Jessica Zheng, Nanxi Wang, Prachi Laud
Peck Period 2
APCS
27 May 2015

Checkers

This project provides code to play a fully functional 2 player game of checkers across different computers, with opponents being able to simultaneously play and chat with each other through a chat box in the same window as the checkers game.

Checkers is a game played between 2 players on a board of 64 squares, with each player starting the game with 12 pieces. Black plays first, after which players alternate turns moving their pieces forward. Pieces can only move diagonally one square unless the move is a jump; a jump occurs when a piece moves over an opponent's piece in a diagonally adjacent square. After a jump, the opponent's piece is removed from the board. Once a piece reaches the opposite side of the board it is made a "king," enabling it to move forwards and backwards and combine jumps in multiple directions. The game ends when one player cannot make a move.
In this version of Checkers, players cannot click on the board when it is not their turn, so that only one player is changing the board at a time. A message at the bottom of the screen indicates which player's turn it is, and all the pieces that could be moved by that player are highlighted in blue. Once the player clicks on the piece he/she would like to move, the possible squares to which that piece could move are highlighted in green. If a jump is possible, only the piece involved in the jump is highlighted.
In our implementation, players are also able to chat with each other through a chat box that appears on the side of the checkers board. Players type messages into the "outgoing" box, and they are displayed in the "incoming" box.

The Checkers, Board, CheckersData, and CheckersMove classes keep track of the current player, position of pieces on the board and the legal moves that a player can make.
The classes of the Checkers project that are inherited from SimpleChat enable the transfer of information between different computers through the use of sockets and ports. A socket represents a single connection between two network applications. It is one end-point of a two-way communication link between two programs running on the network, while ports allow software applications to share hardware resources without interfering with each other.

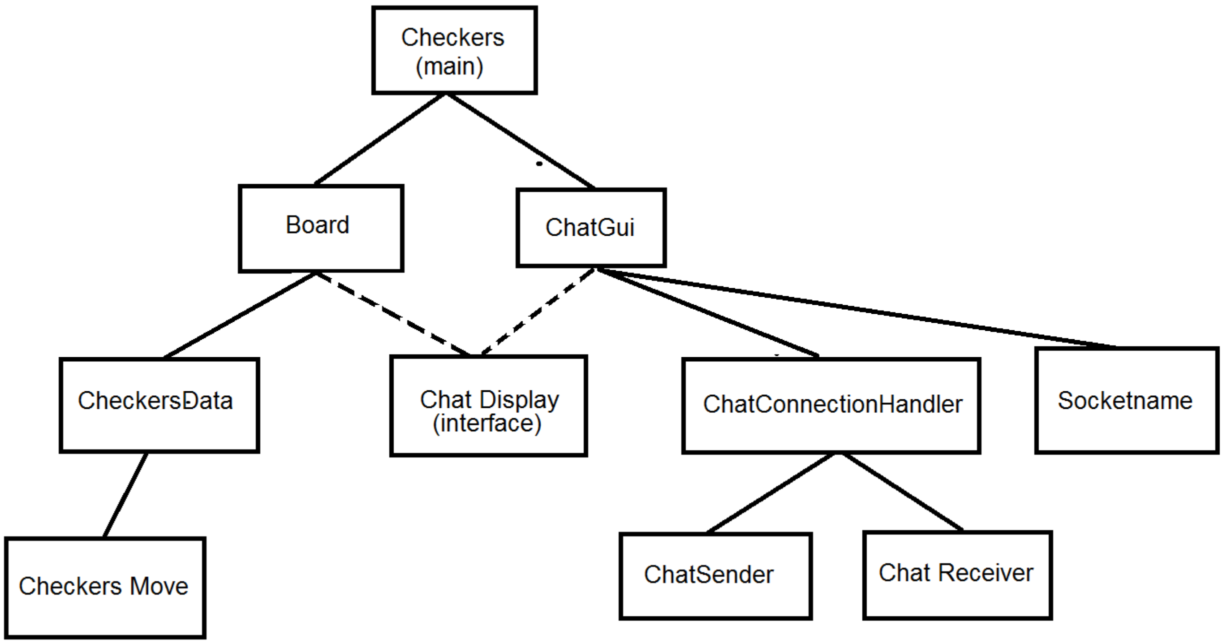
1. Structural design

We have chosen to hold all checker players and empty spaces in a 2D Array, which effectively represents the board. The int values in the 2D array signify which player is currently in a given space (Empty = 0, Red = 1, Red King = 2, Black = 3, Black King = 4). While a given entry in the array may change (according with the position of a checker position), the size of the board itself is constant, so it makes sense to use an array as opposed to an arraylist.

