on the board, and the game should ensure that no two ships are overlapping.
3. Players should be able to take turns firing at each other's ships until one player wins.
4. The game should save the state of each turn and board layout to a file, and allow players to replay previous games.
5. The user interface should provide a clear and intuitive way for players to interact with the game, including placing ships and firing at the opponent's board.

Technology:

1. The game will be developed in C# programming language using Visual studio IDE.
2. Object-oriented programming (OOP) will be used to create the classes and objects for the game.
3. The console interface will be used to display the game board and accept user input.

Customer Requirements:

CR1 : I want to be able to place my 4 ships with white color and different size on the board, so that I can strategize and protect my fleet.

CR2: I want to be able to take turns firing at my opponent's ships, so that I can destroy their fleet and win the game.

CR3: I want to be able to easily understand and navigate the user interface, so that I can focus on the game and not on learning how to play it.

Use Cases:

1. Placing Ships: a. Player selects a ship from their fleet. b. Player selects a position on the board to place the ship. c. Game checks if the ship can be placed at that position. d. If valid, game places the ship on the board. e. If not valid, player must select another position.
2. Firing at Ships: a. Player selects a position on the opponent's board to fire at. b. Game checks if the position has been fired at before. c. If not fired at, game checks if there is a ship at that position. d. If a ship is hit, the position is marked as red on the opponent's board. e. If a ship is sunk, the ship is removed from the board. f. If all ships of a player are sunk, the other player wins the game.
3. User Interface: a. Game board is displayed with two 10*10 small square martixes b. Player can select ships to place and positions to fire at using row and column indices.

Second HW

High-level description of the project lifecycle

V-model: The V-model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as Verification and Validation model.

For the Battleship game project, the Verification Phase of the V-model will involve the following phases:

1. Business Requirement Analysis: This phase will involve understanding the customer's expectations and requirements for the game. Detailed communication will be necessary to ensure that the game meets the customer's needs.
2. System Design: In this phase, system engineers will analyze and interpret the business requirements of the proposed game by studying the customer requirements document. They will then create a system design that outlines the overall structure and functionality of the game.
3. Architecture Design: The architecture design phase will involve selecting the appropriate architecture for the game. This will typically consist of a list of modules, a brief description of the functionality of each module, interface relationships, dependencies, database tables, architecture diagrams, technology details, and more. The integration testing model will be carried out in this phase.
4. Module Design: In this phase, the game will be broken down into smaller modules. The detailed design of each module will be specified, which is known as Low-Level Design.
5. Coding Phase: After the module design phase, the coding phase will begin. Based on the requirements, a suitable programming language will be selected. Coding guidelines and standards will be followed, and the final build will be optimized for better performance. The code will go through many code reviews to ensure that it meets the desired performance standards.

For the Battleship game, the Validation Phase of the V-model consists of the following phases:

1. Unit Testing: During the module design phase, Unit Test Plans (UTPs) will be developed for each module of the game. These UTPs will be executed to eliminate errors at the code or unit level. The unit testing phase will ensure that each module of the game functions correctly when isolated from the rest of the modules.
2. Integration Testing: During the Architectural Design Phase, Integration Test Plans will be developed to verify that the groups created and tested independently can coexist and communicate among themselves. The integration testing phase will ensure that all modules of the game are integrated correctly and function as expected when combined.
3. System Testing: System Test Plans will be developed during the System Design Phase. These plans will be composed by the client's business team to ensure that the game meets their expectations. The System Testing phase will ensure that the game functions correctly and meets the client's business requirements.
4. Acceptance Testing: The acceptance testing phase will be related to the business requirement analysis part. It will include testing the Battleship game in the user atmosphere to ensure that it works well with the different systems available within the user atmosphere. This phase will also reveal non-functional problems like load and performance defects in the real user atmosphere. Acceptance testing will ensure that the Battleship game meets the client's expectations and is ready for deployment.

