Assignment 3 Testing

As a basic test for any obvious errors two computers were set up to play against each other in test code. 10 games were played with varying board setups and first turns. Both computers functioned as intended and were unable to find any errors, and both applied the strategy that they were supposed to.

Test Case: CompTurn

Expected Result: Computer makes appropriate move given the current board status using a hierarchal system.

Pass/Fail: Pass

Actual Result: Using a series of print statements it is determined that the computer is following the hierarchy, after each step only the appropriate move are remaining in the list of available moves. The stages of the hierarchy are listed below.

Test Case: removeBadMoves

Expected Result: Removes certain moves from the list of available moves based on which stage of the hierarchy it is currently on.

Pass/Fail: Pass.

Actual Result: No moves that violate the hierarchy are ever taken and the list of valid moves is updated after each check.

Test Case: movesFromBack

Expected Result: The computer will not move a piece from the back row, unless it has no other options.

Pass/Fail: Pass.

Actual Result: Given multiple different situations the computer will only move out of the back row if it has no other pieces to move or if it can only take a piece from the back row.

Test Case: moveEndsAt

Expected Result: Returns the ending location of a given move.

Pass/Fail: Pass.

Actual Result: The code will return the ending location as expected, however it only works for cases that can be played out realistically. The invalid moves are checked outside the method.

Test Case: movesAvailable

Expected Result: Contains a list of all available moves.

Pass/Fail: Pass.

Actual Result: The list holds only valid moves and will update when move are determined invalid using removeBadMoves.

Test Case: isMoveSafe

Expected Result: The computer should prioritize moves that do not cost it pieces whenever possible

Pass/Fail: Pass.

Actual Result: The computer will not move into range of the opposing players pieces if it allows itself to be captured by such a move, unless it has no available options.

Test Case: hasMorePieces

Expected result: Boolean to return true or false when the colour it is checking has more or less pieces respectively.

Pass/Fail: Pass.

Actual Result: Functions as intended, gives the proper Boolean.

Test Case: SpecialPCCP (piece can capture piece)

Expected Result: Returns a Boolean to determine if a given piece can capture another given piece. Modified from the original PieceCanCapturePiece in order to function as intended with isMoveSafe.*

Pass/Fail: Pass.

Actual Result: Completes the job of the original function while working with isMoveSafe.

*Note that while this is a revised PieceCanCapturePiece it cannot completely replace the old function, as they are still functionally different.

Black Box Testing:

Test Case: Requirement 1.1

Expected Result: Clicking standard setup configures the board in the standard way.

Pass/Fail: Pass

Actual Result: Functions exactly as intended, 12 pieces are setup. No pieces fall on white squares and they are spaced properly.

Test Case: Requirement 1.2

Expected Result: Click load game will load the previously stored state board.

Pass/Fail: Pass

Actual Result: Functions as intended.

Test Case: Requirement 1.3

Expected Result: Upon clicking two player game, two players can play against each other alternating turns.

Pass/Fail: Pass

Actual Result: Functions as intended, only legal move are allowed and the game will play according to the rules outlined in the previous requirements.

Test Case: Requirement 1.4

Expected Result: Upon clicking one player game the game will start and the computer will move for whatever colour it was chosen as.

Pass/Fail: Pass

Actual Result: Computer makes decisions based upon the methods outlined above and can play the game in full with no errors.

Test Case: Requirement 1.5 – standard move

Expected Result: The selected piece jumps one square diagonally, either left or right

Pass/Fail: Pass

Actual Result: Functions as intended, diagonally up left or up right are accepted as a white pawn and diagonally down left or down right are accepted as a black pawn.

Test Case: Requirement 1.5 – jump piece

Expected Result: The selected piece jumps over the second selected piece and in doing so lands one square behind it and the jumped piece is captured.

Pass/Fail: Pass

Actual Result: Functions as intended, pieces can only capture pieces of the opposite colour given they are positioned legally with no piece behind to block.

Test Case: Requirement 1.5 – convert pawn to king

Expected Result: Upon reaching row 8 as a white pawn or 1 as a black pawn the piece will become a king.

Pass/Fail: Pass.

Actual Result: The piece will be given a large K to denote being a king and can now be moved either forward or backwards.

Test Case: Requirement 1.5 – move kings forward and backwards

Expected Result: White kings can move down rows and black kings can move up rows.

Pass/Fail: Pass.

Actual Result: Functions as intended, kings can move in any of the 4 diagonals provided there is nothing to block the path.

Test Case: Requirement 1.6

Expected Result: Upon pressing the save button a .txt file is either created or updated to store the current board status.

Pass/Fail: Pass.

Actual Result: The .txt file will properly store the information and be capable of loading at a later date.

Test Case: Requirement 1.7

Expected Result: Upon either running out of pieces, having no available moves, or clicking the resign button a message will pop up indicating that the game has been won. The board will then return to custom setup.

Pass/Fail: Pass.

Actual Result: The proper message box pops up when the game ends and the board is returned to the setup stage.

Test Case: Requirement 1.8

Expected Result: User interface is highly intuitive and easy to understand the available functions.

Pass/Fail: Pass.

Actual Result: The interface has no hidden functions and every button is easy to recognize.