For the chat box program, everyone involved in the chat (senders and receivers) is stored in a Map, implemented as a Hashmap. The speedy access provided by Hashmap makes it the best data structure to hold the information about different players.

Data	Data Structure
board spaces	int array[][]
players (stored as SocketName objects)	Hashmap()

2. Object oriented design:



Our classes:

Checkers opens the JPanel, sets up the background and layout of the checkers board and chat box using JFrame's paintComponents and contains the nested class Board. Board references CheckersData to ascertain whether a player's move is legal before indicating the legal moves by highlighting the squares on the board. After a move is made it repaints the board and changes the message displayed to reflect the changes. The "new game" and "resign" buttons are defined within this class, as is the method "chatMessage" which receives messages from the other player and determines the appropriate action to take based on the content of the message.

CheckersData holds the location of pieces on the board in a 2D array and compiles a list of legal moves for each player when it is their turn. When a jump is possible, CheckersData only return a list of legal jumps to ensure that the player's move is a jump. After a move is made, CheckersData updates the placement of pieces on the board.

CheckersMove stores the information necessary for a piece to move from one square of the board to another: the current row and column, the row and column of the intended location and whether the move is a jump (if it is moving two squares diagonally adjacent). It also contains an equals method to check whether 2 moves are the same.

ChatConnectionHandler is in charge of listening for connections from and initiating connections to remote hosts. Because it must wait for new connections, the class is designed as a thread, with a main loop that simply blocks while waiting for new connections to arrive. The class also contains methods to initiate outgoing connections in response to commands from the main class. Once a socket has been created (either receiving or outgoing), this class attaches two new threads to it; one for reading data from the socket, and one for writing new data out to it. These threads are implemented using the ChatSender and ChatReceiver classes. This class has been used in our implementation for Checkers.

ChatDisplay is the interface defining the callback methods that must be supported for the socket threads to send data back to. In the default implementation, the ChatGui class implements this interface, and another implementing class is not necessary. This class has been used as is in our implementation for Checkers.

ChatGui creates the graphical user interface for the SimpleChat program. This class fulfills two important roles: first, it creates all of the GUI components necessary for entering text and options to connect to other hosts. Second, it acts as a displayer of messages received from the networking threads, and presents these messages to the user. In this case for the checkers game, the GUI components in ChatGui are not used.

ChatReceiver is a thread class that deals with receiving data from a remote host and relaying it back to the main chat program. You may assume that the socket is already connected in the constructor of this class. From there, your thread should wait for input to come in over the socket, read that data, and send it back to the display. If the socket is closed, the thread should terminate. The thread should also have a method that causes it to terminate, so the main program can kill the thread at any time. It should attempt to close the socket before exiting. This class has been used in our implementation for Checkers.

ChatSender is a thread class that deals with sending data from the program to a remote host. You may assume that the socket is already connected in the constructor of this class. From there, your thread should wait for input from the main program, and send that input out over the socket. If the socket is closed, the thread should terminate. The thread should also have a method that causes it to terminate, so the main program can kill the thread at any time. This class has been used in our implementation for Checkers.

SocketName encapsulates information about a socket connection so that it can be easily displayed and used as a primary key in a hash map. This class has been used as is in our implementation for Checkers.

3. Detailed design

The detailed specs for the Checkers classes have been generated from the Javadoc comments in the source files and are provided in the CheckersDocs.zip file. Open the index.html file to see the documentation.

4. Testing

All JUnit tests are stored in the JUtestCheckers class. Testing is primarily done by simulating various game situations, especially for the getLegalJumps(), getLegalJumpsFrom(), canJump(), and canMove() methods in the Checkers Data class. For example, the canJump() method is tested by setting up the board in a way that one piece is able to make a jump, calling the canJump() method for that piece, and asserting that the value returns true. Junit testing for GUI elements is done by asserting whether a given attribute of a GUI element (such as color, text, boundaries, font) matches an expected value or not.