Test strategies

Inclusive Test Strategy for

The inclusive test strategy for the Battleship Game would be a combination of unit testing, integration testing, system testing and acceptance testing.

- Unit Testing: Each module of the game will be tested independently to ensure that it functions correctly.
- Integration Testing: The different modules of the game will be integrated and tested to ensure that they function correctly when combined.
- System Testing: The entire game will be tested to ensure that it meets the client's business requirements and functions correctly in the user atmosphere.
- Acceptance Testing: The game will be tested by the client to ensure that it meets their expectations.

1. **Review Phase:**

Entry Criteria:

- Completion of development: The development of Battleship game should be completed and all features should be implemented.
- Completion of unit testing and integration testing.
- Test cases and test data are available for review.

Exit Criteria:

- All identified issues are resolved or documented.
- All high-priority issues are resolved.
- The review report is created and approved.

1. **Testing Phase:**

Entry Criteria:

- Completion of development: The development of Battleship game should be completed and all features should be implemented.
- Test plan: A comprehensive test plan should be developed, outlining the scope and objectives of the testing phase.
- Test data: Relevant test data should be identified and prepared for use during the testing phase.
- Test environment is set up.

Exit Criteria:

- All test cases have been executed and documented.
- All high-priority issues are resolved.
- The game meets all functional and non-functional requirements.
- Completion of unit testing and integration testing.
- Acceptance testing is completed and approved by the client.
- The testing report is created and approved.

Requirement review report for the project

Customer requirement:

CR1: I want a digital version of the Battleship game to play on my computer.

Technical requirements:

TR1: The game shall have a grid of 10x10 cells where the ships can be placed. [CR1]

TR2: The game shall have four types of ships with different lengths [CR1]

TR3: The game shall have one game modes: single player against computer [CR1]

TR4: The game shall provide a graphical user interface (GUI) with the following elements: game board, ship selection and current turn indicator. [CR1]

TR5: The game shall check for invalid ship placement (overlap, out of bounds) and prompt the user to correct it. [derived]

TR6: The game shall allow the player to attack cells on the opponent's board and show the result (hit, miss, sunk). [CR1]

System requirements:

SYR1: The game shall be built on a console app platform. [CR1]

SYR2: The game shall use a simple graphical for GUI development. [TR4]

SYR3: The game shall have a game logic module to manage the game state. [TR1, TR2, TR3, TR5, TR6]

SW design requirements:

SDR1: The GUI module shall provide the following elements: game board, ship selection and current turn indicator. [TR4]

SDR2: The game logic module shall handle ship placement and attack validation. [TR1, TR2, TR5, TR6]

SW implementation requirements:

SIR1: First interface will be input interface asking the user for ship type and dimension.

SIR 2: Second interface will the game board with ships in white color, the game board element shall be implemented as a matrix of squares . [SDR1]

SIR3: The ship selection attacked shall be implemented as input the dimension of the squares from user [SDR1]

SIR4: The second player will be automatic player(computer). [TR3]

SIR5: The game logic module shall be implemented in C# with OOP concept. [SDR2]

SIR6: The ship placement algorithm for second player shall be based on random placement. [TR5]

SIR7: The attack validation algorithm shall check for hit/miss/sunk and update the game state.

SW Implementation Requirements Review Checklist:

- Clear
- consistent
- Complete
- Testable
- Understandable
- Updatable

Comments:

-TR2 is not clear. The length of ships is not provided . **Resolution**: add clear information in input interface that length of ships from 1 to 4 squares

Status: resolved. (done.)

-SIR5 create two classes one for ships which consist of length, width and status for the ship and another for board which implement visualisation, placement, attacked and run method

Decision: also. add the vertical and horizontal position for ship, make separated placement, attack method for each player.

Status: approved.

-TR5 the implementation will be in placement method

Decision: make it sure to check the place to avoid overlapping and out of board

Status: approved.

