Shallow Red
Final Design Document

Soroush Vosoughi
Dhanvi Reddy
Sam Dyar

December 10th, 2007
Table of Contents

Requirements........................................................................................................................................2
  Overview........................................................................................................................................2
  Revised Specifications.....................................................................................................................3
  Revised Specifications due to Amendment.......................................................................................4
  User Manuals (In Appendix A, B)..................................................................................................4
  Performance..................................................................................................................................5
  Problem Analysis...........................................................................................................................6

Design ..............................................................................................................................................8
  Overview..................................................................................................................................8
  Class Overviews and Runtime Structure.....................................................................................15
  Module Structure.........................................................................................................................20

Testing .........................................................................................................................................23
  Strategy...................................................................................................................................23
  Test Results.................................................................................................................................24

Reflection......................................................................................................................................26

Appendices ....................................................................................................................................28
  Appendix A – TextUI User Manual............................................................................................28
  Appendix B – GraphicUI User Manual.......................................................................................30
  Appendix C – Specification/Implementation Level Comments.................................................44
  Appendix C – Module Dependency Diagram..............................................................................45
  Appendix E – GUI Additional Features.......................................................................................46
Requirements

Overview

*Shallow Red* is a system for playing 6.170 *Antichess*, a variant of chess in which the goal is to either lose all of your pieces (except the king) or checkmate your opponent. In this document, we attempt to analyze the problem at hand, our approach to the solution and a presentation of the end package that resulted after six weeks.

Our program allows users to play the Antichess game through either a graphical interface or a textual interface. It also provides a framework that can be used to implement a turn-based game through an arbiter (in this case, antichess.mit.edu). It also allows the users to play variants of the Antichess game and also chess. The game supports three types of games (Human vs. Human) (Human vs. Artificial Intelligence) and (Artificial Intelligence vs. Artificial Intelligence)

The complete features of the system are explained in Appendix A of this document.
Revised Specification

The revised specifications are the specifications that changed from our preliminary specifications over the course of implementing the system during the last four weeks.

- The UIHelper class is now called the UIController and the TextUI and GraphicUI no longer extend it.

- The TextUI specification was revised and extended so as to completely specify this user interface (the revisions are discussed later).

- The number of cores of the system is no longer taken into account. We realized that simply generating extra threads did not necessarily guarantee a core from the processor.

- When the amendments came out, we were required to offer some customization to the user so that the user could choose to place any of the three power ups on the game board. We were forced to implement an un-elegant way of doing this, since changing our netbeans code in eclipse would have been next to impossible. Instead, if a user chooses to play a Standard Antichess Game, an input message pops up to ask if they want to customize their power up placement. If they choose yes, three more input messages pop up to ask for a series of board positions for each type of power ups placement. While this is an ugly way of meeting the requirements, it does get the job done by allowing the user to customize power up placement.

- Additionally, if a user wants to load a game and select her/her player type (white or black), they should start a new game with themselves playing as black vs the desired opponent type. After loading a game, the old game board and times will be loaded and assigned to the black and white players. This allows the user, in an awkward fashion, to choose which player he/she plays as when loading a game. While it seems awkward, the alternative would to have some other less intensive options menu pop up when loading a game, to let the user select which player he/she wants to play as. This seemed redundant to us, so we chose to just let the initial options menu in new game do this, and alert the user to this process in the User Manual.
Revised Specifications due to Amendment

Changes in AI
Because of the flexibility of the AI that was talked about earlier in the problem analysis section, the addition of amendments should not force us to rethink anything about our AI. Our AI is a parallelizable Alpha-Beta search that basically looks at all possible future moves in the games and uses an evaluator function to determine the desirability of the leaf states. So having power-ups in the middle of the board does not change anything since the AI will just keep doing what it was doing and if a move results in one of the power-ups to be activated, then the evaluator function is where the difference is seen. So our AI was such that it did not need changing at all when the amendments were released.

Changes in ADT, Game State
When the amendments came out, we were required to offer some customization to the user so that the user could choose to place any of the three power-ups on the game board. We were forced to implement an un-elegant way of doing this, since changing our netbeans code in eclipse would have been next to impossible. Instead, if a user chooses to play a Standard Antichess Game, an input message pops up to ask if they want to customize their power up placement. If they choose yes, three more input messages pop up to ask for a series of board positions for each type of power-ups placement. While this is an ugly way of meeting the requirements, it does get the job done by allowing the user to customize power-up placement.

