SE 3XA3: Test Report
KingMe

Team 9, KingMe
Ardhendu Barge 400066133
Dylan Smith 001314410
Thaneegan Chandrasekara 400022748

April 13, 2021

Contents

List of Tables

List of Figures

Date	Version	Notes
2021/04/11	1	Adding the functional and nonfuncitonal tests
2021/04/12	1	Adding the unit tests and automated testing
2021/04/12	1	Updating the functional tests

Table 1: **Revision History**

In this document we will cover the results of the testing procedures outlined in our Test Plan.

1 Functional Requirements Evaluation

1.1 Application Options

Homescreen Tests

1. FR-AO-1

   Initial State: Game is not launched.

   Input: Tester launches the game.

   Output: Tester is greeted with the homescreen, and provided the option to choose game mode and piece colour.

2. FR-AO-2

   Initial State: Game is on the homescreen.

   Input: None.

   Output: Application remained on the homescreen until the tester chose to begin a game.

New Game Tests

1. FR-AO-3

   Initial State: Game is is in progress.

   Input: Tester selects the new game button.

   Output: Application returned to the homescreen allowing user to begin a new game.

Game Mode Tests

1. FR-AO-4

   Initial State: Game is on the homescreen.

   Input: Tester selects 1-player mode and plays 1 game as red and 1 game as white.

Output: The application allowed the tester to make the first move when they chose red, and then the AI moved the white pieces. The AI made the first move when the tester selected the white pieces, and then waited for the tester to make a move.

2. FR-AO-5

   Initial State: Game is on the homescreen.

   Input: Tester selects 2-player mode and plays 1 game as red and 1 game as white.

   Output: In both cases the application only allows the tester to move the red pieces first, and then the turns alternate. The tester was in control of making moves for both colours.

Colour Choice Tests

1. FR-AO-6/7

   Initial State: Game is on the homescreen.

   Input: Tester selects the red/white pieces and plays a game in both 1-player and 2-player mode.

   Output: The board is displayed with the piece colour that the tester chose on the bottom of the board, regardless of the game mode chosen.

1.2 Gameplay

Valid Moves Tests

1. FR-GP-1/2

   Initial State: Game is in progress and tester has at least 1 king and 1 regular piece.

   Input: Tester selects one of their regular pieces on their turn and then selects one of their king pieces.

   Output: The piece the tester chose is highlighted along with all the valid moves that the piece can make according to the rules of checkers.

2. FR-GP-3

   Initial State: Game is in progress and tester has 1 piece that can move to the last row on the opposite side of the board.

   Input: Tester moves their piece to the last row.

   Output: The piece is displayed with a crown and can now move in any directions.

3. FR-GP-4

   Initial State: Game is in progress and the tester has one of their pieces selected.

   Input: Tester chose to move to one of the valid squares.

   Output: The application displayed the piece in the new square.

4. FR-GP-5

   Initial State: A game is in progress and the tester can capture an opposing piece.

   Input: Tester chose to capture the piece.

   Output: The application move the piece to the new square and removed the captured piece.

Invalid Move Tests

1. FR-GP-6

   Initial State: Game is in progress and tester has selected a piece.

   Input: Tester chose to move to a square that was not highlighted.

   Output: The board remains the same an the same and the selected piece becomes unhighlighted.

2. FR-GP-7

   Initial State: Game is in progress and it is the AI's turn to move a piece.

   Input: Tester attempts to select a piece.

   Output: The application did not allow the tester to chose a piece.

End of Game

1. FR-GP-8/9

   Initial State: Game is in progress and tester is 1 move from winning/losing.

   Input: Tester makes the winning/losing move.

   Output: The application displays the game over screen displaying who won and who lost.

2 Nonfunctional Requirements Evaluation

For several of the Non functional requirements testing was done by having testers use the application and complete a survey. The results of the surveys will be mentioned with the corresponding test results.

2.1 Look and Feel

1. NFR-LF-1

   Test Description: Testing that at least 90% of testers found the application visually appealing.

   Test Output: 100% of testers felt the appearance of the application was visually appealing.

2.2 Usability

1. NFR-UH-2

   Test Description: Testing that the tutorial explains how to use the application.

   Test Output: 100% of users said that they found the tutorial helpful when learning to use the application.

2.3 Performance

1. NFR-P-1

Test Description: Timing how long it takes for the application to reset the board when the user chooses start a new game.

Test Output: It took less than 3 seconda for the application to update the display after the tester chose to start a new game.

2. NFR-P-2

   Test Description: Timing how long it takes for the application to register a users move.

   Test Output: It took less than 1 second for the application to update the display after the tester made their move.

3. NFR-P-3

   Test Description: Timing how long it takes for the AI to move.

