2AA4 Checkers Design Documentation

Group 10

Table of Contents

Preface: Change Log

Software Evaluation Update: Additional Comments
- several structural changes are addressed

Handler: Updates 1-5
- implemented LoadGameClicked() method
- implemented SaveGameClicked() method (has 2 sections, one for if saveData exists, one if it doesn't)
- added "if event.getsource" is load, saveGame, save, cancel (for load game section)
- implemented LoadButtonClicked() method

Checkers: Updates 6-13
- implemented public void setLegalJumpsFor(int[][] board, int player, int row, int col) method
- implemented public void removeLegalJumpsFor(int,board[][], int row, int col) method
- implemented public boolean anyLegalJumps(int[][] board, int player) method
- implemented private boolean canJump(int[][] board, int player, int row, int col) method
- implemented public void jumpMove(int[][] board, int row, int col) method
- implemented public void regularMove(int[][] board, int row, int col) method
- implemented private void setDownJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing)
- implemented private void setUpJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing) method

Added "saveMenu" class
- opens textbox for game name when user clicks save while in game

Menu:
- added method JButton load()

main:
- added saveMenu instantiation
- added file to create saveData file when checkers first opened

LoadMenu:
- added methods including loadList, getSavedGames, setSavedGames, toMenu, getSelectedGame, setMessage

Notes:
- saveData.txt located in Checkers 1.5 (or whatever the folder ends up being called, in the current directory), outputs directory to show where data was saved
- first value is the name of the saved game, the rest is game data that is loaded/saved from the checkers board array

(Assignment 3 Changes)

Checkers Class:
Methods for AI

Menu Class:
Animation added
Sound Button added
Sound added

loadGame Class:
Bugs fixed

Gameboard Class:
Sound Button added
Resign Button added

Handler Class:
AI move added
AI comments added
Game Over added

Additional Note: Game rules in accordance with
http://boardgames.about.com/cs/checkersdraughts/ht/play_checkers.htm

Decomposition

The program has been decomposed into 4 classes, main, Gameboard, Menu, LoadMenu, saveMenu, Handler, and Checkers.

The **main Class** creates objects of the other 3 classes initially, and provides a proper connection between them.

The **Menu Class** contains all objects that are displayed on the Main Menu of the program, as well as several methods that are used by the Handler to access the Menu Class's private objects and global variables.

The **Gameboard Class** contains all objects that are displayed on the gameboard of the program, as well as several methods that are used by the Handler to access the Gameboard Class's private objects and global variables. In addition to this, the Gameboard is updated in it's own class. The reason for this is that by having the Gameboard update itself using primarily the 2D integer array that the Handler manipulates, the Handler does not have to use all of the Icons and Buttons - this task is better left to the class itself.

Public class **LoadMenu** extends JFrame is the menu that is shown when the user wants to load a previously saved game. The class contains a JList inside a JScrollPane so that users can scroll through the list when there are too many saved games. The user can select a specific saved game from the list and can either load or cancel/return to menu.

Public class **saveMenu** extends JFrame is the window that opens when “save” is clicked within the game to save the current game data. This contains a text box in which the game name can be specified and includes an “OK” button to specify when the name has been entered, “Cancel” to cancel the current action and to return to the current game, and error messages to let the user know if there is an error or the name has already been used as a previous saved game.

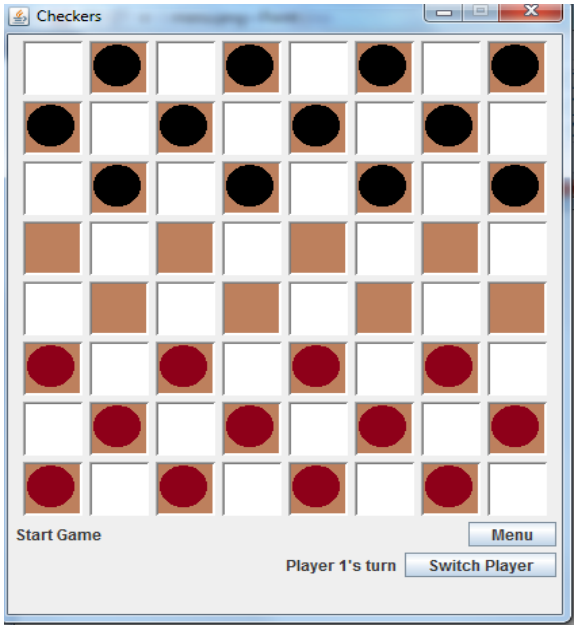
The **Checkers Class** contains all of the methods used to manipulate the 2D array model of the game. This part of the decomposition will be useful when creating AI because the AI will be able to operate using all the Checkers methods without interfering with the game itself.