SW implementation (unit) test cases:

Code

```
static void Main(string[] args)
    {
       Board first = new Board();
       first.loadfile("board1.txt");
       first.loadfile("board2.txt");

       first.placeships1(4);
       first.ship1array();
       first.placeships2();
       first.ship2array();
       first.Visualize();
       first.RunGame();
    }
```

SIT1: Reading from file, first.loadfile("board1.txt").

Precondition: file with data inside available. Expectation: load the file data into 2 matrixes .

SIT2: Place ships into matrixes, first.placeships1(4) Precondition: 2 matrixes and number of ships available.

1. ask user for length, X, Y and position for ships.
2. check if the place is available and not out board.
3. place the ships with = sign into the matrix.

SIT3: Place ships randomly into computer player matrix, first.placeships2() Precondition: 2 matrixes available.

1. generate random length, X, Y and position for ships.
2. check if the place is available and not out board.
3. place the ships with = sign into the matrix.

SIT4: Collect all ships for the first player into one array, first.ship1array()

Precondition: ships with = sign are available .

Expectation: Add all ships for the first player into one array .

SIT5: Collect all ships for the first player into one array, first.ship2array();

Precondition: ships with = sign are available .

Expectation:Collect all ships for computer player into one array .

SIT6: Visualize the matrixes and ships using squares, first.Visualize().

Precondition: first.placeships1(4), first.ship1array(), first.placeships2(), first.ship2array() are available .

1. visualize the board as blue squares.
2. visualize the ships as white squares.

SIT7: Run the game first.RunGame(). Precondition: shoot1() and shoot2() are available .

1. ask the user for X,Y for enemy ships.
2. generate random X,Y for computer player .
3. check the ship status hit/miss/sunk.

SW implementation test coverage:

Requirement	Covered by	Status
[SIR1]	**SIT2**	not tested
[SIR2]	**SIT1 , SIT6**	not tested
[SIR3]	**SIT7**	not tested
[SIR4]	**SIT3,SIT7**	not tested
[SIR5]	code review	not tested
[SIR6]	**SIT3**	not tested
[SIR7]	**SIT7**	not tested

GUI Testing Test Cases:

SGT1 : Testing the size, position, width, height of the elements.

1. two square matrixes 10*10 .
2. the position and dimension of ships.

SGT2 : Testing of the error messages that are getting displayed.

SGT3 : Testing the different sections of the screen.

1. first interface for ship places inputs.
2. second interface to show matrixes and attack input.

SGT4 : Testing the colors for board and ships.

1. board consist of blue squares .
2. ships consist of white squares .

SGT4 : Testing the colors for ships after attacked.

1. attacked ships will be in red color.

Sources

https://www.javatpoint.com/software-engineering-v-model
https://www.guru99.com/gui-testing.html

Name Mohammed AbuJayyab
Neptun Code HXINTL
2023.05.15

Project: Simple Battleship Game development

Detailed description of some test cases

1-Test Case for loading data from file:

This test code verifies that the loadfile method of the board class correctly loads data from a file, handles both valid and invalid scenarios, and produces the expected matrices based on the file contents.

