Testing Document

Group 2

Contents

1 Changelog: Major Additions

- Subsection 3.1 Valid movements
- Subsection 4.9 Board placement C# code
- Section 5 Save
- Section 6 Load

2 Introduction

This document will outline testing procedures for the Checkers game. When first starting the game, the user is shown a menu with two(2) options - both of which correspond to a method of board setup

1. PLAY
2. CUSTOM
3. LOAD

3 PLAY

When the user clicks on PLAY, a standard 8x8 checkers board with all 24 pieces (12 white, 12 black) should be generated. To test this functionality click on the play button and board is displayed. On first run, the board will be the default checkers setup.

3.1 Valid Movements

To move a piece in the game, the user is required to use a mouse to click and drag the piece to a tile on the board. If the held piece belongs to the player, and movement is valid for that piece type, the piece moves to the released location, otherwise the piece snaps back to its original location. To determine what is actally a legal movement, we run a method called setValidMovements(board) to generate valid movement information which is stored in the piece. Note: Code is also in UnitTests.cs.

Testing setValidMovements()

- Testing on the simple case of one piece, A1=W. The test checks if the actual movement matches the expected movement.

```
48 [TestMethod]
49 public void setValidMovementsTestSimpleCase()
50 {
51     using (Game1 game = new Game1())
52     {
53         Board board = new Board();
54         board.setUpBoard("A1=W"); // White piece in bottom
       left corner
55         String[] expectedResult = { "illegal", "1,1", "illegal
       ", "illegal" };
56         String[] actualResult = new String[4];
57 
58         game.setValidMovements(board, 0, 0);
59         for (int i = 0; i < 4; i++)
60         {
61             actualResult[i] = board.getPiece(0, 0).
       getValidMovements()[i].ToString();
```

```
62              if (actualResult[i].Split(',')[0] == "-99")
    actualResult[i] = "illegal";
63              Console.WriteLine(expectedResult[i] + " : " +
    actualResult[i]);
64              Assert.AreEqual(expectedResult[i], actualResult[i
    ]);
65          }
66
67      }
68  }
```

- Testing the default board of 12+12 pieces. Read UnitTests.cs for the full code.

```
70  [TestMethod]
71  public void BoardLegalMovesTest_DefaultSetup()
72  {
73      // The default new game board
74      String boardSetup = "A1=W,A3=W,A7=B,B2=W,B6=B,B8=B,C1=W,C3
        =W,C7=B,D2=W,D6=B,D8=B,E1=W,E3=W,E7=B,F2=W,F6=B,F8=B,G1=W,
        G3=W,G7=B,H2=W,H6=B,H8=B";
75      String[][] expectedValidMoves = ...
76      BoardLegalMovesTest(boardSetup, expectedValidMoves);
77  }
```

- Testing a board with kings and pieces all over. Read UnitTests.cs for the full code.

```
80  [TestMethod]
81  public void BoardLegalMovesTest_MidgameSetup()
82  {
83      // A setup of the middle of a checkers match with multiple
         kings.
84      String boardSetup = "A1=W,A3=W,A7=B,B4=B,B8=B,C1=BK,C5=B,
        C7=W,D4=BK,D6=W,E3=W,F4=B,F6=B,F8=WK,G1=W,G3=W,G5=W,G7=WK,
        H6=B";
85      String[][] expectedValidMoves = ...
86      BoardLegalMovesTest(boardSetup, expectedValidMoves);
87  }
```

4 CUSTOM

When the user clicks on CUSTOM, they will be prompted to enter positions for all of their pieces.

4.1 No Input

In the event that the user inputs nothing, the console will display an incorrect input message and prompt the user for a correct input

Test Cases

- Return Key

4.2 Input Incorrectly Formatted

In the event that the user inputs the piece information incorrectly, the console will display an appropriate incorrect input message and prompt the user for a correct input.

Test Cases

- c
- A1=e
- W=A1
- A1 = W
- A1 = KW
- 2=2
- =====
- !@**
- A(5-4)=W
- AA3=W
- A1=W, C1=B
- A1=W, C1=B
- G1=34,A7=B

4.3 Invalid Location - Not on Solid Square

In the event that the user inputs a location that corresponds to a light square instead of a solid square, the console will display an appropriate incorrect input message and prompt the user for a correct input

Test Cases

- A2=W
- B1=B
- C4=W
- F5=W
- H3=B

4.4 Invalid Location - Out Of Board Bounds

In the event that the user inputs a location that does not exist on the board, the console will display an incorrect input message and prompt the user for a correct input.