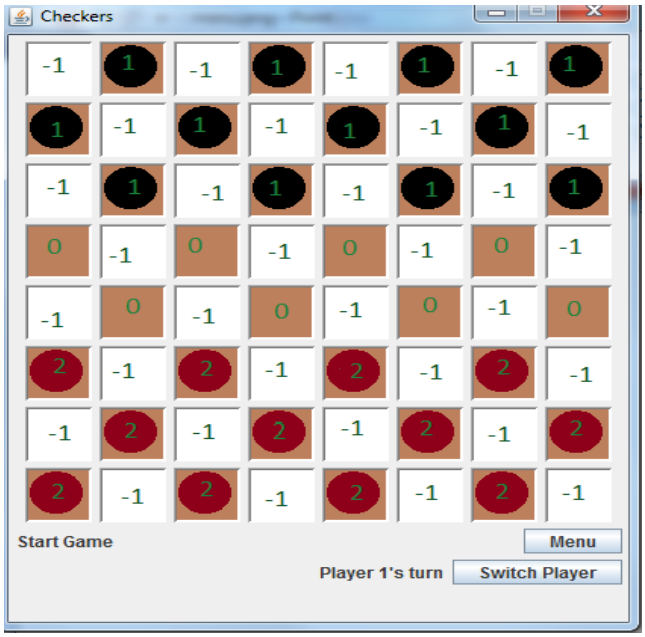
The **Handler Class** does most of the computational work. The Handler class extends the Checkers class so that it can make proper changes to the game. All events are handled by this class, so this class is given access to the Menu object created in main, and the Gameboard object created in main. Since these classes use private global variables and objects, the Handler Class uses these classes methods to manipulate their states. Since the Gameboard has a method to update itself, the Handler can simply modify the 2D array that represents the game in the Gameboard class when it needs to change the state of the game (e.g. when a play moves a piece). From there, the Gameboard class will update itself based on the 2D array that is contained in its class.

The Handler Class consists of a method for each swing object. When the object is used the corresponding method while run. This will make the class very organized and easy to edit, because the effects of using each button, checkbox, etc. will be restricted to a single method.

THE 2D ARRAY
-1 - This is a white square
0 - this is a brown square that is empty
Regular Pieces
1 - player 1 has an unselected piece here
2 - player 2 has an unselected piece here
3 - player 1 has a selected piece here
4 - player 2 has a selected piece here
Regarding Legals for Regular Piece
11 – player 1 can legally move a here with the regular selected piece
12 – player 2 can legally move here with the regular selected piece
13 – player 1 can legally jump here with the regular selected piece
14 – player 2 can legally jump here with the regular selected piece
Kings
21 – player 1 has a king on this square
22 – player 2 has a king on this square
23 – player 1 has a selected king on this square
24 – player 2 has a selected king on this square
Regarding Legals for King
31 – player 1 can legally move here with the selected king
32 - player 2 can legally move here with the selected king
33 – player 1 can legally jump here with the selected king
34 – player 2 can legally jump here with the selected king
100 – custom setup in progress, player can place square here freely

This system may seem complicated at first, but this system comes with several benefits. The first is that the Handler can control the state of the game without manipulating any of the Gameboard classes GUI components. This system also allows for more behind the scenes computation, as the program can make a number of changes to the array to test certain possibilities out (when implementing the AI for example) and the user will never see the result until the game.update() function is called. Also, games can easily be saved and opened using text documents. So long as the programmer understands the legend above that explains all of the numbers, the programmer does not need to know what is going on in the Gameboard class, and can still effectively program the entire game. The standard setup and its array interpretation is shown below.





where board[0][0]=-1 (top left corner), board[0][1]=1, board[5][0]=2, etc..

Public Interface

Menu

Objects:

This interface consists of one JButton and two JCheckBoxs. The JButton is called “start” and the JCheckBoxs are called “Single” and “Double” (that is, these are the names of the variables that hold them). All three of these objects are handled using the Handler class, which implements ActionListener. Because of this, the functionality of these entities will be explained in the explanation of the methods of the Handler Class below.

Methods:

There are three methods for this class, and they all exist only for encapsulation purposes.
The start() method returns the JButton “start”.
The Single() method returns the JCheckBox “Single”.
The Double() method returns the JCheckBox “Double”.

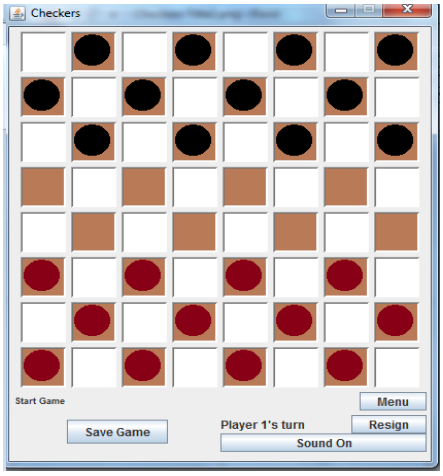
(UPDATE 2)

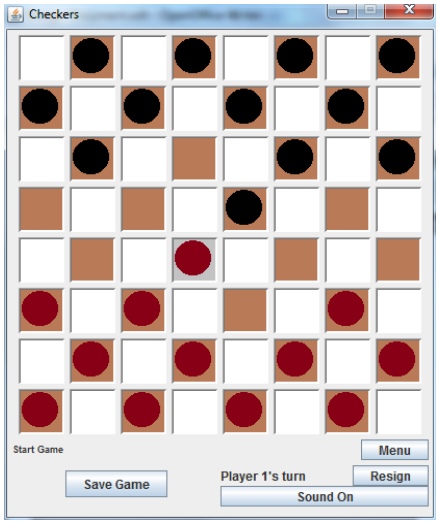
Animation: The animation works by using two images. The handler alternates between the two using a timer to create the animation.

Sound Button: The sound button is used to mute and unmute audio. The state in which the button is in will be the same as the corresponding button in the gameboard class. The text on the button is used to determine what pressing the button will do. If the button reads “Sound On” then pressing it will turn the sound on. If the button reads “Sound Off” then pressing the button will turn the sound off.

