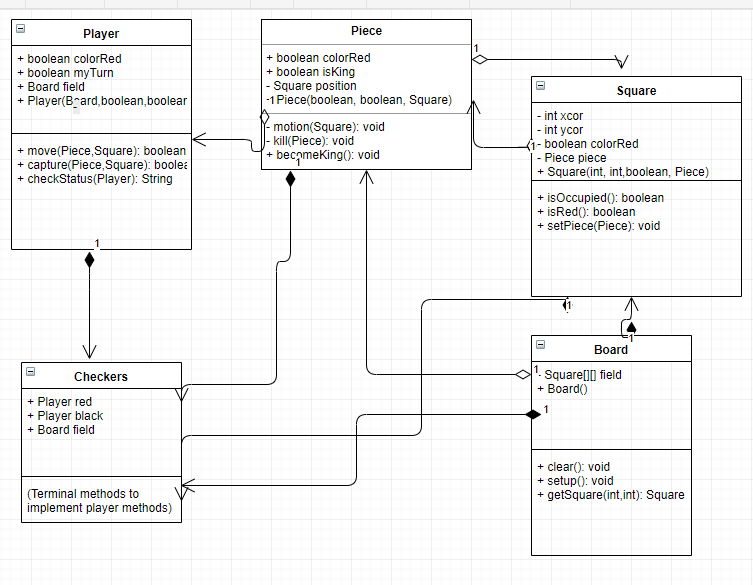
Project Name: Checkers

Project Description:

I plan to create a checkers game. It will have at least a feature for two players to play against each other, and I want an additional feature to play against a computer opponent (which uses fairly simple and basic strategy). Have a feature as a “hint” for which immediate next move captures the most pieces.

* Minimum Viable Product: A checkers game for two real life players to play against each other.
* Expected product: A checkers game for two players to play against each other. There will be two modes: player vs player and computer vs player. For both modes, there is an option to reveal the next immediate move that will capture the most pieces. The strategy that the computer will use against a player is going to use that strategy (it will choose the move that captures the most pieces each time)
* If I have extra time: A computer vs computer game mode, each player with the same skill as computer vs player.
  + I want to develop a computer with a more sophisticated strategy, such as maybe looking one move ahead. Perhaps, compare the immediate next move that captures the most pieces with the immediate next move after that for the player that captures the most pieces. Compare the amount of pieces captured for both moves. If the player will capture more pieces, then the computer will select a different immediate next move that captures the same number of pieces. The computer will again compare the move that captures the most pieces after that for the player. The computer will continue to do this, and then choose a move that captures the second most pieces, and continue choosing alternatives until it arrives upon a move where the number of pieces the computer will capture is greater than the number of pieces the player will capture on the next move. If a move like that can not be found, then the computer will select a random move that captures the same number of pieces as the computer.

UML Diagram:



Feature order to implement:

1. Player vs player game mode
2. “Move to capture most pieces” option
3. Player vs computer game mode that uses the move that captures most pieces strategy. We’ll call that “Difficulty 1”

If I have extra time:

1. Computer vs computer game mode both at Difficulty 1.
2. Player vs computer game mode where the computer selects a random move to make per move. We’ll call that “Difficulty 0”
3. Computer vs computer game mode with both computers at “Difficulty 0”
4. Player vs computer game mode when the computer figures out if the immediate move that captures the most pieces is actually the best move based on the player’s move after that. We’ll set that to “Difficulty 2”
5. Computer vs computer game mode both at Difficulty 2
6. A general Player vs computer game mode where you select the difficulty of the computer.
7. A general Computer vs computer game mode where you select the difficulty of both computers.
8. (somewhere in between any of the previous steps): implement a button to give a stronger hint, that implements the strategy used in Difficulty 2

How I plan to break up the work: I am working alone, so that is not necessary.

What features are dependent on other features:

* Square with coordinates depend on the Board setup() method working
* Move and capture methods of player depend on motion and kill methods of Piece working.
* Making the “move that captures the most pieces” will depend on the player vs player game mode to work.
* Making the computer vs player game mode with computer at Difficulty 1 depends on the code for the “move that captures the most pieces” strategy to work

Extra:

* Computer vs computer game mode depends on Difficulty 1 strategy to work
* Any computer modes that involve Difficulty 2 will depend on code for that strategy to work.

A Board will be a 2D-array of Squares, size 10 by 10. The Board() constructor will “clear”, giving each square a color, a pair of coordinates, and an “occupied” status. The clear() method will clear the board of pieces (set all the Squares’ “occupied” field to false). The setup() method will change certain Squares’ occupied statuses to occupied. It will also set up pieces’ positions to their respective squares, and will set up their isKing status and color status using the Piece(boolean, boolean, Square) constructor from Piece.

A Square has a color, a pair of coordinates, and a piece. It has no methods (or possibly some get methods, for its coordinates, or a boolean isOccupied() method). The Square(int, int, boolean, Piece) allows for Board to set squares, and allows Piece to control whether the Square is occupied or not. The Square’s color, and coordinates, may not be changed at any time. The Square’s coordinates are set when the board is cleared, and setup.

A Piece has a color, an “isKing” status, a position (which is a certain square). The constructor Piece(boolean, boolean, Square) allows for a set up of the piece.

The motion(Square target) takes in a square as an argument. It sets the current Square’s piece reference to null, sets the current piece’s Square reference to the target Square, and sets the target Square’s piece reference to current piece. The kill(Piece) method takes a square as parameter. It sets that piece to null.

The Player has a color status, a status of whether it is its turn or not, and a Board. The constructor Player(Board,boolean,boolean) sets up the player color. The methods move(Piece,Square) will have the player pick a piece and move it. There will be a conditional to check the isKing status to prevent the piece from moving backwards, and a conditional to check if the next square in its direction is occupied. If the piece is not King and is trying to move into an occupied square, then the method returns false. If the piece is null, or the target Square is null, then the method returns false.

The method capture(Piece,Square) works very similarly to the move(Piece,Square) method, but capture instead checks if the selected piece can move two steps forward. Optional rule in progress: allowing for multiple captures, or only double capture.