Edinburgh Napier UNIVERSITY

Coursework Report

Svetlozar Georgiev
40203970@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Keywords – Checkers, Draughts, Java, Swing, Game

1 Introduction

The aim of this report is to describe the implementation of a Checkers game in Java, using the SWING framework[1]. The main features of the finished product are:

- A Graphical User Interface.
- Three different game modes which allow the user to play against another player or a Computer player, or to watch a game between 2 computer players.
- A replay system. Any finished game (or a game in progress) can be saved and later re-watched.
- Undo and Redo. A move made by a player can be reversed if they think they have made a mistake. Reversed moves can be redone.
- Visual features: the player can select a colour scheme for the game board. Furthermore, an arrow is displayed, showing the last move made.
- An information panel, allowing the player to see the number of checkers of each colour, the player whose turn it is currently, a history of all moves during a game and the current game's game mode.

2 Design

2.1 Software architecture

The Java programming language was selected for the implementation of this application because it is an Object Oriented Language and it provides all necessary data structures required to implement a Checkers game.

The MVC (Model-View-Controller)[2] architectural pattern was used in the development of the software in order to separate the game logic from the Graphical Interface, thus making code easier to understand and maintain. The three components of the pattern are:

- **The Model** - the fundamental behaviour and data of the application.
- **The View** - the User Interface of the application. It displays the data from the model in a suitable way for the GUI.
- **The Controller** - the link between the View and the Model. It uses the user input to make calls to objects and the view to perform appropriate actions.

2.2 Data structures

The main data structures used in the application are:

- **Arrays**

  The game board is represented by a two-dimensional array. The elements of such an array are arranged in rows and columns. This makes it very suitable because a Checkers board is an eight by eight grid of tiles. All the pieces are stored in the board array and accessing a specific one is simple if its position (row and column on the grid) are known. Furthermore, when checking for valid moves, having access to rows and columns, is a big advantage.

- **Array Lists**

  Array Lists have many advantages over normal arrays: they don't have a fixed size, it is easy to find out if they are empty and it is easier to check if they contain a certain element. Possible moves for a checker are stored in Array Lists. When validating a move, it is tested if the destination tile is contained in any of the Array Lists containing valid moves.

- **Linked Lists**

  Used in the replay system to store all the moves in a game. The reason Linked Lists were selected is because new elements are always appended to the end of the list and the order of elements is retained. Furthermore, elements can be retrieved or removed both from the head and the tail of the list. When the linked list is used to replay a game, elements are retrieved in order from its head (each one being removed after use). However, when used for the undo and redo functionality, elements are retrieved from the tail.

2.3 Algorithms

A number of algorithms were implemented in the application:

- **An algorithm to find all possible valid regular moves for a certain checker**

When a player clicks on a checker they would like to move, its type is checked and its position on the board is stored in the variables row and col. Depending on its colour, a checker can move only up or down. If the selected checker is black, the applications checks if the tiles at row+1, col-1 (which is the previous row and the left column) and row+1, col+1 (which is the previous row and the right column) are empty. For white checkers, the tiles checked are row-1, col-1 (which is the next row and the column to the left) and row-1, col+1 (which is the next row and the column to the right). If any of these tiles are empty, their position is added to an Array List which is later used to validate a player's move.

- **An algorithm to find all possible valid jumps for a certain checker**

  Similar to the algorithm for regular moves, the selected checker's position is stored in the variables row and col. If the checker's colour is black, the tiles with position row-2, col-2 (which is 2 rows above and 2 columns to the left) and row-2, col+2 (2 rows above and 2 columns to the right) are empty. For white checkers, the tiles with position row+2, col-2 (2 rows below, 2 columns to the left) and row+2, col+2 (2 rows below, 2 columns to the right) have to be empty. However, in order for a jump to be valid, there has to be an enemy piece to jump over. Therefore, it is checked if the tiles diagonally to the left and diagonally to the right are occupied by a checker of the opposite type. If a valid jump is then found, it is added to an Array List of all possible jumps for a checker.

- **An algorithm to generate the AI player's moves**

  This algorithm utilises both algorithms mentioned above. An Array List of all the checkers of the AI player's colour which can jump is created. If it is not empty, a random piece from it is selected. Finally, a random jumping position for it is selected from a list of its possible jump positions. Checking for available jumps first is done in order to make sure the AI always jumps if it's able to (which is what Checkers rules state). If none of the AI player's checkers can jump, however, an Array List containing all its pieces which have valid regular moves is created. A random checker from it is then selected. An Array List for all its valid move positions is created. Finally, a random position from it is selected. If there are no valid jumps and no valid regular moves, then the AI player must have lost the game.