Sound: The theme song was created using an online 8-bit music making program. The .wav file is kept in the UI package and is played by default when the program is opened.

GameBoard





Objects:

This interface consists of 2 JLabels, 8 Icons, 3 traditional JButtons, 3 JPanels, and 64 beveled JButtons.

The 64 beveled JButtons compose the 64 pieces of the checkers board. They are each given an image icon and a rollover icon according to the assignment of the update() function, which is described in detail below. The JButtons are named btn00, btn01 …(etc), btn62, btn63, and they are then stored in a JButton array called "BTNS" that is used so that the buttons can be easily iterated through.

The 3 JPanels are used to hold the 2 JLabels and 3 traditional JButtons. The labels are named "message" and "lblplayer", and "message" corresponds in the picture (on left) to the text that says, "Start Game", while "lblplayer" corresponds to the text that says "Player 1's turn. "lblplayer" always states who's turn it is between player 1 and player 2. The Switch Player button switches from player 1 to 2 or from player 2 to 1. The Menu button will take the user back to the main menu, but the game will remain as it is.

Methods:

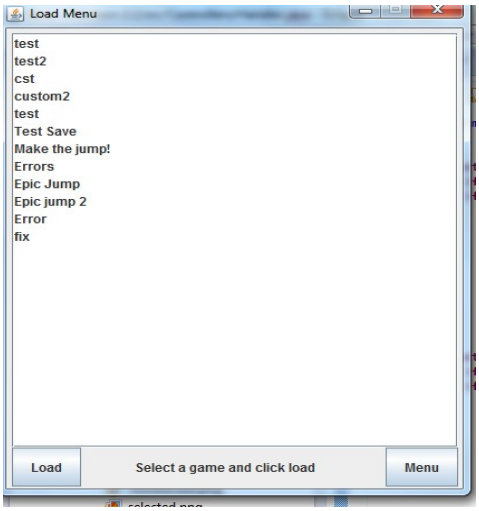
This class contains several methods for encapsulation purposes. Player() will tell us who's turn it is in the game. setPlayer() allows us to set the player whose turn it currently is. SwitchPlayer returns the JButton switchPlayer. toMenu() returns the JButton toMenu. Init() set up the initial state of the board(with piece), and customInit() set up the initial board without the piece. getColour() would return the state or the colour of the square. setupComplete() returns the JButton setupComplete. setlblPlayer() will show users whose turn at the moment. Board() will return the current board, and board(int board[][]) will set the input board to the current board).

Sound Button: The sound button is used to mute and unmute audio. The state in which the button is in will be the same as the corresponding button in the gameboard class. The text on the button is used to determine what pressing the button will do. If the button reads "Sound On" then pressing it will turn the sound on. If the button reads "Sound Off" then pressing the button will turn the sound off.

Resign Button: This button is actually the switchPlayer JButton, with the text changed. This is done because the button needs to be in the same location, and there is no need to waste more memory on a second button because the two are never used at the same time. Pressing this button will cause the active player to forfeit the game.

update() uses a single for loop that iterates through all 64 JButtons but also takes into account the coordinates that each button is located at in the corresponding 2D array that represents the state of the game. This is the method that connects the 2D array representation to what the user actually sees. The i-th JButton corresponds to row value i/8 and column value i%8. If board[row][col] is -1 in the array, the function will set that JButtons image icon to white and its rollover icon to white as well. However, if the value is 0 it will set the image icon to brown and the rollover icon to brown as well. If 1, then the icon is set to blackbrown, which is the image of a brown square with a black piece on it. If 2, then the icon is set to redbrown, which is the image of a brown square with a red piece on it. This system continues intuitively for all the values of the description of the 2D array found in the Decomposition document.

LoadMenu



Objects:

```
private JScrollPane scrollPane;
private JButton Load, toMenu;
private JPanel panelTop, panelBottom, panelWhole;
private JLabel instr;
private String[] savedGames;
private JList<String> list;
```

Methods:

As well as public encapsulations methods which return each of the objects individually.

Public JButton save() returns the JButton when “ok” has been clicked and the logic whether or not it is a valid name is determined and returned.

Public String getSaveName() returns a string of the game name the user wishes to save the game as.

Public void setNotification(String not) sets the message in the save menu to let the user know if there has been an error or the game name is invalid.

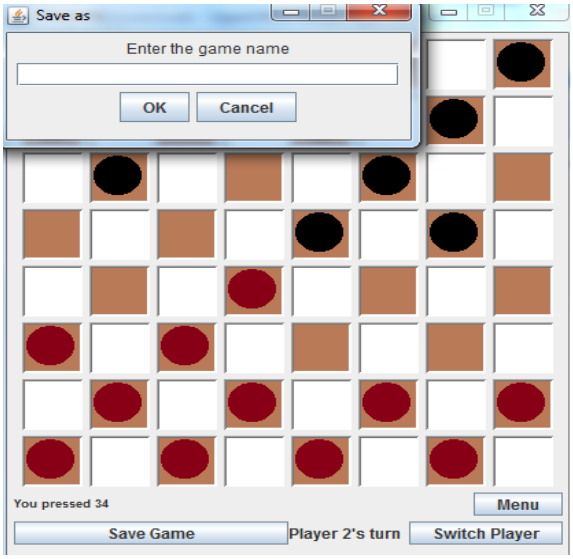
Public clearTextField() clears the text field so that the previous data entered into the text field does not appear.

