Final Project Report

Group 6

Chess for Two

Seung-hyun Francis Baek
Evan R Briones
Peter W Hall
Anthony Joseph Wurtz

Problem Statement

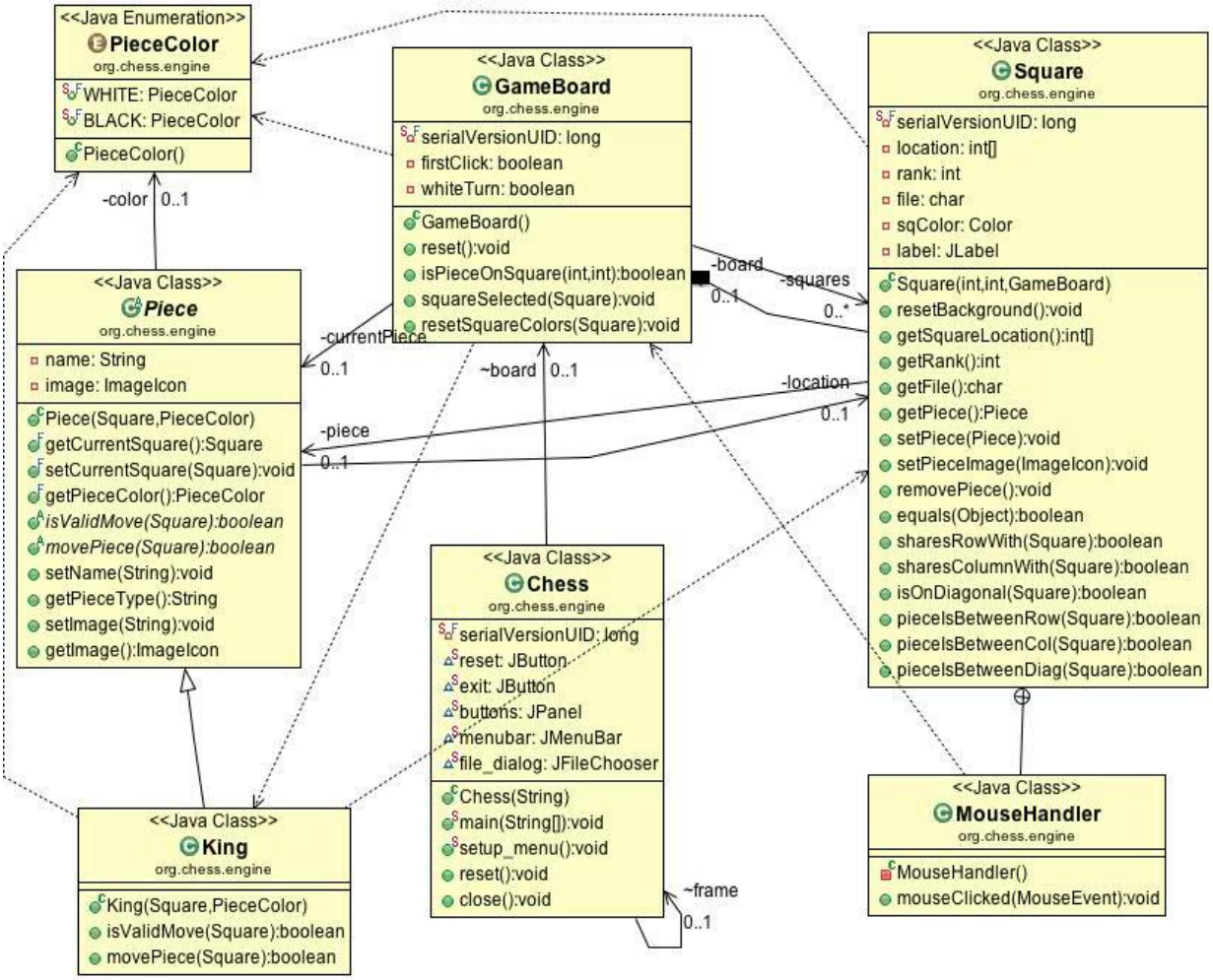
Chess for Two is a two player virtual chess board with valid move checking. The valid moves are based on typical chess rules.

System Boundary

Chess for Two incorporates necessary functions for two player local opponent chess. It will not feature functionality for remote players or AI at this time. Save and restore state was planned, but not implemented in the given time.

Domain Analysis

UML Diagram



Class GameBoard - Composed of Square objects of multiplicity 64: *has-a* relationship with Square.