Additionally, if a user wants to load a game and select her/her player type (white or black), they should start a new game with themselves playing as black vs the desired opponent type. After loading a game, the old game board and times will be loaded and assigned to the black and white players. This allows the user, in an awkward fashion, to choose which player he/she plays as when loading a game. While it seems awkward, the alternative would to have some other less intensive options menu pop up when loading a game, to let the user select which player he/she wants to play as. This seemed redundant to us, so we chose to just let the initial options menu in new game do this, and alert the user to this process in the User Manual.

User Manuals
Refer to Appendix A for TextUI User Manual
Refer to Appendix B for GraphicUI User Manual
Performance

AI Performance

As part of the performance requirement, we were asked to make the AI multithreaded. We were told that our AI would have access to a 32 core machine. Since the core of our AI is an Alpha-Beta search, making the AI multithreaded and parallelizable was relatively easy and straightforward. All we did had to do was to assign different trees for different threads to be search simultaneously.

Additionally, given the fact that the in tournament our AI's game against other AI is timed, we had to find a way to speed up the AI's decision making. This was done by punning and cutting unnecessary game trees. Also, our evaluator function used to evaluate the state in the game had to be extremely fast since it was called at the each leaf of the game tree.

Lastly, the fact that multiple (32) Alpha-Beta search trees would be replicating the board and asked about legal moves, they would also be performing and undoing moves on those board copies on frequent basis, meant that the underlying abstract data types representing the chess board, pieces and state of the game had to be efficient as well.

The amount of resources that our system uses is flexible. This means whether the system has only one core or 32 the AI will try its best given the resources and it will try to never time out of a game. Since out Alpha-Beta search is recursive, the amount of memory it uses does increase linearly as the search tree gets deeper and deeper. That by itself might prevent the AI to efficiently use all the cores available to it. However, given the fact that almost all computers today have at least 1G of memory, this should not be a problem. Our game also stores a very small book of opening moves which should not cause any problems since its size is negligible.
Problem Analysis

For the ADT, the first issue we faced was deciding on what objects had knowledge of the game state. How much should each Piece know about itself? How much should the Board know about its game state? Should the Board or Piece handle legal piece movement for a game?

Also, if the rule set of a game is known, how do we set up the Board? How do we even represent the Board? How do we represent a move on the Board? What object creates a piece? All these answers were resolved during the preliminary design at the start of the project, and resulted in a very satisfactory ADT.

Flexibility was also a key concern when designing our ADT. We knew that there would be an amendment that could possibly affect the board size, game play / rules, piece positions or any combinations of these. When choosing an Abstract Data Type representation for Antichess game, we had to strongly consider how each representation would affect the flexibility of our program.

For the GUI, there were two very important runtime features. First of all, the GraphicUI should have a responsive move handling method. The move handler must be able to get both human and computer moves and play games of: human vs. human, computer vs. human, human vs. computer and computer vs. computer.

The second issue, is having an up to date clock that the user can see. This should include two time capabilities. The first capability is that it should be able to be given an initial time and decrement instantaneously while the player the timer belongs to, is deciding upon a move. The user should be able to observe their decreasing time state on the GraphicUI so that they will be aware of their time (assuming they are playing a timed game).

Related to this issue is the second important feature: the game GUI must always be update and aware of the time. When a player's time reaches 0, the game should declare that the opponent has won and end the game. It should not wait until a player finally makes a move to update the time, because the player could theoretically never make a move, and the game would never be ended.

Here we discuss the other potential problems and our design solutions to them.
User Interface Design/Implementation Choices

Here is the current design for the User Interfaces.

![Diagram showing GUI, TextUI, and UIController](image)

Figure 1.5 GUI and TextUI extend the UIController class

We decided to implement a Helper class because of the redundancy in the code between the user interfaces.

AI Problem Analysis

The three main goals for our AI is to be parallelizable, time conscious and flexible to more than one type of game.

In order for the AI to be parallelizable, it would need to have separable and independent parts that can be threaded. An Alpha-Beta search is the perfect candidate for this since various game trees can be independently searched.

In order for the AI to be time conscious, it has to have a way to estimate how deep it can search given the time constraints. One way to do this is to kill the AI thread if it takes more than a certain amount of time depending on many variables such as time left and difficulty level.