Public JButton cancel() returns the JButton when “cancel” has been clicked, and runs appropriate logic to cancel the current action (in this case, saving the current game).

(UPDATE 2)
Bugs fixed

The bug in the menu was fixed by separating some of the methods and have the menu clear itself after every use, in order to avoid accidentally accumulating unwanted information.

SaveMenu



Objects:

```
private JTextField gameName;
private JButton ok;
private JButton cancel;
private JTextArea notification;
```

Methods:

As well as public encapsulations methods which return each of the objects individually.

Public JButton Load() returns Load to tell the handler that the user has clicked the load button. When the load button is selected, the logic is handled in the handler class.

Public void loadList() loads the saved games from the global string array. It writes the data to the list view for the user to see.

Public String[] getSavedGames() retrieves the data from the saveData.txt file.

Public void setSavedGames(String[] savedGames) sets the global string array to the parameter.

Public JButton toMenu() returns the user to the menu and hides the LoadMenu frame.

Public String getSelectedGame() reads which saved game in the list view has been selected and returns it.

Public void setMessage(String message) sets the message in the load menu for errors or for instructions.

Private Interface

The Handler Class

This class controls most of the functionality of the program. It uses the methods and objects of the main, Menu, and Gameboard class in order to create the proper 2D array representation of the game. The class implements ActionListener, so this is where the breakdown of the methods will begin.

private void actionPerformed(Action event) is the only method of the interface ActionListener. This method is activated every time the user interacts with a public entity. Contained within this method are several if statements that are activated when a specific public entity is interacted with, and the corresponding function is then called. The first thing that the actionPerformed method does is "refresh" the global variables, menu, game, board, and player. By doing this, all the methods in the class have access to the menu and game objects that were created in the main class, as well as a copy of the game board in the state it was at when they player clicked something, and the player that clicked it.

private void StartButtonClicked() first checks to see if a game is in progress. It does this by checking if the value of "setup" is 2, which means that a game is in progress. If the value of setup is not 2, then no game is in progress, and the method shows the user an option window that allows the user to indicate whether they want a regular or custom setup. A regular setup will use the init() function (described below) to setup a regular game, and a custom setup allows the user to place pieces of there own. The custom setup is run using the function CustomSetup(), which is only ever run if the value of global variables "blacks" or "reds" is greater than 0. This indicates that there are black or red pieces that have not yet been placed, and therefore the game has not started yet. For this reason, when the user choose setup custom, the StartButtonClicked() function will set blacks=12 and reds=12 so that the program knows there are pieces that need to be placed.

private void init() uses two for loops to create the standard setup for a checkers board using the 2D array system described in the Decomposition.

private void CustomSetup() is a variation of CheckersSquareClicked() (described below) because CustomSetup() indicates what clicking on a square does while there are pieces (reds or blacks) that still need to be placed before the game can begin. Every time a black pieces is placed in a valid position blacks=blacks-1 will run (same thing goes for reds). However, if the user clicks on an invalid square, no counter will be reduced, and instead the global variable "message", which is displayed on the gameboard, will tell the user that they have tried to place a piece on an illegal square.

private void CheckersSquareClicked(int board[][], int i) handles all of the logic behind the consequences of a user pressing a specific square given a specific game state. This method takes in the i-th button and then converts this into coordinates by using row=i/8 and col=i%8. The row and col variables represent the row and col of the square that was just clicked. From there, it activates one of several cases which will determine the change that is applied to the board.

private void setupCompleteClicked() will activate when the user clicks the JButton "setupComplete", which the user can use during custom setup to indicate that they are done setting up and that the game can begin. This will be used whenever 12 black pieces or 12 red pieces are not needed (for instance if the user is setting up a game partway through). For this reason, the method will set global variables reds=0 and blacks=0, which will cause the CustomSetup() method to no longer be called when squares are clicked.

private void SwitchPlayerClicked() will activate when the user clicks the JButton "switchPlayer". This method has two if-statements that will change the value of game.player to 1 (using encapsulation method game.setPlayer(1)) if the current value of player is 2, and likewise, it will change the value from 2 to 1 if the current player value is 2.

private void displaySetupMessage() sets the initial message the player sees.

(UPDATES 1-5 START HERE)

Private void LoadGameClicked() opens up a new frame where a list of games can be viewed and a specific game can be selected. That game data is then read and placed into the game board array and the checkers board is then updated and displayed. Inside this method, the saveData.txt file (which is the game's save data) is checked for existence. If it exists, then the user can select from the list view which game they would like to load. Each game is given a game name during the save process.

Private void SaveGameClicked() handles if "save" is clicked in the game window. If the file "saveData.txt" does not exist, it is created in the current directory. If it already exists, it continues on with the save procedure. During this "save" procedure, a text box is opened up in which the user can specify what they would like to name the game. The game name is the first element in the comma-separated line of data. After the game name is written to saveData.txt, the data of the array is appended to the same row of data in the file. This method also prints the location of the saveData.txt file.

Private void LoadButtonClicked() handles when the "load" button is clicked. It handles the actual loading of the data in saveData.txt. The method searches for the specified game name and loads that game's game board array, or data, through a buffered reader and then writes it to the game's array, and sets up the new game board.

(UPDATE 2)
AI move: The method aiMove will use inhereted methods from the Checkers class (namely chooseMove) in order to choose a move to make. Then it will select the initial square and move it to the final square using local functions. A more detailed description can be found on page 16 of this document.