- **An algorithm to validate a move**

  The positions of the checker which is to be moved is stored in the variables row and col and the destination tile's position is stored in the variables destRow, destCol. Firstly, it is ensured that the player is trying to move their own checker. Then it is checked if an Array List, containing all of the player's checkers which can jump, is empty. If it is not, it is checked if the list contains the piece they are trying to move. This ensures the player always moves a checker that can jump. If they are trying to jump, it is checked if the position destRow, destCol is contained in the Array List of valid jump positions. If a jump is valid and the checker can jump again, the player is allowed to make another move. If they are making a regular move, it is checked if the Array List storing valid move positions contains the position destRow, destCol. Finally, if the move is not valid, an error message is displayed.

3 Enhancements

Although the project has all necessary features listed in the specifications, there is still room for improvement.

Firstly, the AI player's moves are completely random, which makes it a very easy opponent. While this is a good solution if the users are beginners, a feature to select the AI difficulty could be a great addition.

Furthermore, implementing an algorithm which finds the best move for the AI player depending on the board state could be beneficial.

Allowing the user to select the colour of their checkers is another desired feature.

Another improvement would be the addition of a score system which would make the game more competetive.

The addition of sounds and animations would make the game feel more interactive.

4 Critical evaluation

In the finished application all features work to a satisfactory extent.

Firstly, the replay system allows the user to save any number of replays. A full game can be saved as well as a game in progress. What is more, the user has the ability to choose the name for the replay, meaning they can easily find the desired replay later.

Secondly, the undo feature allows a player to undo as many moves as they see fit. However, the redo option can only redo the last undone move. Usually, software which has undo and redo options, allows the user to undo and redo any number of actions.

Thirdly, the player is allowed to play against an AI opponent. While this feature works well, it is completely random. It can be improved as described in the Enhancements section.

5 Personal evaluation

Although all desired features were implemented, a number of problems were encountered during the development of the project.

Learning how to create an application utilising the SWING GUI framework proved to be a challenge. While there are many resources[3][4] on the Internet, it is difficult to find current tutorials which teach the best practices. What is more, SWING is not thread-safe[5], meaning any GUI code has to be called in the Event Dispatch Thread. Due to this, a problem where the whole application would crash randomly was encountered. The solution was simple: creating the main window had to be done in a SwingUtilities invokeLater call. The result of this was that any GUI code run from the game window would be executed in the SWING thread rather than the game thread. Furthermore, unlike C#, where there is a visual GUI designer, SWING interfaces have to be created by hand. This means creating the right layout and positioning all components correctly is much more difficult and sometimes was achieved by trial and error.

Another problem was found when implementing the replay system. The index of the last move added to the replay Linked List would be stored. When a move was undone, the index would be decremented. On the other hand, when a move was redone, the index would be incremented, thus keeping track of the position of the current last move. However, while undo and redo worked, the replay system had problems because moves which were undone were still stored in the list. This problem was fixed by redesigning the whole system. Instead of using an index, when a move is undone, it is saved and removed from the tail of list. This move then becomes the redo move (which just has to be reversed). Because of this change, however, a player is only allowed to redo the last move they have undone.

Choosing the correct architecture for the application code was a difficult task as well. The most obvious solution was using the MVC pattern as it is intended for software with a graphical interface. Combining game logic and GUI code would have definitely resulted in code which is hard to read and maintain. However, separating the code proved to be a good decision as it allowed for easier and faster development.

Ultimately, all problems were fixed and the application works as intended.

References

[1] "Swing framework." https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html. Accessed: 20/10/2017.

[2] "Codinghorror." https://blog.codinghorror.com/understanding-model-view-controller/. Accessed: 27/10/2017.

[3] "Tutorialspoint." https://www.tutorialspoint.com/swing/. Accessed: 22/10/2017.

[4] "Stackoverflow." https://stackoverflow.com/questions/tagged/swing. Accessed: 20/10/2017.

[5] "Adtmag." https://adtmag.com/articles/2000/08/25/swing-and-multithreading.aspx. Accessed: 29/10/2017.