SE 3XA3: Test Report
Othello

Team #2, JSM Corporation
Jinesh Patel and patelj60
Mohammed Mirajkar and mirajkam
Sankar Renganathan and renganas

December 5, 2018

Contents

List of Tables

List of Figures

Date	Version	Notes
2018-12-05	1.0	Evaluations
2018-12-05	1.1	Test Results
2018-12-05	1.2	Comparisons

Table 1: **Revision History**

This document will go over the testing procedures and results that have been used and observed throughout the development process for Othello.

1 Functional Requirements Evaluation

1. Test Name: FR-UserValidity
   Results: All green cells representing the moves the user can make is valid according to the rules of Othello

2. Test Name: FR-AIValidity
   Results: All green cells representing the moves the AI can make is valid according to the rules of Othello

3. Test Name: FR-Invalid
   Results: Board state does not change when use makes invalid move

4. Test Name: FR-Save
   Results: Current state of board is saved in local storage after user clicks save button

5. Test Name: FR-Load
   Results: State of board in local storage is retrieved after user clicks load button

6. Test Name: FR-Undo
   Results: State of board is restored to the most recent state of the board on the history stack

7. Test Name: FR-Reset
   Results: State of board is 2 black and white pieces in the middle

8. Test Name: FR-Score
   Results: The white score corresponds to number of white pieces on board while black score corresponds to number of black pieces on board

9. Test Name: FR-Victory
   Results: Victory screen displays a win message if user has a higher score than AI, lose message if user has a lower score than AI, and a tie message if user has the same score as AI

2 Nonfunctional Requirements Evaluation

2.1 Usability

Method of Testing: A group of 8 people comprising of a group of 5 McMaster students from different faculties, 1 student in Grade 11, 1 student in Grade 6 and one student in Grade 1 were surveyed while playing JSM corporation's implementation of Othello

1. Test Name: NFR-Interface
   Results: All participants were able to navigate through the main screen and understand the functionalities of the button with an average of 2.5 minutes. The maximum time it took for an individual to get comfortable with the interface was 3 minutes and 10 seconds which is significantly lower than the expected time it would take to get used to the interface which was 5 minutes.

2. Test Name: NFR-Rules
   Results: All participants were able to understand the rules of the game within a minute which was lower than the expected time of 3 minutes.

2.2 Performance

1. Test Name: NFR-AITime
   Method: In-built timer is used to record the execution time for the AI to make a move
   Results: AI makes moves in under 1 second

2. Test Name: NFR-MoveTime
   Method: In-built timer is used to record the execution time for the board state to change after the user selects a move
   Results: User's selected move is displayed in board in under a second

3. Test Name: NFR-AnimTime
   Method: In-built timer is used to record the time it takes for an animation to complete
   Results: Animations/transitions displayed on interface are completed in under 2 seconds

2.3 Operations and Environment

1. Test Name: NFR-IO
   Method: Othello was played using a logitech mouse, Razer gaming mouse, and an HP Pavillion trackpad
   Results: Othello is compatible with all mouse types listed above

2. Test Name: NFR-Browser
   Method: Othello was played using FireFox, Chrome, Edge, Safari and Opera
   Results: Othello runs on all browsers listed above

2.4 Maintainability

1. Test Name: NFR-Git
   Method: McMaster software student was asked to navigate through git repository and find certain file
   Results: Student was able to find AI module under 10 seconds, Board module in under 10 and was able to identify the main purpose of the module in under 3 minutes

3 Comparison to Existing Implementation

This section will not be appropriate for every project.

4 Unit Testing

Unit Testing is performed on modules Board, History and AI as these are the fundamental parts of the software that don't have high integration and high dependency with the other modules. The strategy for unit testing was to test all the module functions that are avaialable to the external interface with as many possibilities of input to the functions that can test the greatest amount of equivalence classes. The Unit testing results can be found in the resulting generated file under the src folder, called 'test-report.html'. Coverage of the unit tests are found under Code Coverage Metrics.

4.1 History

4.1.1 Testing Constructor

1. Test default constructor.
    - Initial: No initial state.
    - Input: Create a history instance with no parameters.
    - Expected Output: The history instance has an empty timeline.
2. Test constructor with predefined previous state.
    - Initial: No initial state.
    - Input: Create a history instance with the predefined previous states.
    - Expected Output: The history instance timeline has the predefined previous states in the same order.

4.1.2 Testing canUndo()

1. Test canUndo.
    - Initial: A history instance filled with some previous states and another instance with no previous states.
    - Input: Call the canUndo function with no parameters.
    - Expected Output: `true` when there are previous states and `false` when there aren't.