AI comments: The method AIcomment uses a random number to select one of 31 cases, each containing several comments that the AI can make, depending on the given state of the game.

Game Over: The checkGameOver method is used after each move to see if the game is over. If the game is over, a sound clip is played according to who won, and the String message is updated with the results, namely "Player 1 wins!" or "Player 2 wins!", depending on who has more pieces when the game is done.

Checkers Class

private void getSelected() uses two for loops to iterate through the entire 2D array (refer to Decomposition document for details on the 2D array). The method will search the array for a value which corresponds to a selected piece, and then it will return the selected piece. In the picture to the left the values {4,3} would be returned.

private void setSelected(int row, int col) increases the value of the element board[row][col] by 2, signifying that it is selected, and then calling the method setLegalsFor(row,col).

private void setLegalsFor(int row, int col) will set all the legal squares that the element board[row][col] can legally move to.

private void Deselect() will get the selected element from the array, and then decrease its value by 2, signifying that it is no longer selected, and then call the method removeLegalsFor(row,col).

removeLegalsFor(int row, int col) will remove all the possible legal moves from the element board[row][col].

(UPDATES 6-13 START HERE)

public void setLegalJumpsFor(int[][] board, int player, int row, int col) will set the legal jumps for the square given by the coordinates in the parameters. Uses setUpJumps and setDownJumps. setUpJumps will only run if the piece belongs to player 2 or it is a king. setDownJumps will only run if the piece belongs to player 1 or it is a king. The method checks if there are empty positions 2 diagonals away from the selected piece. If an enemy piece exists between the two then a legal jump is set. If a legal jump exists, all other legal moves are removed then the legal jumps are placed on the board.

public void removeLegalJumpsFor(int,board[][], int row, int col) will remove any legal jumps that could be made by the piece given by the coordinates in the parameters.

public boolean anyLegalJumps(int[][] board, int player) will return true if any piece on the board (belonging to “player”) can make a jump. The method uses canJump method on each piece and returns true if any piece can jump, and false otherwise.

private boolean canJump(int[][] board, int player, int row, int col) will return true if the piece on the square given in the parameters can make any jumps. This method uses setLegalJumpsFor and removeLegalJumpsFor in a for loop. First it selects the given piece, then it sets the legals for the piece, and then searches the board for a jump, before remove legal jumps and deselecting the piece.

public void jumpMove(int[][] board, int row, int col) will handle the case where the selected piece is to move to the coordinates given in the parameters when this represents a jump. This involves removing the piece that is located between the selected piece's initial coordinates and final coordinates (given by row, col in the parameters).

public void regularMove(int[][] board, int row, int col) will handle the case where the selected piece is to move to the coordinates given in the parameters when this represents a regular move (from one square to an adjacent square).

private void setDownJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing)
sets the upward orientated jumps for the piece at board[row][col].

private void setUpJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing)
sets the downward orientated jumps for the piece at board[row][col].

(UPDATE 2)
Methods for AI:

public ArrayList<int[][]> getMoves(int[][] board, int player) iterates through the entire board, selecting each pieces that belongs to the player given in the parameters, and then checks to see if the piece can make any moves. If the piece can make a move, the move is added to an ArrayList by using a 2D int array as follows:

int[][] move = new int[2][2];
move[0][0] = The Initial Row Coordinate
move[0][1] = The Initial Column Coordinate
move[1][0] = The Final Row Coordinate
move[1][1] = The Initial Column Coordinate

Once every piece has been iterated through, the ArrayList containing all of the moves is returned.

public int[][] chooseMove(int[][] board, int player, int difficulty) first calls getMoves(board, player) in order to collect all the possible moves the player can make. Then, the method gets the ratio that each move will produce of player/enemy pieces using the getRatio method. Depending on the difficulty setting, the method will choose either the highest ratio or the lowest ratio. The difficulty setting Easy skips the ration process and chooses a move at random from the ArrayList using the method chooseRandom.

private int minIndex(ArrayList<Double> list) method iterates through the array list and returns the maximum value (which corresponds to a ratio).

private int[][] chooseRandom(int[][] board, int player) method chooses a random move from the ArrayList created by getMoves(board,player). First the method creates a random number between 0 and the list size. Then it returns the move located at the random number's index.

private int maxIndex(ArrayList<Double> list) method iterates through the list of ratios and returns the index of the list with the highest value ratio.

public int move(int[][] board, int player, int startRow, int startCol, int finishRow, int finishCol) method moves the piece at board[startRow][startCol] to the square board[finishRow][finishCol].

private int moveSelected(int[][] board, int player, int row, int col) method is similar to the method in the Handler class, only it does not have some of the calls that the Handler method makes to game and menu objects.

public double ratio(int[][] board, int player) method returns the ratio of player pieces to enemy pieces.

private double getRatio(int[][] board, int playerTurn, int player, int counter) is a recursive method that creates the recursion tree described on page 16. The base case is the method ratio(board,player) and the recursive step gets the average ratio of all the children using the getRatio method.

public int getNumPieces(int[][] board, int player) method sums the number of pieces on the board belonging to the player given by the parameters and then returns the value.

public int sumAllPieces(int[][] board) method adds the number of pieces of both players together and returns the value.

public int[][] copy(int[][] move) method iteratively copies each value from the given 2D array into a new 2D array which it then returns once the loop is complete.

protected static boolean isLegal(int i) method returns true if the given number correlates to some legal move value and false otherwise.