Test Cases

- A9=B
- B12=W
- I1=W
- J10=B

4.5 Too Many White Pieces

In the event that the user inputs too many white pieces (>12), the console will display an appropriate input message (along with how many pieces you inputted) and prompt the user for a correct input.

Test Cases

- A1=W,C1=W,E1=W,G1=W,A3=W,A5=W,A7=W,B8=W,B6=W,B4=W, B2=W,E1=W,E3=W
- E5=B,A1=W,C1=W,E1=W,G1=W,A3=W,A5=W,A7=W,B8=W,B6=W, B4=W,B2=W,E1=W,E3=W

4.6 Too Many Black Pieces

In the event that the user inputs too many black pieces (>12), the console will display an appropriate input message (along with how many pieces you inputted) and prompt the user for a correct input.

Test Cases

- A1=B,C1=B,E1=B,G1=B,A3=B,A5=B,A7=B,B8=B,B6=B,B4=B, B2=B,E1=B,E3=B
- A1=B,C1=B,E1=B,G1=B,A3=B,A5=B,A7=B,B8=B,B6=B,B4=B, B2=B,E1=B,E3=B,E5=W

4.7 Overlapping Pieces

In the event that the user inputs a location that is already filled with a piece, the previous piece will be overwritten by the new piece

Test Cases

- A1=B,A1=W
- A1=W,A1=WK
- A1=W,A3=B,A1=B,A3=W
- A1=W,A1=W

4.8 Accepted Board Configurations

In the event that the user inputs the correct format, the console will display a confirmation message and the custom game board will be generated

Test Cases

- A1=W
- a1=w
- A3=W,B2=B

4.9 Testing in Code (UnitTests.cs)

- A method to test board setups that should pass. In this example, there are three different board setups. These setups pass the tests described in the previous sections above.

```
10 [TestMethod]
11 public void BoardTestPass()
12 {
13     try
14     {
15         Board board = new Board();
16         board.setUpBoard("A1=W, A3=W, A7=B, B2=W, B6=B, B8=B, C1=W,
       C3=W, C7=B, D2=W, D6=B, D8=B, E1=W, E3=W, E7=B, F2=W, F6=B, F8=B, G1=
       W, G3=W, G7=B, H2=W, H6=B, H8=B");
17         board.setUpBoard("A1=W, A3=W, A7=B, B2=W, B6=B, B8=B, C1=W,
       C5=W, D8=B, E1=BK, E5=W, E7=B, F6=W, F8=B, G1=W, G7=B, H2=W, H6=B, H8
       =B");
18         board.setUpBoard("a1=wk, A3=bk, A7=bK, B2=Wk, B6=B, B8=B, C1
       =W, C5=W, D8=B, E1=Bk, e5=W, e7=B, F6=W, F8=B, g1=W, G7=B, H2=W, H6=B
       , h8=B");
19     }
20     catch
21     {
22         Assert.Fail(); // No input should reach this.
23     }
24 }
```

- A Method to test board setups that should fail. In this example, there are 9 setups to test. Each one fails in a way as described in the sections above.

```
25 [TestMethod]
26 public void BoardTestFail()
27 { // This test will return successfully if all of the inputs
       fail.
28     Board board = new Board();
29     String[] badSetups = { "\n", ",", "A1=W ,", "A1=FFFF", "
       A1=FFX", "Aq=vvcxW", "A2=W, B2=BK", "A1=KKKK",
30                            "A1=W, A3=W, A7=B, B2=W, B6=B, B8=B, C1=W,
       C5=W, D8=B, E1=BK, E5=W, E8=B, F6=W, F8=B, G1=W, G7=B, H2=W, H6=B, H8
       =B"};
31     foreach (String input in badSetups)
32     {
33         bool didFail = false;
34         try
35         {
36             board.setUpBoard(input);
37         }
38         catch (Exception e)
39         {
40             didFail = true;
41         }
42         finally
43         {
44             Assert.IsTrue(didFail); // Each input should fail.
45         }
46     }
```

5 SAVE

The user can press the save button at any time while playing. Currently, the save file is saved into a folder on the desktop. If the user doesn't have write permissions to the desktop, a message will appear saying "Save Unsuccessful".

6 LOAD

When the user clicks on LOAD, if the savefile exists, the game will switch to the playing state with the board set up. Loading of board configurations is handled internally by the same system as SETUP so the same limitations are shared, such as: the user cannot load a board with more than 12 pieces for each player.