4.1.3 Testing undo()

1. Test undo.
    - Initial: A history instance filled with some previous states.
    - Input: Call the undo function with no parameters.
    - Expected Output: The previous state of the object.
2. Test errors.
    - Initial: A history instance with no previous states.
    - Input: Call the undo function with no parameters.
    - Expected Output: The functions throws a `CannotUndo` error.

4.1.4 Testing reset()

1. Test reset.
    - Initial: A history instance filled with some previous states.
    - Input: Call the reset function with no parameters.
    - Expected Output: The history timeline becomes empty.

4.1.5 Testing push(data)

1. Test push.
    - Initial: A history instance with no previous states.
    - Input: Call the push function with states of an object.
    - Expected Output: The history timeline contains all those states in the order they were pushed.

4.1.6 Testing time()

1. Test time.
    - Initial: A history instance with a defined amount of previous states.
    - Input: Call the time function with no parameters.
    - Expected Output: The number of previous states.

4.1.7 Testing peek(i)

1. Test peek.

    - Initial: A history instance with some previous states.
    - Input: Call the peek function with an integer within the time amount of history.
    - Expected Output: The state of the object at the specified time.

4.1.8 Testing timeline()

1. Test timeline.

    - Initial: A history instance with some previous states.
    - Input: Call the timeline function with no parameters.
    - Expected Output: The previous states all in the correct order.

4.2 AI

4.2.1 Testing pick function for different difficulties

1. Test pick function to see if it acts the same as the sub pick function at difficulty 0.

    - Initial: An ai instance with difficulty set to 0.
    - Input: Call the pick function with no parameters.
    - Expected Output: A valid move and the pickMoveRandom function has been called with the return value of pick the same as the one from pickMoveRandom.

2. Test pick function to see if it acts the same as the sub pick function at difficulty 1.

    - Initial: An ai instance with difficulty set to 1.
    - Input: Call the pick function with no parameters.
    - Expected Output: A valid move and the pickMoveAverage function has been called with the return value of pick the same as the one from pickMoveAverage.

3. Test pick function to see if it acts the same as the sub pick function at difficulty 2.

    - Initial: An ai instance with difficulty set to 2.
    - Input: Call the pick function with no parameters.
    - Expected Output: A valid move and the pickMoveSmart function has been called with the return value of pick the same as the one from pickMoveSmart.

4.2.2 Testing pickMoveRandom

1. Test if it returns one of the moves given.
    - Initial: An ai instance with difficulty set to 0.
    - Input: Call the pickMoveRandom function with a set of possible moves.
    - Expected Output: The returned move is one of the possible moves.
2. Test function errors.
    - Initial: An ai instance with difficulty set to 0.
    - Input: Call the pickMoveRandom function with no moves.
    - Expected Output: Throws an error called `NoMoves`.

4.2.3 Testing pickMoveAverage

1. Test if it returns a valid move.
    - Initial: An ai instance with difficulty set to 1.
    - Input: Call the pickMoveAverage function with a board state.
    - Expected Output: The returned move is a valid move according to the board state.
2. Test the preferencing.
    - Initial: An ai instance with difficulty set to 1.
    - Input: Call the pickMoveAverage function with certain board states.
    - Expected Output: Returns the prefered move that was computed by the tester.
3. Test for no moves.
    - Initial: An ai instance with difficulty set to 1.
    - Input: Call the pickMoveAverage function with a board state that does not allow any moves.
    - Expected Output: Returns `null`.

4.2.4 Testing pickMoveSmart

1. Test if it returns a valid move.
    - Initial: An ai instance with difficulty set to 2.
    - Input: Call the pickMoveSmart function with a board state.

- Expected Output: The returned move is a valid move according to the board state and corresponds to the move returned by the minimax algorithm.

2. Test the algorithm.

   - Initial: An ai instance with difficulty set to 2.
   - Input: Call the pickMoveSmart function with certain board states.
   - Expected Output: returns a move that is computed by the minimax algorithm

3. Test for no moves.

   - Initial: An ai instance with difficulty set to 2.
   - Input: Call the pickMoveSmart function with a board state that does not allow any moves.
   - Expected Output: Returns `null`.

4.2.5 Testing getDifficulty

1. Test function.

   - Initial: An ai instance set to a certain difficulty.
   - Input: A call to the getDifficulty function.
   - Expected Output: The returned difficulty is the same as the set difficulty.

4.2.6 Testing setDifficulty

1. Test function.

   - Initial: An ai instance set to a certain difficulty.
   - Input: A call to the setDifficulty function with a different difficulty.
   - Expected Output: The difficulty of the ai instance is updated.