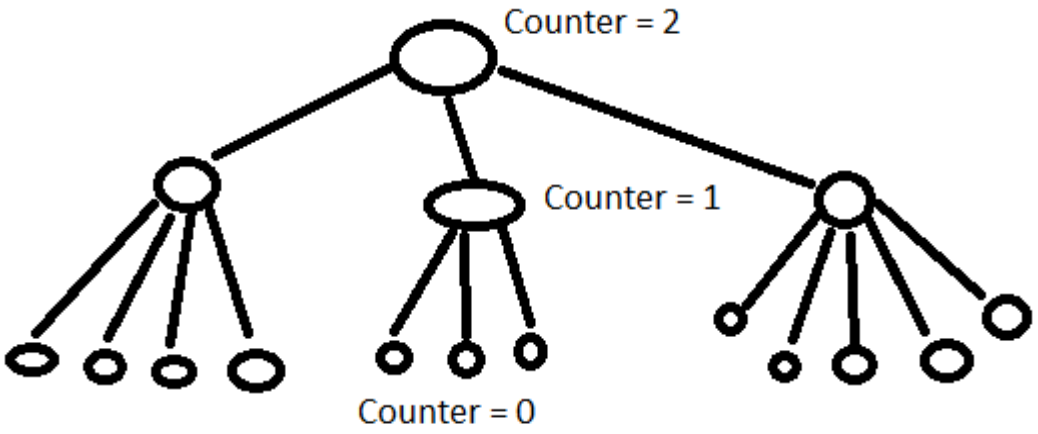
public boolean isPiece(int[][] board, int player, int row, int col) method returns true if the number at board[row][col] belongs to the player given in the parameters. Otherwise it returns false.

public boolean isLegal(int[][] board, int row, int col) method returns true if the number given by board[row][col] correlates to a legal move and false otherwise.

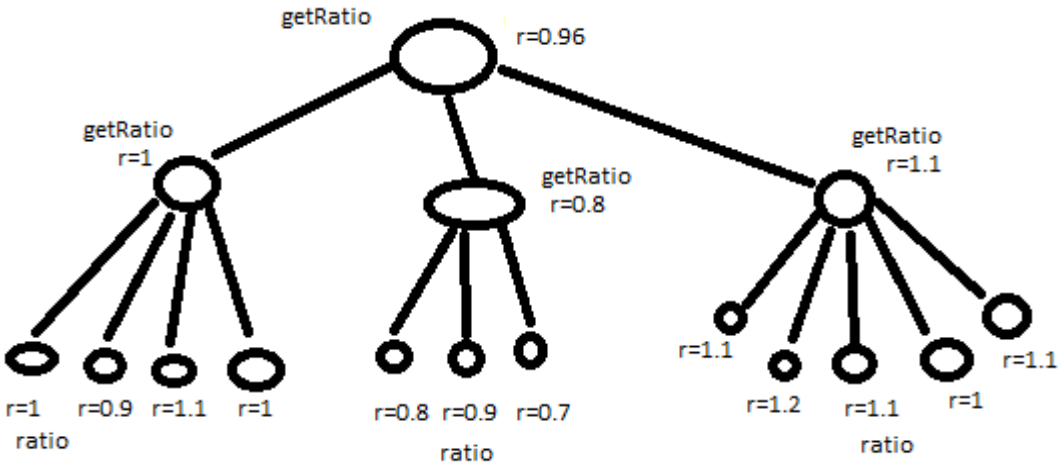
public void copy(int[][] board1, int[][] board2) method copies the values from board1 into board2, overwriting previous values in the process, using an iterative for-loop.

public boolean areEqual(int[][] board1, int[][] board2) method iteratively compares each value of the two arrays to each other to see if they are equal, provided their index's are identical (e.g. board1[i][j]==board2[i][j]). If, for all values, the two boards are equal, then the value true is returned. If any one comparison is false, then the method returns false.

AI Algorithm



The above tree represents how the AI calculates the ratio for a given move. The root of the tree represents the move that can be presently made. Each of the roots children represents a move that can be made after the root move is made. Each level down the tree the counter is reduced by one, until it reaches 0 which is when the base case is triggered. At the base case (the leaves of the tree) the parent of the leaves will average the ratios of all of the children. This ratio will then be used by its parent, along with the ratio of all the other children to that parent, to calculate it's average ratio. This process continue until the root is reached. In the diagram below, r=1 (for example) means that the ratio calculate at the given node is equal to 1, whether recursively (getRatio method) or not (ratio method).



The process will make a tree similar to the one above for each move it can make in the current game state, and then choose the one resulting in the highest ratio (Hard Mode and Medium Mode), or lowest ratio (Very Easy Mode). The difference between Hard Mode and Medium Mode is the height of the tree. In Hard Mode the tree is generally grown to a size of 8 initially, while the Medium tree is only grown to a size of 4 (ie. Counter=8 and Counter=4, respectively). The process can take a lot of time, so in order to speed up the process a method of tree trimming has been implemented, and is described as such: Whenever a given node sees that it has 7 or more children it will subtract 2 from the value of the counter so that each of the children will not grow as high, saving time at the expense of some accuracy.

Uses Relationship

The “uses” relationship is generally viewed as a hierarchical structure that depicts the relationships between unique modules. It shows the dependencies modules have on one another and define “how” the services are implemented or exported through interfaces. Hierarchies in the uses relationship organize the modular structure through something known as “levels of abstraction”, in which each level defines an abstract machine for the next level. Since levels are in a hierarchy, if k is the maximum level of all nodes of Mj such that Mi is relational to Mj, then there exists level k+1 in Mi. Hierarchies also make the software design easier to understand, thus making it easier to build and test.