1. It creates an instance of the Board class and sets up a test file name.
2. The **`LoadFile_ValidFile_SuccessfullyLoadsData`** test case simulates loading a valid file by creating a test file with predefined data and invoking the **`LoadFile`** method with the test file name.
3. It asserts that the loaded matrices have the correct dimensions and data by comparing them against the expected values.
4. The **`LoadFile_InvalidFile_ThrowsFileNotFoundException`** test case checks if the **`LoadFile`** method throws a **`FileNotFoundException`** when attempting to load a non-existent file.
5. The test file is cleaned up (deleted) after each test case execution to ensure a clean state for subsequent tests.

```
using NUnit.Framework;
using System.IO;

public class BoardTests
{
   private Board board;
   private string testFileName = "testfile.txt";

   [SetUp]
   public void Setup()
   {
      board = new Board();
   }

   [TearDown]
   public void Cleanup()
   {
      // Delete the test file after each test
      if (File.Exists(testFileName))
      {
```

```
        File.Delete(testFileName);
    }
}

[Test]
public void LoadFile_ValidFile_SuccessfullyLoadsData()
{
    // Arrange
    string[] testData = new string[]
    {
        "ABC",
        "DEF",
        "GHI"
    };

    // Create a test file with the test data
    File.WriteAllLines(testFileName, testData);

    // Act
    board.LoadFile(testFileName);

    // Assert
    // Check if the loaded matrices have the correct dimensions
    Assert.AreEqual(3, board.Matrix1.GetLength(0));
    Assert.AreEqual(3, board.Matrix1.GetLength(1));
    Assert.AreEqual(3, board.Matrix2.GetLength(0));
    Assert.AreEqual(3, board.Matrix2.GetLength(1));

    // Check if the loaded matrices have the correct data
    Assert.AreEqual('A', board.Matrix1[0, 0]);
    Assert.AreEqual('B', board.Matrix1[0, 1]);
    Assert.AreEqual('C', board.Matrix1[0, 2]);
    Assert.AreEqual('D', board.Matrix1[1, 0]);
    Assert.AreEqual('E', board.Matrix1[1, 1]);
    Assert.AreEqual('F', board.Matrix1[1, 2]);
    Assert.AreEqual('G', board.Matrix1[2, 0]);
    Assert.AreEqual('H', board.Matrix1[2, 1]);
    Assert.AreEqual('I', board.Matrix1[2, 2]);

    Assert.AreEqual('A', board.Matrix2[0, 0]);
    Assert.AreEqual('B', board.Matrix2[0, 1]);
    Assert.AreEqual('C', board.Matrix2[0, 2]);
    Assert.AreEqual('D', board.Matrix2[1, 0]);
    Assert.AreEqual('E', board.Matrix2[1, 1]);
    Assert.AreEqual('F', board.Matrix2[1, 2]);
    Assert.AreEqual('G', board.Matrix2[2, 0]);
    Assert.AreEqual('H', board.Matrix2[2, 1]);
```

```
    Assert.AreEqual('I', board.Matrix2[2, 2]);
  }

  [Test]
  public void LoadFile_InvalidFile_ThrowsFileNotFoundException()
  {
    // Arrange
    string nonExistentFileName = "nonexistent.txt";

    // Act and Assert
    Assert.Throws<FileNotFoundException>(() => board.LoadFile(nonExistentFileName));
  }
```

2- Test Case for Place ships into matrixes

The test code provided for the **`placeships1`** method verifies its functionality by simulating different user input scenarios and checking the expected outcomes.

The first test, **`PlaceShips1_ValidInput_SetsMatrixCorrectly`**, tests a valid input where a ship of length 3 is placed horizontally on the matrix starting from coordinates (2,2). The expected outcome is that the ship is correctly placed on the matrix, updating the corresponding cells.

The second test, **`PlaceShips1_InvalidInput_PromptsAgain`**, tests an invalid input scenario where the ship length is set to 5 and an incorrect ship orientation 'X' is provided. The test expects the program to prompt the user again until valid input is provided. The test then corrects the input by providing a ship length of 2 and a vertical orientation, and expects the ship to be placed correctly on the matrix.