   Test Output: It took less than 1 second for the application to update the display with the AI move, after the tester made a move.

2.4 Operational and Environment

1. NFR-OE-1

   Test Description: Testing that the application does not require internet.

   Test Output: When tester disconnected from internet the application continued to work correctly.

2.5 Maintainability and Support

1. NFR-MS-1

   Test Description: Testing that the source code is documented.

   Test Output: Tester was able to generate documentation.

2. NFR-MS-3

   Test Description: Testing that the source code follows the pep8 format.

   Test Output: Tester did not significant discrepancies in the formatting.

2.6 etc.

3 Comparison to Existing Implementation

This section will not be appropriate for every project.

4 Unit Testing

We used unit testing to test the modules that were integral to our system. The board class stores the state of the game and contains several methods that are needed by other modules. We were able to used unit testing choosing different board states and checking the outcomes of the methods. The piece class is relied on by the board class, so we also used unit testing to test that the different methods worked correctly on different instances of a piece.

4.1 Piece Module

Testing Constructor

- Test Constructor
  Instantiate a piece with a row, column, color, and direction. Test to make sure the member variables were set correctly.

Testing Move Method

- Test Move()
  Move a piece to another row and column. Test to make sure the row and column member variables were updated correctly.

Testing MakeKing Method

- Test makeKing()
  Make a piece a king. Test that the isKing member variable was updated correctly.

4.2 Board Module

Testing Constructor

- Test board()
  Instantiate a board with a regular board or a flipped board. Test to make sure the turn is set to "RED".

Testing Set Board

- Test setBoard()
  Instantiate a board with a regular board and a flipped board. Test to make sure that red pieces array has been populated with the correct pieces. Test to make sure that white pieces array has been populated with the correct pieces. Test to make sure that board state array has been populated with the correct pieces.

Testing Reset Board

- Test resetBoard()
  Emulate the user choosing red and then reset the board. Emulate the user choosing white and then reset the board. Test to make sure that red pieces array has been populated with the correct pieces. Test to make sure that white pieces array has been populated with the correct pieces. Test to make sure that board state array has been populated with the correct pieces.

Testing Get Pieces

- Test getPieces()
  Call get pieces with both white and red and make sure that the pieces returned are in the correct locations.

Testing Change Turn

- Test changeTurn()
  Call changeTurn and make sure that the turn updates correctly given the current turn. Test for both white and red.

Testing Remove Piece

- Test remove()
  Test that the piece to remove is in the board state and is in its piece array. Then call remove. Test that the piece has been remove from both the board state and the piece array. Test for both red and white pieces

Testing Move Piece

- Test move()
  Test that the piece to move is in the board state. Then call move specifying the location to move the piece and the piece to remove to make that move. Test that the piece has been moved in the board state and that the captured pieces are removed.

  1. Test that a piece can move 1 space and does not need to capture any pieces.
  2. Test that a piece can skip opposing pieces.
  3. Test that when a piece moves into the last row that it is kinged.

Testing Get Valid Moves

- Test getValidMoves()
  Test in multiple scenarios by creating different board states. Test that the valid moves function generates the correct dictionary corresponding to all the possible moves that the piece given can make, according to the rules of checkers.

  1. Test edge cases to make sure that moves aren't generated that would move the piece off the board.
  2. Test a piece that is blocked, it should not have any valid moves.
  3. Test a piece that can capture an opposing piece, make sure the capture is in the valid moves and the piece to be removed is in the list of pieces to be removed.
  4. Test a king piece to make sure moves are generated in both directions.

5. Test a king piece to make sure it can capture pieces in both directions.
6. Test a piece that can capture multiple opposing pieces to make sure moves are generated correctly.

Testing Check Game End

- Test checkGameEnd()
  Test in multiple scenarios by creating different board states. Test that the function accurately determines the end of game and produces the correct winner.

  1. Test that the game ends when all white pieces are captured, and that the winner is set to red.
  2. Test that the game ends when all red pieces are captured, and that the winner is set to white.
  3. Test that if all the white pieces are blocked and can't move, and it is white's turn to move, the end of game is reached and red is declared the winner.
  4. Test that if all the red pieces are blocked and can't move, and it is red's turn to move, the end of game is reached and white is declared the winner.

Testing Evaluate Board

- Test evaluateBoard()
  Test in multiple scenarios by creating different board states. Test that the function accurately returns the score of the board.

  1. Test a board in which the score should be 0.
  2. Test a board in which the score should be positive.
  3. Test a board in which the score should be negative

5 Changes Due to Testing

5.1 Board Module

By performing the unit tests for the get valid moves function we were able to see that the moves were not being generated correctly for a king piece that could make a double jump in both directions. We were able to find the fault in the code and fix the error as a result.