For example, a module A that uses B means that module A requires the correct operation of module B, and module A can only provide its services through module B correctly exporting B’s services. Module A is able to access B’s services through its interface and is statically defined. In other words, A is defined as the client of B, and B is a server.

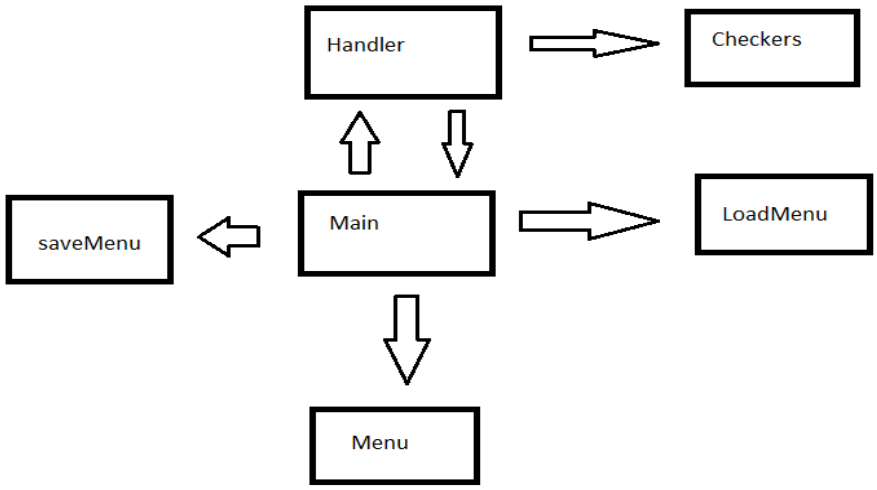
In the Checkers source code provided, “main” instantiates the “handler”, “menu”, and “gameboard” modules. Trivially, these are implemented in the “main” function because the Java Virtual Machine always looks for the “main” function first. The “handler” module would be considered the server, analogous to “B” in the example previously provided. In order for other modules to work, they require correct operation of the “handler”. Using the object-oriented paradigm, handler is instantiated as an object and its functions can be called through its interface. It can be seen in “main” that the menu, game, and handler are all statically defined since there will be only one instantiation of these objects. In order for the gameboard and menu to provide their services, handler must correctly export its services. B, or the handler, is known as a server (though only theoretically alike to the “server”, or web server, one would be more familiar with), since it provides the services “menu” and “gameboard” need to provide their own services.

As an example, the “gameboard” module displays the visual pieces onto the board. In order for the gameboard to move its pieces (a service provided by gameboard), the handler module must correctly manipulate the elements in the array representation of the gameboard. This provides the “how” in “gameboard”. The handler includes the “onClick” functions as well for the menu and gameboard which are essentially “actionListeners” to monitor when an interactive button is clicked.

Some other services provided by Gameboard are to update the board, initialize the board, initialize a custom game, switch players, display a message, go to menu, signal complete setup, etc. The “menu” has the service of providing options including single player, two player, regular game, or custom game, and these services are delivered through the services exported by the “handler” module.

Uses relationships are essential to the efficient organization and structuralizing of the software design. They aid in modularizing the design, as well as enabling low coupling and high cohesion.

(UPDATE)



These six boxes represent the six classes and the arrows represent the relationship between them. For example, Handler extends Checkers, so there is an arrow, but Checkers does not use any of the other class in any way. Main creates objects of Handler, LoadMenu, Menu, and saveMenu, and the Handler class accesses these objects in order to manipulate the game.

Traceability

The Gameboard accomplish the requirement of the 8 by 8 checkers board using the 2D array given to it by the Handle class. The requirement of the user being able to setup the initial positions is fulfilled by the method CustomSetup() in the Handler class, which activates when the user has chosen a custom setup from the Menu class. The ability to place a king is satisfied through the users ability to double click on a piece during setup to make it a king. This is implemented through the CustomSetup() method in Handler as well as the update() method in Gameboard. CustomSetup() will change the value of the pieces accordingly in the array when a piece that has already been clicked is clicked, and this change of value can be interpreted by the update() method to show the king on the game board.

The requirement that states users must be warned if a position is illegal during custom setup is satisfied by the JLabel message notifying the user whenever they click on a button that corresponds to a position in the 2D arrays which element is a value of -1 (signifying a white square). The requirement that there shall be a maximum of 12 pieces is implemented through the use of global variables “reds” and “blacks” in the handler class, which are decreased by one (from black if player one clicks, from red if player two clicks) each time the user clicks a valid square to place a piece. The requirement that states there must be a way for the user to signify that the setup is complete is satisfied through the JButton setupComplete, which when pressed, will set global variables reds=0 and blacks=0, and also cause it's own visibility to be set to false. Once “reds” and “blacks” are both zero the game will begin.

(UPDATE 2)
(Requirement #: Explanation)

1.1/1.2: Player can choose to start a new game using the Start Button, or they can load a previously stored game using the Load Game button, which uses the saveData text file that stores saved games.
(Refer to pages 7 and 10 for more details)

1.3: By selecting Two Players the user will deactivate the AI, allowing both moves to be made by human players.