Class Square - *has-a* relationship with Piece, which has multiplicity 0..1. Contains inner class MouseHandler for processing events generated by clicking a Square

Class Piece - Superclass generalizing all playing pieces.

Class King, etc - *is-a* relationship with Piece.

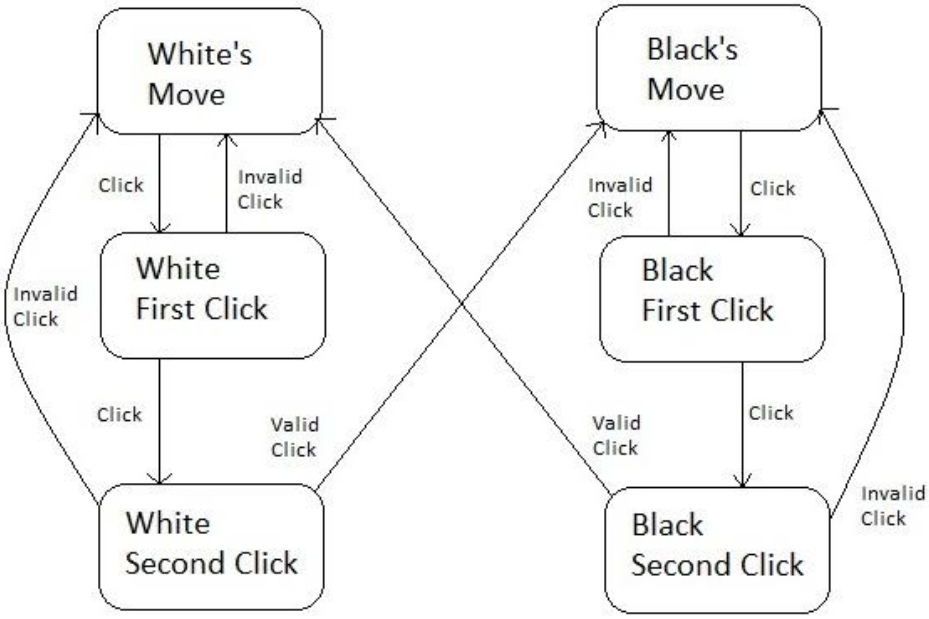
See Implementation for detailed analysis below.

Interaction Analysis

Program begins in the play state, ready for a game of chess with all parameters such as piece location reset to initial values.

Interaction with the program in play state consists of players taking alternating turns. Each turn consists of the player clicking a piece to move, followed by clicking a destination square. If the move is invalid, the piece is deselected, and the move is reset.