Finally, in order for the AI to be flexible, it needs to have a way of easily performing on different kinds of games. An excellent way to do this is to completely separate the search part of the AI from the static evaluation part. The search part (which is also the parallelizable part) will be exactly the same for any turn based game. It will just traverse the game tree and look through possible future moves. It is the static evaluator that will have to be changed from game to game. The static evaluator is basically responsible for deciding how appealing a certain game state is to end up at for a player. Clearly, different games have different states that are appealing. So the static evaluator is where the change will have to be made. Static evaluators though are usually rather not too complicated (there are, however, some very complicated ones that are beyond our scope, like the evaluator for Deep Blue for example had more 6000 features in it) so it should be relatively easy to have the AI play different games of the same genre such chess, antichess or powerup-antichess.
Design

Overview

A good design for the AntiChess project must address several important issues. These are the User Interface, Thread Handling, Artificial Intelligence and Game State. To that effect, the overall design of our system is modular and allows for great extensibility. It is comprised of three high level modules – AI, UI and ADT. Each of these modules is further divided into classes as required by the specifications and are discussed below.

AI Module

There three main parts to the artificial intelligence of our AntiChess program, a heuristic evaluator, a search algorithm and finally AiPlayer. These two sections are described below:

Heuristic Evaluator

The Heuristic Evaluator class is an evaluator that has the same functionality as a normal Static Evaluator in a chess game. It is different in that it is not static so as to support the concurrent search algorithm used here.

Specification for the features of the Heuristic Evaluator:

   Initially the heuristic evaluator is expected to calculate the heuristic based on four features:
   - The 'material' value which is just the sum of all the piece positions times the piece values.
   - 'pawn' scores is a heuristic of how many pawns there are left and their positions, for end-game scenarios.
   - King safety, how safe is the king in the given state.
   - a heuristic to gauge the mobility of the pieces with respect to those of the opponent.

Possible Heuristics for pieces:

   Pawn: 8
   Bishop: 5
   Knight: 5
   Rook: 3
   Queen: 1
   King: infinity (double.Max)

Search

Our system will use a concurrent min-max searching algorithm. Whenever the AI player has to make the move, it calls the static search method with the state of the game (the state of the board and the time left in the game). The search method then divides the search tree of all possible moves into N-1 pieces (Where N is the number of cores of the system. It's N-1
because 1 core is dedicated to the GUI) and assigns each part of the tree to a thread which then use a min-max search to find the best possible move for the AI.

At each level of the search the heuristic evaluator is called to evaluate the state of the game and return a score. This score is used by the min-max search which tries to maximize the score of the AI and minimize the score of the opponent. The Image bellow shows how the min-max search tree is divided into N-1 parts and how each part runs on a separate thread (core).

![Image](Current State Of The Game)

*Figure 1.6* The Concurrent min-max search tree

**AiPlayer**

The AiPlayer does not observe the game. In other words, the AiPlayer does not know about the game. Whenever it is the AiPlayer’s turn to move, the AiPlayer object is called by the UI with the current state of the board and the time that each player has left. The AiPlayer will then find the best move for that game state (using the AI algorithms described above) and returns the move. The UI then updates the state of the game accordingly.

**UI Module**

The UI Module provides the external interfaces for the user to the game system. The important classes to be noted here are the TextUI, GraphicUI and UIController. The design decisions for the UIs are based on the fact that both interfaces have and share a lot of common functionality. Both the TextUI and GUI use the UIController class. The UIController class helps to define and process the common requirements of TextUI and GUI.

**TextUI Design Choices**

The given specification for the TextUI allowed little room for design or extension. Given that our system uses the UIController for most of the actual gameplay the decision was made to pass the PrintStream from the TextUI to the UIController to allow for a cleaner, more readable codebase.

**GUI Design Choices**

- Multiple screens VS Single screen
We decided to have multiple screens for the menu and the options instead of having them as pop-ups simply because the current design is more intuitive and streamlined to the Anti-Chess application that we’ve built.

- **Board as a grid of JPanels**
We also decided to represent the actual Chess board as a grid of JPanels. Since we pass in the number of rows and columns we need, our design is more flexible to allow for missing parts or extra rows.

One of the other representations we considered was having the board as a grid of simple rectangles. This did not seem to offer any distinct advantage over our current design and hence we chose not to implement the chess board this way.