1.4: The aiMove method uses a combination of methods from the Checkers class to determine which move the computer will make, depending on the users requested AI difficulty. (Detailed explanation on page 16)

1.5: setLegalJumpsFor, setLegalMovesFor, removeLegalJumpsFor, removeLegalMovesFor, moveSelected, jumpMove, and regularMove are all used to implement the correct movement of the pieces in the game.
(refer to pages 10 and 11 for more details)

1.6: The Save button can be used to save games, which uses the saveMenu class and saveData text file.
(refer to pages 7 and 10 for more details)

1.7: The switchPlayer button has been reused during game play as the Resign button, since switchPlayer is only a function needed during custom setup, which is a time when Resign is not needed.

1.8: The SquareClicked method notifies the user whenever they click an illegal square, when they click an empty square before selecting a piece, or when they select a piece that is not their own. This is done by updating the global string “message” which is displayed on the GUI gameboard after the actionPerformed method is complete (updating is the last thing the method does).

Software Evaluation

Problems Solved:

Since it is the first-phase development of the whole game, the program completed the following things to make it a base for the further developments for it:

1. The Chat Box appears when the program runs and gives three choices to users, "a single player", "two players" or simply directly starts the game.
2. Each of these three choices actually gives only two modes of display for the game board, "Regular" or "Custom", and they also appear as chat box when the user has made the first selection.
3. In "Regular" mode a 8 * 8 regular game board will be automatically set up with all checkers at the right spots. In "Custom" mode, an empty 8*8 checker board shall be given and the user can set the board by themselves, but it only allows the right checker at the correct spot.
4. After entering the game, a press button "Menu" allows the user to go back to the initial chat box, and the "switch player" button allows user to switch between player 1 and player 2.
5. If the user starts the game then go into menu and wants to come back to the game, he can choose whether to continue the game or start a new game though a new chat box.

The Criteria for the part I program and its scores:

For each applicable criterion, rate the program:

5 = Outstanding 4 = Good 3 = Satisfactory 2 = Poor 1 = Unsatisfactory

1. An 8-by-8 checkers board with light and brown squares, with the light square at the bottom right corner.

5

2. The user is able to set the initial position of pieces on the board at opening positions by clicking or using notation.

4

3. Users shall be warned if the position is illegal, and pieces cannot be placed on the illegal squares.

4

4. Maximum numbers of white and black pieces are both 12.

5

5. All the components are shown in a graphical interface.

5

6. Moves can be easily made.

5

7. Games can be easily setup.

4

8. The program is intuitive.

4

9. The AI is difficult to beat.

4

10. The AI is fast.

3.5

Additional Commentary on Design Decisions:

In choosing the 2D array design choice we also considered using classes for the checkers pieces. At first this seemed intuitive because when playing a game of checkers you might imagine the several properties that a piece has, such as the coordinates, or being a selected piece, or being a king'd piece. However, unlike a real checkers game, these pieces do not actually move from one place to another, it only appears that way to the user. Each square is merely a button, which is already an object, and it's icon must reflect the state of the game. For this reason, we decided it would be a simpler design if we used a 2D array to represent the current state of the game, in the same way that the icons merely represent the current state of the game. An advantage to doing this is that it is much easier to do computational work behind the scenes with arrays then with objects. If we used objects, and we wanted to let the computer try a move without showing it to the user, and see what the results were, we would have to change the properties of the existing pieces, and then change them back. Using the 2D array we can create many arrays independent of the game and only send the optimal outcome as the computers next move, for example. Also, it would be very difficult to save games using objects for checkers pieces, because we would have to implement a system of converting the state of the game into text. With the 2D array, this is already done, so saving games would be easy to implement.

(Updates)

Structural Changes

In accordance with constructive criticism given the program has been separated into packages. Also, many methods have been modified so that they work through the use of parameters rather than relying on global variables. Several methods have been made private where they are not publicly needed and the Handler class was divided into the Handler and Checkers classes in accordance with the request that UI components be more separate from game data representation. The new Checkers class contains absolutely no UI components, and is completely independent from all other modules.

Cohesion and Coupling

The cohesion and coupling are both relatively high in this design. Each class uses it's own functions primarily, but because the user needs to interact with the various User Interface components it is often required that the classes interact with each other. In order to try and reduce coupling the gameboard class is able to update itself, but then there is the problem of mixing UI components with game data representation. Although this is undesirable, the amount of coupling and encapsulation that would be required to have the Handler update all of the gameboard's icon images would be undesirable as well.

(Updates 2)

AI Algorithm Evaluation

The AI algorithm is decently capable of selecting a good move when the user has put the difficulty on hard mode. However, the algorithm can be slow, and this cannot be fixed by using dynamic programming or any other technique since every possibility is unique (ie. the resulted board from moving one piece is never the same as the resulted board from moving a different one). However, if multi-threading were to be implemented, some time could be saved, as the ratio to multiple moves could be calculated in parallel, since they are each independent of one another. Also, there is no way of knowing how many possible moves there will be after a given move is made. An example would be when you have to make a jump you might only have two moves, but after the jump you might have

seven for each way you could jump from. If you let the tree height of the recursion be large, because you see that there is only two moves, the resultant trees could still be very large, because each of the seven moves would take on the same counter value minus one, and if each of those seven moves has seven more moves, the time would become very noticeable. In order to prevent this, if a node sees it has seven or more children, it will reduce the counter which it uses to grow out each tree, and so reduces the time each one will take. Because this reduces the accuracy, other methods of increasing the run-time of the algorithm would be preferable.