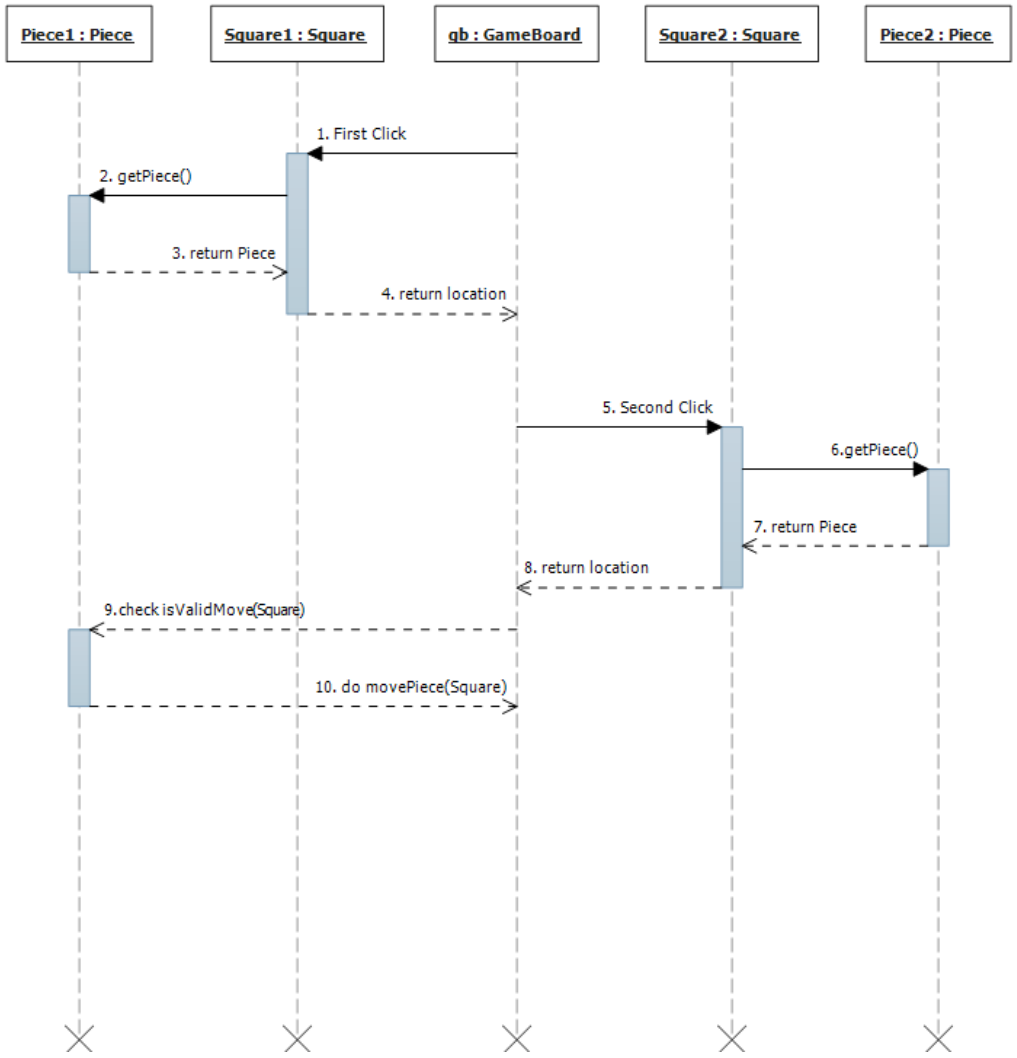
State Model



The static GameBoard object is the only class to undergo state changes. It implements player turns through a boolean variable *whiteTurn*, which switches polarity each time a valid move is made. Each state corresponding to white player's turn or black player's turn has it's

associated first-click and second-click states, for a total of six states. If the move is not valid, first-click or second-click return back to the current players move-state, but if second-click results in a valid move, then the state changes to the other player's move-state. Both first-click and second-click are implemented in GameBoard with the boolean variable *firstClick*, which is true on the first click, and false on the second click. The boolean *firstClick* resets to true for both a valid move, transitioning to the other players move-state, or an invalid second-click, returning to the current player's move-state.

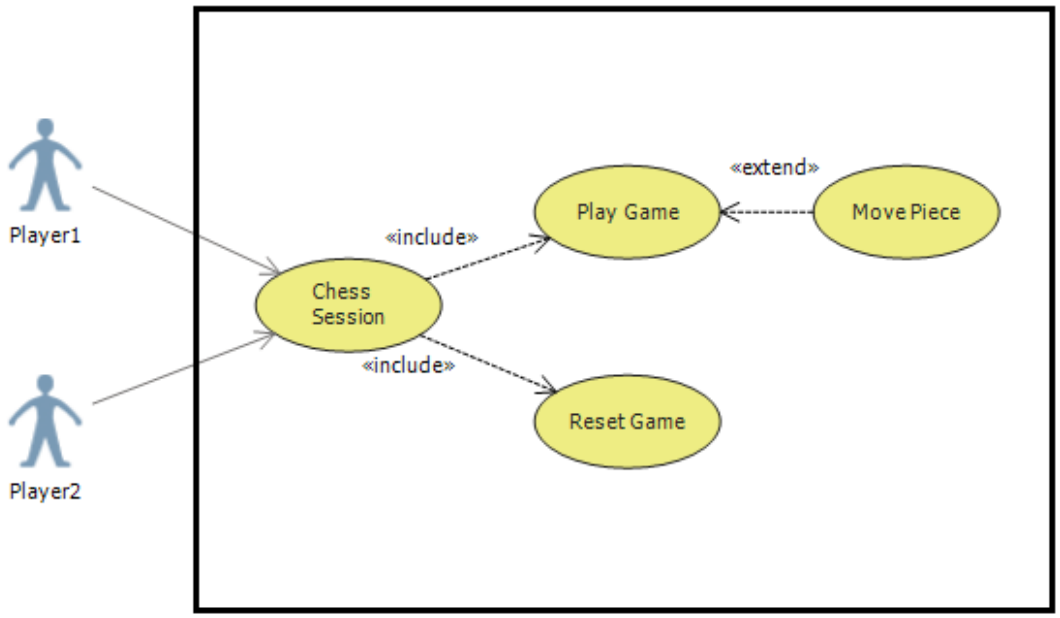
Sequence Diagram



The state diagram and sequence diagram show the method by which pieces move when players click pieces. The first click is to get data from the clicked Piece and its position on the board. The second click is to get the destination and Piece information from the destination square. If the clicks are correct and the movement is valid, the piece moves to the desired square on the board.