4.3 Board

4.3.1 Testing Constructor

1. Test default constructor.

   - Initial: None
   - Input: Size
   - Expected Output: Default 8x8 board

2. Test constructor with specific board state.

- Initial: None
- Input: Size and State of board
- Expected Output: 8x8 board with the predefined state

3. Test default constructor with invalid size.
    - Initial: None
    - Input: Invalid size
    - Expected Output: Exception for invalid size
4. Test constructor with invalid state.
    - Initial: None
    - Input: Invalid state
    - Expected Output: Exception for invalid state

4.3.2 getBoardSize Test

1. Test getBoardSize().
    - Initial: Current state of board.
    - Input: None
    - Expected Output: The current size of the board

4.3.3 isCellInBoard Test

1. Test isCellInBoard
    - Initial: No initial state.
    - Input: The i^{th} row and j^{th} column of the board
    - Expected Output: `true` if given coordinates are in the board, `false` if not

4.3.4 Testing isCellValue

1. Test isCellValue
    - Initial: Current board state
    - Input: The i^{th} row and j^{th} column of board as well as a cell value
    - Expected Output: `true` if cell value matches the value in the given cell coordinates, `false` otherwise
2. Test isCellValue Error

- Initial: Current board state
- Input: The i^{th} row and j^{th} column of board that is invalid as well as a cell value
- Expected Output: A thrown error indicating invalid cell location or value.

4.3.5 Testing Valid

1. Test Valid

   - Initial: Current board state
   - Input: The i^{th} row and j^{th} column of board as well as which play's turn it currently is
   - Expected Output: True if the cell is valid as well as the possible score for that cell

2. Test Valid Error

   - Initial: Current board state
   - Input: The i^{th} row and j^{th} column of board that contains an invalid value as well as which play's turn it currently is
   - Expected Output: A thrown error indicating invalid cell location or value.

4.3.6 Testing Move

1. Test Move

   - Initial: Current board state
   - Input: The i^{th} row and j^{th} column of board as well as which play's turn it currently is
   - Expected Output: None (Function does not return anything. Output is seen in UI.)

2. Test Move Error

   - Initial: Current board state
   - Input: The i^{th} row and j^{th} column of board that is invalid in terms of the rules as well as which player's turn it currently is
   - Expected Output: Error thrown indicating an invalid cell location or invalid move.

4.3.7 Testing getCellValue

1. Test setCell

    - Initial: Current board state
    - Input: The i^{th} row and j^{th} of board as well and the value of the piece in that position.
    - Expected Output: None (Function does not return anything as it is a mutator. Verification is done by getting the value of the boards property. In whch case, the state attribute of board must have value v at index(x,y).

2. Test setCell Error

    - Initial: Current board state
    - Input: The i^{th} row and j^{th} column of board that is invalid in terms of the board size
    - Expected Output: Exception thrown indicating an invalid value or invalid coordinates

4.3.8 Testing getCellValue

1. Test getCell

    - Initial: Current board state
    - Input: The i^{th} row and j^{th} column of board to get cell value from.
    - Expected Output: The value of the piece currently in that cell

2. Test getCell Error

    - Initial: Current board state
    - Input: The i^{th} row and j^{th} column of board that is invalid in terms of the board size
    - Expected Output: Exception thrown indicating invalid coordinates

4.3.9 Testing getCount

1. Test getCount

    - Initial: Current board state
    - Input: valid piece type/value
    - Expected Output: The number of pieces of value v on the board.

2. Test getCount Error

- Initial: Current board state
- Input: invalid piecetype/value
- Expected Output: Exception thrown indicating invalid value.

4.3.10 Testing setState

1. Test setState
    - Initial: Current board state
    - Input: any valid board state
    - Expected Output: None (Function does not return anything as it is a mutator). Verification is done by getting the value of the board objects state attirbute and comparing it to another board object with the same state.
2. Test setState Error
    - Initial: Current board state
    - Input: invalid board state
    - Expected Output: Exception thrown indicating come form of invalid board state.

4.3.11 Testing findValidMoves

1. Test findValidMoves
    - Initial: Current board state
    - Input: the player's piece color whose turn it currently is
    - Expected Output: All valid moves for the player whose turn it is currently .
2. Test findValidMoves Error
    - Initial: Current board state
    - Input: invalid player designator
    - Expected Output: Exception thrown indicating no such color for player exists.

4.3.12 Testing ignore

1. Test ignore
    - Initial: Current board state
    - Input: A callback function
    - Expected Output: The callback function should be called 0 times.