- **Design Features**
We also have a lot of extra features for our GUI. Although, it would be time consuming to create the functionality we do not expect it to be too much of a problem since our modular design ensures that we only make a couple of function calls to complete the desired objective of each feature.

**Game State/ADT Module**

For the game knowledge, we ended up placing the burden squarely on the Board and three connected classes that combined to generate legal moves. The result was a very flexible, but straightforward, design that allowed us to successfully handle and update the code for the PowerUp Amendment in less than four hours (for the ADT).

The board itself was represented by a 2D Piece array. This was a very good choice in the sense that almost any board game can be represented by a rectangular array of positions. Monopoly, for example, is an array of positions, with only the border positions active, and the rest inactive/null. We felt because of this, we would be able to incorporate most standard board games into our design. In the worst case scenario, of perhaps 4 person chess where the active game board piece aren't in the shape of a rectangle, we assumed we could have made a 12X12 array of positions and just made the four squares in each corner null.

The Piece class was a well conceived class. It has only the barest functionality assigned to it (ie. Name, color, value, isActive and hasMoved). By choosing to represent the piece ownership by a String color, it allowed us to make pieces for any number of players – not just black or white. In fact, color does not necessarily have to represent a color, but any String that designates who the Piece is owned by, such as “player1” or “neutral”. The flexibility of the Piece class was demonstrated by the easy creation of power ups (where the color can be set to “neutral” and heuristic value set to 0) and variant chess and anticchess Pieces, like the dragon_king with its own special movements. Going back to the Monoply example, we could easily have created a “Shoe” or “Cannon” piece for a designated player. If we had confined any of these values to a boolean instead of a String, we would have had trouble adjusting to the amendment.
The only drawback from this is that the Piece knows very little about itself. It is dependent upon the Board class, which is one of the primary reasons that the Board class grew into such a sizable class in the program. The lack of self awareness, however, also made the Piece class very easy to duplicate without worrying about mutability.

An exception to the previous statement, are the two fields in Piece: hasMoved and isActive. While neither are used very often, they become very handy to check if a player had moved during game play (important for en passant or castling) or if the piece is active (should the Piece be in captured pieces list?). Because these fields potentially change during game play, they are mutable, which makes the Piece class mutable, and forced us to make sure these class couldn’t be mutated if we copied Pieces from the Board. However, the ease the two fields offered in handling special cases (like castling) outweighed the slight modifications we had to make to the Piece class observer methods.

The Board itself know very little about the actual legality of piece moves, or generating these moves. Instead, we split the burden of finding Piece moves amongst three other classes. Depending on the ruleset passes to Board, the board can tell if the game being played is a variation of chess or antichess. It can then pass a copy of itself to either of two appropriate classes, which can filter and return the correct available moves for the board.

Our design for getting legal moves for pieces was a very good choice. We have one class that knows the rules for each type of chess/antichess Piece's movement. If given a Piece name, and a 2D array of Pieces, it can call the pieceNameGetLegalMove's method to get the Pieces available aggressive and passive moves. This of course means that we have to write methods for each Piece type (from pawn to variant chess piece gryphon). This reduces flexibility but is very easy in the sense that it is straightforward. For each Piece type, we should know possible directional movement assigned to that Piece type and whether it can “leap” pieces. From this, we can apply this rule/knowledge to check each path for the Piece on the 2D array of Pieces to see if it can capture an opponent's Piece or whether the path is free or blocked. This allows the class to be used by any chess or antichess and could easily be modified in order to generate moves for other types of Pieces, such as checker pieces.

The drawback to this is of course, that for every game, it has to check if the Piece is the same as some Piece that may not even be part of the current game type. For example, if I want to create a set of Moves for a pawn in a standard game of antichess, I have to check if the piece is a “snake” and every other variant of chess pieces. This is counter intuitive, but drastically reduces the amount of duplicated code.

While the AntichessAndChessLegalPieceMovement class offers a umbrella style set of all possible moves, the AntichessPlayerPiecesMoves and ChessPlayerPiecesMoves filter these moves so that only the appropriate Moves for the current game type are actually seen by the Board. Both classes must check that each move generated by th LegalPieceMovement class don't result in the King being in
check, and the AntichessPlayerPiecesMoves class must also make sure that if a player has a capturing move, than it only allows the aggressive moves.