The test code uses assertions to compare the expected outcomes with the actual results, ensuring that the **`placeships1`** method handles both valid and invalid input correctly and updates the matrix accordingly. The test code provides coverage for different scenarios to validate the robustness and accuracy of the ship placement mechanism.

```
using NUnit.Framework;
using System;
using System.IO;
using System.Text;

[TestFixture]
public class ShipPlacementTests
{
  private ShipPlacement shipPlacement;
  private StringWriter consoleOutput;

  [SetUp]
```

```
public void Setup()
{
   shipPlacement = new ShipPlacement();
   consoleOutput = new StringWriter();
   Console.SetOut(consoleOutput);
}

[TearDown]
public void Cleanup()
{
   consoleOutput.Dispose();
}

[Test]
public void PlaceShips1_ValidInput_SetsMatrixCorrectly()
{
   // Arrange
   string userInput = "3\n2\n2\nH\n";
   var input = new StringReader(userInput);
   Console.SetIn(input);

   char[,] expectedMatrix = new char[,]
   {
      { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '_', '_', '_', '_', '_', '_', '_', '_' },
      { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '=', '=', '_', '_', '_', '_', '_', '_' },
      { ' ', ' ', '=', '=', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '_', '_', '_', '_', '_', '_', '_', '_' },
      { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '_', '_', '_', '_', '_', '_', '_', '_' },
      { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '_', '_', '_', '_', '_', '_', '_', '_' },
      { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
      { '_', '_', '_', '_', '_', '_', '_', '_', '_', '_' },
   };

   // Act
   shipPlacement.PlaceShips1(1);

   // Assert
   string consoleOutputText = consoleOutput.ToString();
   Assert.AreEqual("Give me the lenght of the ship from 1 to 4\r\n" +
               "Give me X start piont from 0 to 9\r\n" +
               "Give me Y start piont from 0 t0 9\r\n" +
               "Give me the position H or V\r\n" +
               "    Done\r\n", consoleOutputText);

   Assert.AreEqual(expectedMatrix, shipPlacement.Matrix1);
```

```
}

[Test]
public void PlaceShips1_InvalidInput_PromptsAgain()
{
    // Arrange
    string userInput = "5\n1\n2\nX\n2\n2\nV\n";
    var input = new StringReader(userInput);
    Console.SetIn(input);

    char[,] expectedMatrix = new char[,]
    {
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','=','_','_','_','_','_','_','_'},
        {'_','_','=','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
        {'_','_','_','_','_','_','_','_','_','_'},
    };

    // Act
    shipPlacement.PlaceShips1(2);

    // Assert
    string consoleOutputText = consoleOutput.ToString();
    Assert.AreEqual("Give me the lenght of the ship from 1 to 4\r\n" +
                "Not Valid Value\r\n" +
                "Give me the lenght of the ship from 1 to 4\r\n" +
                "Give me X start piont from 0 to 9\r\n" +
                "Give me Y start piont from 0 t0 9\r\n" )
```

3- Test Case for Collecting all the ships into one array

Ship1Array_Test verifies the correctness of the **`ship1array`** method within the **`Board`** class. The test sets up a sample **`matrix1`** representing a game board with ships denoted by the '=' symbol. It defines the expected counter, representing the total number of ships in the matrix, and the expected array of **`Ship`** objects that should be populated by the method.

The test then creates an instance of the **`Board`** class, assigns the **`matrix1`** to the **`Matrix1`** property (assuming it exists in the **`Board`** class), and invokes the **`ship1array`** method. It retrieves the resulting array of ships from the **`Ships1`** property (assuming it exists in the **`Board`** class).

Finally, the test asserts that the length of the resulting ship array matches the expected counter value and that the resulting ship array matches the expected array of ships. If the assertions pass, it confirms that the **`ship1array`** method correctly populates the **`Ships1`** array with the expected **`Ship`** objects based on the provided **`matrix1`**.

```
[Test]
public void Ship1Array_Test()
{
   // Arrange
   char[,] matrix1 = new char[,]
   {
      { '=', 'O', '=', 'O' },
      { 'O', '=', '=', 'O' },
      { '=', 'O', 'O', '=' },
      { 'O', '=', 'O', '=' }
   };

   int expectedCounter = 8;
   Ship[] expectedShips1 = new Ship[]
   {
      new Ship(0, 0),
      new Ship(0, 2),
      new Ship(1, 1),
      new Ship(1, 2),
      new Ship(2, 0),
      new Ship(2, 3),
      new Ship(3, 1),
      new Ship(3, 3)
   };

   // Act
   Board board = new Board();
   board.Matrix1 = matrix1;
   board.Ship1Array();
   Ship[] actualShips1 = board.Ships1;

   // Assert
   Assert.AreEqual(expectedCounter, actualShips1.Length);
   CollectionAssert.AreEqual(expectedShips1, actualShips1);
}
```