Use Case

Two users play the game at once, and when the application is started, it is ready to begin. The 'Chess Session' includes two cases 'Play Game' and 'Reset Game'. The 'Play Game' case, where the players alternate turns, includes 'Move Piece' where users click squares to move chess pieces. The 'Reset Game' case is invoked when the Reset button is clicked, or when 'New Game' is selected from the 'File' menu. This causes all Piece objects to be cleared from the GameBoard, and new Piece objects to be placed into initial positions on the GameBoard.



Implementation

Concepts of Object-Oriented design were utilized to aid in the implementation of *Chess for Two*.

Composition

The static GameBoard object is composed of sixty-four Square objects. These squares only exist as constituents of the GameBoard; hence they are members of the GameBoard in this strongly aggregated form. When the GameBoard is destructed, the Squares too necessarily will be freed from memory.

The squares determine their color based on position within the GameBoard, not from an external method call or constructor input. Hence, a simple loop is necessary to populate the GameBoard with squares, rather than sixty-four individual lines for Square object instantiation.

Inheritance

An abstract class Piece was used as the superclass for all playing pieces within the game. This allowed each Square to contain a Piece, as it is not possible to know what type of piece will go into a Square prior to runtime. Each playing piece inherits from Piece general properties shared by all playing pieces. All pieces extend Piece, and thus inherit member fields *PieceColor color*, *String name*, *ImageIcon image* and *Square location*. All pieces inherit *setCurrentSquare(Square)*, which is used to move the Piece to the destination square. The two abstract methods *isValidMove(Square)* and *movePiece(Square)* must be implemented by each subclass, as these behaviors are different for each piece type.

GUI Implementation

The GUI is implemented by making the GameBoard class a JPanel, which is added to a static JFrame in the main() method. This JFrame utilizes the BorderLayout layout manager, so that the GameBoard can take the CENTER region, and another panel with Reset and Exit buttons takes the SOUTH region.

The GameBoard utilizes the GridLayout layout manager, so that each Square can fill a position in the grid. Each Square initially extended the JTextField class, and contained the text of the piece type in the color of the piece, but in the current revision, each Square is a JPanel containing a JLabel displaying an image of the piece. The border for each Square is set to empty, giving a clean and sharp transition from Square to Square within the GameBoard grid.

The only class with an action listener is Square, so that each Square is clickable. Square contains an inner class MouseHandler which extends MouseAdapter, an abstract class which makes implementing the MouseListener interface easier, as all methods of MouseListener have been implemented with empty bodies in MouseAdapter. In our inner class MouseHander, the *mouseClicked(MouseEvent e)* method has been overridden with our custom actions when a Square is clicked. This class is then registered as a MouseListener with the *addMouseListener()* method of JPanel() in the constructor of Square.

Scope/Encapsulation

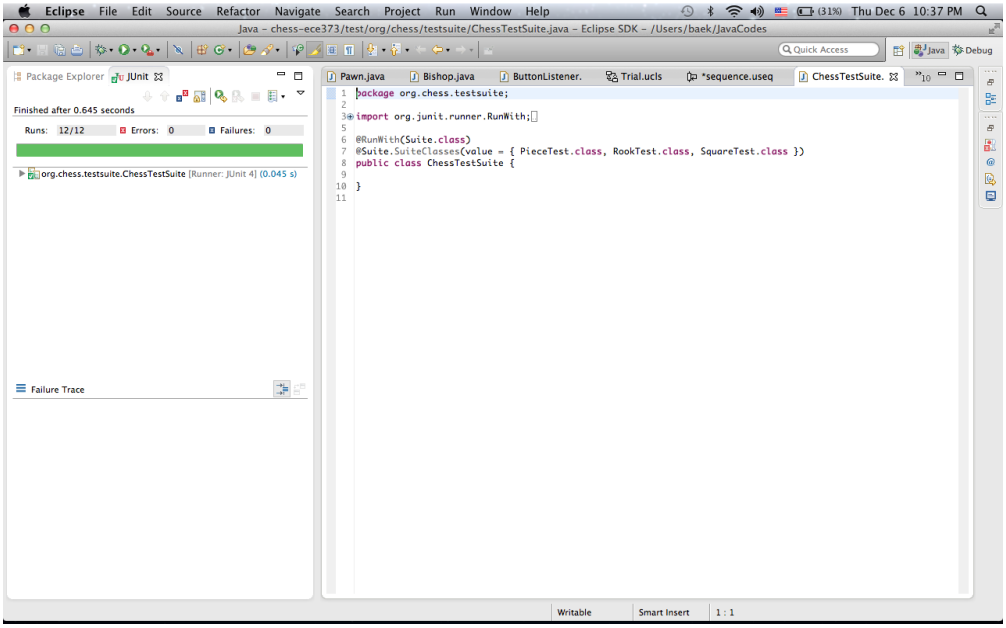
Each Piece knows where it is on the board, but only the GameBoard has a global view of where all pieces are. Thus, when a move is made, the selected Piece can only check if a destination Square is valid for that particular piece's type, as it is out of the scope for the Piece to check if there is another Piece in the destination square. The GameBoard does this global piece conflict checking. The Piece encapsulates the valid movement checking so that the GameBoard is free to only detect conflicts and captures.