This is also a good choice of code, since Standard Antichess, PowerUp Antichess and Variant Antichess can each use the AntichessPlayerPieceMoves class to generate the possible moves for the game. Similarly, the Standard Chess and Variant Chess games can use the ChessPlayerPiecesMoves. This also helps conserve the amount of code used in the program.

Additionally, a fair degree of flexibility still remains in this code. If we wished to play an entirely different game, say monopoly, we could create a simple move generator class for the game that only allowed orthogonal movements in a clockwise direction. The board is completely separated from generating legal moves; it doesn't need to know how pieces in a game move, it just needs to know what game is being played on it.

That being said, the Board needs to know how to handle each move. This is done in updateBoard(Move m). Since there are certain particular moves for games (like en passant or castling or pawn-to-queen), amendments must be added to how the Board hands certain moves by particular types of Pieces. For each Powerup amendment, we had to add a clause to the update method in order to correctly update the Board.

Perhaps a better method, that would keep the board separate from the game being played, would to have included a makeMove method in each Game's individual Game class. This would also lead to large amounts of duplication of code, however, so since we knew there would be no further amendments after the power up notification, we added the amendment clauses to the the updateBoard method for coding simplicity.

As mentioned, each game has its personal Game class. These classes contain the static values (such as variant piece names and piece heuristic values) along with a setBoard method. This returns a 2D array of Pieces, (the representation of the board state) with the Game Pieces set to their initial starting positions for a new game. This further separates the Board from knowing too much about the game, and made it easy for us to customize placement of amendments for Powerup and Variant antichess, without altering other Game method.

One very important design choice was the choice to store all movable pieces positions with a map of their moves in the Board, and update this map after each move. For each move, the AI and UI controller make at least three calls to Board that depend on the Board having some knowledge of moveable pieces, which would mean that Board would have to make three separate calls to the LegalMoves classes, which are the most inefficient time related calls in the program. For this reason, we decided to store the Map in the Board so that only one call would ever be needed to be made. This freed up a lot of time for the AI search at the cost of making the program less memory efficient.
The Board class also contains a method, undoMove, which allows the Board to undo the last move (mutation free) so that the AI, when searching the tree, may perform a move, undo, and perform another move, instead of having to make a copy of the Board and performing two separate moves. This should help the AI search performance by not forcing the AI to make timely copies of the Board for each move. The only drawback is that this method adds an extra layer of complexity to the updateBoard method, but that is a small price to pay.

The GraphicUI is the most important class with regards to the user. For this reason, a lot of preliminary planning was put into making the game GUI as usable possible for the players.

Our first design decision was to separate the GUI into three pages. When opening the page, we greet the user with a Main menu page. We actually had planned on making a very nice Shallow Red logo for this page, but because of netbean code conflicts, it became very hard to do this, which would explain the plainness of the GUI opening page. However, it would have been a good choice to meet the player with a nice graphic, and the menu still allows them to choose from the very basic options.

The transition from Main Menu to Game Board is made by the Options Menu. We chose to create an options menu since this would keep the user from having to using a small drop down menu in the Game Board. Also, there are numerous options, so it seemed reasonable to create a whole page for options, since we didn’t want to squish all the user options into a small frame.

The creation of three frame pages, however, does create a little more clicking and transitioning for the user. We decided that most users are used to this, though, most computer games have the main menu/options menu/game format, so we tried to follow a format that’s tried and proven.

The choice for a player time table was more or less forced upon us by our initial choice of a table in the release of netbeans GUI for the preliminary design document. We actually would have preferred to use labels, since that is the component that was used in the Stopwatch code that was given to us in the project packet. Labels also could have been made more aesthetically pleasing (both bold and colorful to capture user’s attention) than our table. However, netbeans code was uncooperative, and we were forced to tweak the code so that the table could be the watch listener for both the black and white players.

The game GUI also offers a bunch of various features that were not required for the game, but we deemed would be of use or interest to the user.

The most important decision was to include colored animation for movable pieces and legal piece moves. When playing antichess, it is very difficult to quickly find the piece you must move; you may try moving a pawn for five seconds until you finally realize that a move has opened up that forces your bishop to capture across the board. With colors, you can easily see what pieces actually can/must move, and what their legal movements are. This feature is further described in the user manual. We also realized that some players may not like this feature, in which case they can turn it off, so there is very little downside to this design decision.
The “pause” option was a nice feature for allowing the user to take a break from a game to do something else; a user does not want to get stuck playing a antichess game if he/she gets a call or something in the middle of a game. However, we also realize that this feature can be abused by users. By pausing the game, the user can see the board state without their time counting down. For this reason, the “pause” option is disabled during ranked games.