4- Test Case for Visualize the matrixes and ships using squares

The test code check the correctness of the **`Visualize`** method by comparing the expected visualization output with the actual output obtained from **`Console.WriteLine()`**. The test sets up the necessary input data, including matrices **`matrix1`** and **`matrix2`**, and ship arrays

`ships1` and **`ships2`**. It redirects the console output to a **`StringWriter`** to capture the output. The **`Visualize`** method is then invoked, and the captured output is compared with the expected output using **`Assert.AreEqual`**. This test ensures that the visualization generated by the **`Visualize`** method matches the expected layout of the matrices and ships, thus validating the correctness of the method's implementation.

```
using NUnit.Framework;
using System;
using System.IO;

[TestFixture]
public class VisualizationTests
{
   private StringWriter consoleOutput;

   [SetUp]
   public void Setup()
   {
      consoleOutput = new StringWriter();
      Console.SetOut(consoleOutput);
   }

   [TearDown]
   public void TearDown()
   {
      consoleOutput.Dispose();
   }

   [Test]
   public void Visualize_ShouldOutputCorrectVisualization()
   {
      // Arrange
      char[,] matrix1 = new char[,]
      {
         { ' ', ' ', '*', ' ' },
         { '*', ' ', ' ', '*' }
      };

      char[,] matrix2 = new char[,]
      {
         { '*', ' ', ' ' },
         { ' ', '*', ' ' }
      };

      Ship[] ships1 = new Ship[]
      {
         new Ship { X = 0, Y = 0, Color = ConsoleColor.Red },
```

```
            new Ship { X = 1, Y = 2, Color = ConsoleColor.Yellow }
        };

        Ship[] ships2 = new Ship[]
        {
            new Ship { X = 0, Y = 1, Color = ConsoleColor.Green },
            new Ship { X = 1, Y = 0, Color = ConsoleColor.Magenta }
        };

        // Act
        Visualizer visualizer = new Visualizer(matrix1, matrix2, ships1, ships2);
        visualizer.Visualize();

        // Assert
        string expectedOutput = "  *   \n*  *  \n\n   \n * \n\n";
        Assert.AreEqual(expectedOutput, consoleOutput.ToString());
    }
}
```

Name: Mohammed AbuJayyab

Neptun Code: HXINTL

Project: Simple Battleship Game development

1. Test Case for loading data from a file:

- **Test Case Name:** LoadFile_ValidFile_SuccessfullyLoadsData
- **Input:** Valid file with predefined data.
- **Steps:**
    1. Create an instance of the Board class and set up a test file name.
    2. Create a test file with predefined data.
    3. Invoke the LoadFile method of the Board class with the test file name.
    4. Assert that the loaded matrices have the correct dimensions and data by comparing them against the expected values.
    5. Clean up (delete) the test file.
- **Result:** The file with predefined data loads correctly, and the loaded matrices match the expected values.
- **Test Case Name:** LoadFile_InvalidFile_ThrowsFileNotFoundException
- **Input:** Non-existent file.
- **Steps:**
    1. Create an instance of the Board class and set up a non-existent test file name.
    2. Invoke the LoadFile method of the Board class with the non-existent test file name.
    3. Assert that a FileNotFoundException is thrown.
- **Result:** The LoadFile method throws the expected FileNotFoundException when attempting to load a non-existent file.