6.1 Save File

If the save file exists, but the text inside doesn't hold legal parsable information, the game will stay in the menu and tells the user a save cannot be found. Only the first two lines of the save file are parsed. The first line contains who's turn it is to go out of BLACK, WHITE. And the second line contains the board setup

as a string in the same format as inputted by a user. The user should not be writing in this file manually.

Test Cases

For each test case: We load the game and check if it is the right player's turn, if the pieces are in the correct place, and if the pieces's valid movements are correct. The following are 4 test cases with actual results matching expected results:

- Default board (Pass)

```
BLACK
A1=W,A3=W,A7=B,B2=W,B6=B,B8=B,C1=W,C3=W,C7=B,D2=W,D6=B,D8=B,E1=W,E3=W,E7=B,F2=W,F6=B,F8=B,G1=W,G3=W,G7=B,H2=W,H6=B,H8=B
```

- Custom board (Pass)

```
BLACK
A1=W,A3=W,A7=B,B4=B,B8=B,C1=BK,C5=B,C7=W,D4=BK,D6=W,E3=W,F4=B,F6=B,F8=WK,G1=W,G3=W,G5=W,G7=WK,H6=B
```

- Empty board (Fail)

```
blAcK

```

- Extra information (Pass)

```
WHITE
a1=wk,a3=w,A7=B,B4=B,B8=B,C1=BK,C5=B,C7=W,D4=BK,D6=W,E3=W,F4=B,F6=B,F8=WK,G1=W,G3=W,G5=W,G7=WK,H6=B
extra information and junk here
is allowed in case we want to save score and such later on

```

7 Assignment 3

7.1 Structural Coverage Testing

Each piece has 8 possible moves, the following function isMovementLegal will test each of the cases for movement assuming no pieces need to jump.

Test set that covers all statements (excluding else branches):

```
{<x = 1, y = 1, p = WH>, <x = 1, y = -1, p = WH>, <x = -1, y = 1, p = WH>, <x =
  -1, y = -1, p = WH>,
<x = 2, y = 2, p = WH>, <x = 2, y = -2, p = WH>, <x = -2, y = 2, p = WH>, <x =
-2, y = -2, p = WH>,
<x = 1, y = 1, p = BL>, <x = 1, y = -1, p = BL>, <x = -1, y = 1, p = BL>, <x =
-1, y = -1, p = BL>,
<x = 2, y = 2, p = BL>, <x = 2, y = -2, p = BL>, <x = -2, y = 2, p = BL>, <x =
-2, y = -2, p = BL>}
```

```
Given:
x = {-2,-1,1,2} -- relative grid movements right and left
y = {-2,-1,1,2} -- relative grid movements up or down
p = {BL, WH} -- black or white piece
King = {True, False} -- attribute of piece being a king
occupied(x,y) = true if relative board space is nonempty, else
    false


bool isMovementLegal (x, y, p)
    if (x = 1 || x = -1):
        if (y = 1):
            if (!occupied(x,y) && ( King || p = WH )):
                return true
            else:
                return false
        elseif (y = -1):
            if (!occupied(x,y) && ( King || p = BL )):
                return true
            else:
                return false
        else:
            return false
    elseif (x = 2 || x = -2):
        if (y = 2):
            if (occupied(x-1,y-1) && ( King || p = WH )):
                return true
            else:
                return false
        elseif (y = -2):
            if (occupied(x-1,y-1) && ( King || p = BL )):
                return true
            else:
                return false
        else:
            return false
    else:
        return false
endfunc
```

7.2 System testing & Acceptance Testing

The following testing procedures are Black-Box testing.

Test choosing 1v1 or vs AI. The current turn player should not be able to pick up the opponent's pieces. Check that for the first turn, only white is allowed to complete a movement.

If Playing against the AI and the AI has no legal move to make (but the AI still has pieces), the timer will run out and the player will win.

If there is a legal jump, test that you shouldn't be able to move any non-jumper pieces. Then for each piece, check that legal moves match the behaviour as defined in the previous section with isMovementLegal().

Returning to the menu will allow the player to switch between playing against another player or AI. While switching, the player may also choose which colour they want to play as.

7.3 Regression Testing

Run all the previous tests from before, all should pass, except for one: BoardLegalMovesTest_MidgameSetup(). Previously, forced jumps weren't implemented and were not accounted for. The test has been corrected and now passes again.