The “undo” option was another added feature to assist the user during game play. Because the pieces are made by the mouse, the chance of an accidental move are slight, but possible. Also, the user may make one stupid move in the game and quickly realize it – no one wants to have play through a whole game and lose because of one bad move. For either case, we thought the undo move would be a nice way to counteract either scenario. Obviously this method is disabled for ranked games, since users would be tempted to use the option until they finally won the game.

“Hint” was another clever method that could assist user game play. By searching the AI quickly, the user can get an idea of a decent move that he/she can make. This should help develop some sense of a good strategy for antichess or any other game they may be playing.

Having ranked games and a scoring table was a clever idea that follows from the world rankings of chess players. This offers a feature that few other games have: the ability to benchmark yourself against other players and see just how good you are. There is one flaw in this method though. Any player can enter any user name. If “Tom” was the top ranked game, I could play a game as “Tom” and intentionally lose it in order to lower his ranking. To make the game protected would have taken too much effort, so this flaw remains.

Lastly, we chose to three other games to the GUI (Standard Chess, Variant Chess and Variant Antichess). The user can select any of these bonus game and play. We felt this was a good design decision, because the more features and attractive games you can offer a player, the more likely he or she will be likely to play your game instead of some other teams game.

Overall, for the actual look and design of the GUI, using netbeans was a good decision. It gave us very nice graphics, and if we had planned just a little better, we would not have run into the issues of trying to alter the source code. The fact that the code is very resistant to any manual changes is one drawback that cost us a couple times though.
Class Overviews and Runtime Structure

Runtime - TextUI

The TextUI is invoked by compiling and running the system with the TextUI set as the main class. The end result is an interactive system loop through which a user can play either a human vs. human, human vs. computer or computer vs. computer game.

Fig. Human vs. Human TextUI game in progress
GraphicUI Runtime Structure

The GraphicUI has four persistent screens/components.

The Main starting page of the GUI:

![Main page of the GUI](image)

Fig. The Main page of the GUI
On Selecting New Game, the user is taken to an Options Menu where the user can set the parameters to start a new game.
Finally, the actual game screen

![Chess Game Interface](image)

*Fig. The Main Graphical User Interface*
Also, the scores are represented as shown in this screenshot.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Wins</th>
<th>Losses</th>
<th>Ties</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rohan</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1.4</td>
</tr>
<tr>
<td>2</td>
<td>Sam</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0.4</td>
</tr>
<tr>
<td>3</td>
<td>Nand</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Fig. The Scores screen
Module Structure

Here is the module dependency diagram for our entire design:

**Figure - Module Dependency Diagram**

**Legend**

- Interface
- Module/Class

- Uses (Method invocation)
- Implements/Extends
- Weak Use (dashed arrow)

**Figure - The legend for the MDD**
Other Important Dependency Diagrams

UIController
Testing

Strategy
The overall strategy for testing was to have thorough glass box, black box and usability tests. The person writing a class will of course be responsible for glass box testing his class.

For black box testing, every class will be tested according to its specifications.

Our tests will be focused around three categories: correctness, performance (of the AI) and usability in that order. Correctness will of course be the main focus of our testing, however whenever we are satisfied of the correctness of the code, we will test performance of our AI to make sure it is satisfactory. The same thing is done for usability. We will not, however, ever sacrifice correctness for performance or usability.

In order to test the AI performance, after making sure the code passes all correctness tests, we ran our AntiChess AI against the AntiChess programs available on the MIT AntiChess website. We checked whether our AI's performance is at least average (meaning our AI should have at least as many wins as losses). We also tested our AI against the best AI of the previous years and see how well our AI does against the best.

To test the usability, when the game is finalized and after thorough glass box and black box testing, we will ask a few volunteers to play our game. The purpose of this test and the subsequent changes would be to improve usability and will not affect code correctness or performance in any way.
Test results

For the game state classes, we performed extensive JUnit tests. There were many different parts of the game board and movement (ADT) section of our game that needed to be tested for correct functionality and to make sure their states weren't mutated when copies of each object were changed.