4.3.13 Testing notify

1. Test notify
    - Initial: Current board state
    - Input: Call notify
    - Expected Output: The set of callback functions has been called.

4.3.14 Testing listen

1. Test listen
    - Initial: Current board state
    - Input: A callback function.
    - Expected Output: The callback function should be called 1 or more times.

5 Changes Due to Testing

5.1 Board Module

The valid function was returing the wrong potential score which was changed such to return the correct amount.

5.2 AI Module

The minimax algorithm for the hard AI was changed to incorporate a region-based thinking heuristic that we use in our medium AI. The change was mandated after it was discovered during testing that the hard AI felt easier to beat compared to the medium AI. After incroporating the region-based algorithm, the feedback from testers all marked the AI as being much harder to beat.

5.3 History Module

No changes were made to the history module.

6 Automated Testing

Automated Testing is performed using Jest (requires Node.js and npm to be installed) in the Visual Studio Code IDE. The report is generated through Jest-HTML-Reporter plugin for Jest. Automated testing was best suited towards unit testing as stubs and drivers are very easily created for the particular modules and functions that were tested through unit testing. It was best to test the integrated software and UI elements with manual testing as

it was more time efficient to do so for the schedule of this project as writing the test cases for UI has a large learning curve. The results of the automated testing are given in the Unit Testing section. You can find the coverage report of these tests in the Code Coverage Metrics section.

To run automated tests, the best way is to use a Unix based terminal with the cloned repository and from the src directory run the command `./node_modules/jest/bin/jest.js --coverage`. This should output the testing results as well as the coverage in the terminal and also generate 'test-report.html' for the test results as well as 'index.html' under a generated folder named 'coverage/lcov-report'.

7 Trace to Requirements

Test	Requirements
Functional Requirements Testing	
FR-Victory	FR2, FR12
FR-UserValidity	FR5, FR6
FR-AIValidity	FR4, FR5
FR-Invalid	FR5, FR7
FR-Save	FR8, FR11
FR-Load	FR11, FR12
FR-Undo	FR10
FR-Score	FR5
FR-Reset	FR3, FR9
Non-functional Requirements Testing	
NFR-Interface	NF1, NF2, NF3, NF4, NF5,NF6, NF14
NFR-Rules	NF5, NF6
NFR-AITime	NF7
NFR-MoveTime	NF8
NFR-AnimTime	NF9
NFR-IO	NF10
NFR-Browser	NF11
NFR-Git	NF12, NF15

Table 2: Trace Between Tests and Requirements

8 Trace to Modules

Test	Requirements
FR-Victory	M3, M4
FR-UserValidity	M2, M4
FR-AIValidity	M2, M4, M6
FR-Invalid	M2, M4
FR-Save	M1,M5
FR-Load	M1,M5
FR-Undo	M1,M5
FR-Score	M1, M2
FR-Reset	M1, M5
Non-functional Requirements Testing	
NFR-Interface	M1
NFR-Rules	M1
NFR-AITime	M1, M6
NFR-MoveTime	M1
NFR-AnimTime	NF9
NFR-IO	N/A
NFR-Browser	N/A
NFR-Git	N/A

Table 3: Trace Between Tets and Modules

9 Code Coverage Metrics

Our group has made tests that cover 100% of all statements in the tested modules which consist of the AI class, the Board class, and the History class respectively. This number has been retrieved from the our tests using the Jest testing application. A picture has been provided below to show that our tests truly cover 100% of the tested modules. Additionally, one can also refer to the coverage document which is in the location 'Coverage/Lcov Report'.

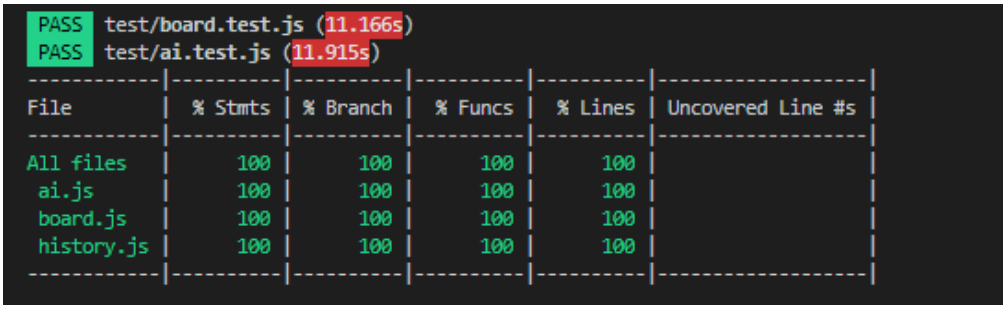

Figure 1: Code Coverage Metrics