Testing and Results

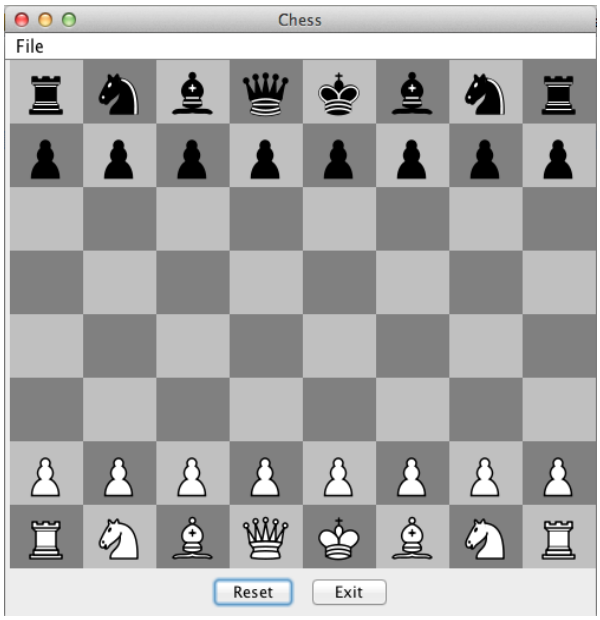
Some test classes were used to check the system. For example, the ChessTestSuite contains SquareTest, PieceTest, and RookTest which check the Square class, Piece class and Rook class, respectively. After the GUI was implemented, we did verification of the program by using the JFrame screen to determine whether pieces moved with the desired behaviors.

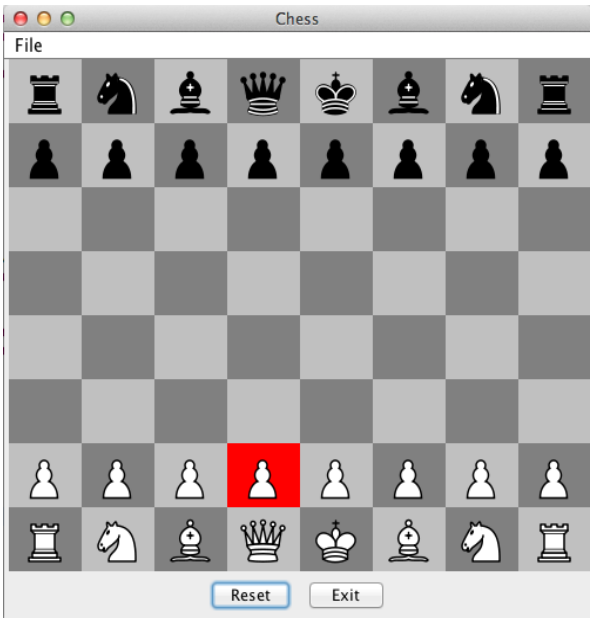
Screenshot of Testing

.



Screenshots of Results





Extensibility

Chess for Two currently does not cover special rules such as Castling, Promotion, or En Passant. We can add such special rules by implementing more methods in the Piece classes. For example, when a white pawn reaches black's edge, we can implement a method to check this condition and do the promotion.

Also, saving and loading could be implemented by writing to a file the current piece for all squares, including null for empty squares, and then populating all squares with the appropriate piece from this file.