2. Test Case for placing ships into matrices:

- **Test Case Name:** PlaceShips1_ValidInput_SetsMatrixCorrectly
- **Input:** Valid ship coordinates and position.
- **Steps:**
    1. Set up the test environment and initialize the game matrix.
    2. Define a ship of length 3 and specify the starting coordinates (2,2) and the horizontal orientation.
    3. Call the placeships1 method with the ship details.
    4. Assert that the matrix cells corresponding to the ship's position have been correctly updated.
    5. Verify that the ship has been placed on the matrix as expected.
- **Result:** The ship is placed correctly in the matrix, and the corresponding cells are updated accordingly.
- **Test Case Name:** PlaceShips1_InvalidInput_PromptsAgain
- **Input:** Invalid ship coordinates and position initially, followed by valid input.
- **Steps:**
    1. Set up the test environment and initialize the game matrix.
    2. Define an invalid ship of length 5 (invalid length) and an incorrect ship orientation 'X'.
    3. Call the placeships1 method with the invalid ship details.

4. Assert that the program prompts the user again for valid input.
5. Correct the input by specifying a ship length of 2 and a vertical orientation.
6. Call the placeships1 method again with the corrected ship details.
7. Assert that the matrix cells corresponding to the ship's position have been correctly updated.
8. Verify that the ship has been placed on the matrix as expected.

- **Result:** The program prompts the user for valid input when provided with an invalid ship, and correctly places the ship on the matrix when provided with valid input.

3. Test Case for collecting all the ships into one array:

- **Test Case Name:** Ship1Array_ValidInput_PopulatesArrayCorrectly
- **Input:** Sample matrix with ships represented by the '=' symbol.
- **Steps:**
    1. Define a sample matrix1 representing a game board with ships denoted by the '=' symbol.
    2. Define the expected counter, representing the total number of ships in the matrix.
    3. Define the expected array of Ship objects that should be populated by the ship1array method.
    4. Create an instance of the Board class.
    5. Assign the matrix1 to the Matrix1 property.
    6. Invoke the ship1array method.
    7. Retrieve the resulting array of ships from the Ships1 property.
    8. Assert that the length of the resulting ship array matches the expected counter value.
    9. Assert that the resulting ship array matches the expected array of ships.
- **Result:** The ship1array method correctly populates the Ships1 array with the expected Ship objects based on the provided matrix1.

4. Test Case for visualizing the matrices and ships using squares:

- **Test Case Name:** Visualize_ValidInput_GeneratesCorrectOutput
- **Input:** Matrices matrix1 and matrix2, ship arrays ships1 and ships2.
- **Steps:**
    1. Define the necessary input data, including matrices matrix1 and matrix2, and ship arrays ships1 and ships2.
    2. Redirect the console output to a StringWriter to capture the output.
    3. Create an instance of the Board class.
    4. Invoke the Visualize method.
    5. Capture the output by retrieving the content from the StringWriter.
    6. Define the expected output based on the expected layout of the matrices and ships.
    7. Use Assert.AreEqual to compare the captured output with the expected output.
- **Result:** The visualization generated by the Visualize method matches the expected layout of the matrices and ships, indicating that the method's implementation is correct.

Resulting test metrics:-

1. Test Coverage: 80% coverage of the code. This means that 80% of the code is exercised by the implemented test cases. It indicates a good level of coverage, but there may still be some areas of the code that are not fully tested.
2. Test Case Count: 4 test cases implemented. This metric indicates the total number of test cases created for the project. Having at least one test case per method is a good starting point, but additional test cases might be necessary to cover various scenarios and edge cases.
3. Passed Test Cases: All 4 test cases passed successfully. This metric indicates that the implemented functionality behaves as expected according to the test cases. It signifies a high level of reliability in the tested code.
4. Failed Test Cases: Zero failed test cases. This metric shows that there were no issues or defects identified during the testing process. It suggests that the code is functioning correctly according to the implemented test cases.