5.2 Game Module

For the game module manual testing revealed that the check game end method was being invoked at the wrong time. It was working correctly in situations where one player captured all its opposing pieces, however if would not realize the game was over if one player no longer could make a move. After discovering this fault we were able to fix the error by checking the game status before each player moved their pieces.

6 Automated Testing

We used pytest to automate our unit tests. We also used pytest-cov to generate a report based on the coverage of our test cases. We used automated testing to check that any modifications to the application did not cause any undesired changes to the board and piece classes. These tests were run before merging any changes.

7 Trace to Requirements

Requirement ID	Requirement Description	Priority	Test Case ID
FR1	Home Screen	High	FR-AO-1 FR-AO-2
FR2	Allow user to choose piece colour	Low	FR-AO6
FR3	Allow only valid moves	High	FR-GP-6
FR4	Show valid moves	Medium	FR-GP-1 FR-GP-2
FR6	Display is updated after each move	High	FR-GP-4 FR-GP-3 FR-GP-5
FR7	Start new game	Medium	FR-AO-3 FR-AO-6
FR8	Win state	High	FR-GP-8
FR9	Loss state	High	FR-GP-9
FR10	Win/Loss state notification	Medium	FR-GP-8
FR11	Remove jumped pieces	High	FR-GP-5
FR12	User's turn	High	FR-GP-7
FR13	Red pieces have first turn	High	FR-AO-6
FR14	King pieces can move both ways	Medium	FR-GP-2
FR15	Piece Promotion	High	FR-GP-3
FR16	Highlight selected piece	Low	FR-GP-1
FR17	Different game modes	Low	FR-AO-4 FR-AO-5
NFR1	Look and feel of application	Low	NFR-LF-1
NFR2	New game (response time)	Medium	NFR-P-1
NFR3	User makes a move (response time)	High	NFR-P-2
NFR3	AI makes a move (response time)	High	NFR-P-3
NFR4	FPS of application	Low	NFR-P-4
NFR5	Documentation of code	Low	NFR-MS-1
NFR6	Modularized code	Medium	NFR-MS-2
NFR7	Coding style	Low	NFR-MS-3
NFR8	Game should not compromise security	High	NFR-SR-1
NFR5	Game should have a tutorial	High	NFR-UH-2
NFR11	Game does not need internet	High	NFR-OE-1

Table 2: **Requirement Traceability Matrix**

8 Trace to Modules

Test Cases	**Modules Tested**
FR-ApplicationOptions	Menu GUI
FR-GamePlay	Piece Board AI
NFR-LookAndFeel	GUI
NFR-UsabilityAndHumanity	GUI Game
NFR-Performance	GUI Board Minmax

Table 3: **Module Traceability Matrix**

9 Code Coverage Metrics

The unit tests written to test the board and piece class provide 100% statement coverage of the two mnodules. We used pytest to run the tests and pytest-cov to generate the coverage results. The following image results from running the test cases and generating the coverage report. To run the tests yourself navigate to the scr folder and run make tests. To produce the coverage report yourself navigate to src folder and run make coverage.

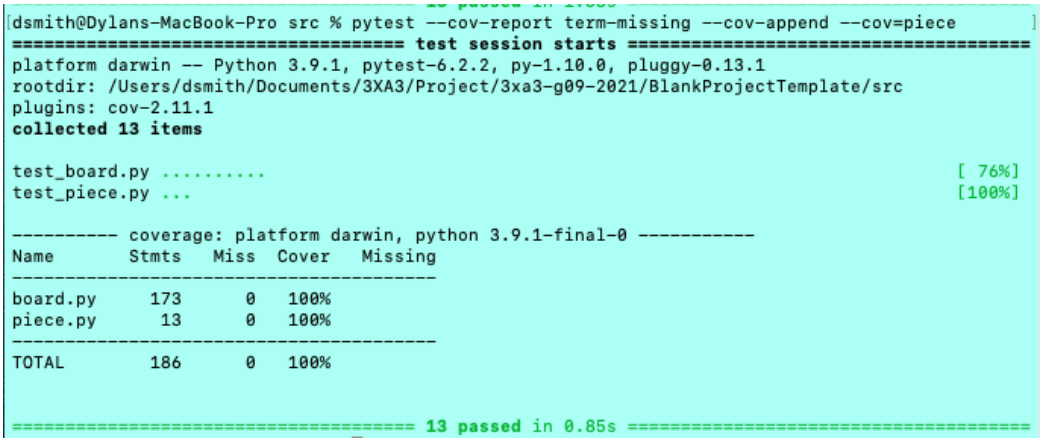

Figure 1: Code coverage