Glass box were performed on all small class in the ADT. Classes like Move and Piece had constructors and observer methods tested to make sure there weren't any low level bugs in any of the helper classes used by the main ADT classes.

As described by the Module Dependency Diagram, the classes with the most functionality in the ADT were Board, AntichessAndChessLegalPieceMovement, AntichessPlayerPiecesMoves and ChessPlayerPiecesMoves. While we were able to perform a few small JUnit tests on certain methods in these classes, the most important tests were completed as pseudo integration tests in BoardTest. BoardTest is a glassbox test of the Board class. It contains a set of tests that can be used to test the correctness of the implementation of specific methods in the Board class.

Because Board creates a new AntichessPlayerPiecesMoves, a_p_p_m, or ChessPlayerPiecesMoves, c_p_p_m, object (depending on the game type being played) before every move is made, and many of the Board methods are single calls to the a_p_p_m or c_p_p_m method calls, it would be redundant to test the methods of AntichessPlayerPiecesMoves. Similarly, both PlayerPiecesMoves classes depend on calls to AntichessAndChessLegalPieceMovement. Board depend on the three other subclasses, therefore, the correct performance of the Board object also indicates the correctness of the other three class. For this reason, the Board class is intensely tested, while the other three classes only have the fringe/odd scenarios of their individual methods tested.

Many of the Tests for the Board class follow a set strategy. First, Pieces on a board are moved to desired positions. These Moves are often invalid, but are executed so as to reduce the amount of moves to reach the desired board state. It is possible to do this because the Board's update move does not check to see if the Move is valid (it assumes the user has already checked that the Move is valid). By using the Board.print() method, we were able to easily visualize the state of the Board. We were then able to use our knowledge of the game rules to check whether Board returns the correct moves for a Piece and other specific board state that could possibly occur in the game.

Similarly, for the GraphicalUI, we spent most of our time using observable tests. The human vs. human gameplay was very helpful in allowing us to set up particular situations where the game may have difficulty coming up with the correct moves or exceptions.

The GraphicalUI was almost entirely tested by running different scenarios on. From loading to saving games, undo and hint moves, and simply moving player pieces, all these scenarios had to be visually
tested by running the game GUI. The most serious drawback to this type of testing is that the games are addicting; once you start playing a game to test a small update, it is hard to stop playing the game until you actually finish it.

**AI Testing**

One way to test the multithread capability of our AI was to have our AI running on the staff provided 32 core computer play against our AI running on a single core computer. Out of the several tests, our multithreaded AI had always either defeated the single core AI or had tied. The actual percentage of wins was about %80 while the ties were the other %20. The main reason there were so many draws was because our evaluator did not know about 3-fold drawing and that caused the AI to sometimes unknowingly tie with the opponent.

Additionally, we tested our AI on different computers with different specifications against the Nihilist who were grandmasters of last year's competition. The AIs then outputted the average depth of their Alpha-Beta search tree. The following table describes our results:

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Speed of each Core (Gig Hz)</th>
<th>Memory (Gig Byte)</th>
<th>Average Depth of Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2.2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>2.8</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>32</td>
<td>1.2</td>
<td>2</td>
<td>17</td>
</tr>
</tbody>
</table>

*Table 1. Performance of AI on different machines*

It is clear and intuitive that the more cores available to the AI, the deeper its Alpha-Beta search can go. One of the reasons that there doesn't seem to be any direct relation between the number of cores and the average depth of search is that our AI play cautiously as to not time out. So even if it might be possible for the AI to search somewhat deeper into the game trees specially when running on multi-core systems, our AI eyes on the side of caution and will only search as deep as it is strongly confident will not time out.

The results of this test, though a bit sporadic, do clearly outline the fact that having more cores on the system will enhance the performance of the our AI player. This is tacit validation of the correctness of our parallelizable AI.
Reflection

Evaluation

We believe we have a relatively successful design which is both modular and flexible. The design features, as previously discussed, meet the required elements while offering a variety of optional features that we have implemented.

One of the best features of our design is the limited code duplication between similar classes. For example, the GUI and TextUI both extend UIController which contains common methods shared between the two user interfaces.

Our overall design seemed to serve us well. We did make some revisions to the specifications as we went along but always adhering to general scheme that we had initially come up with. Synchronizing some threads within the GUI did have some issues but they were resolved before the final release.

Some specific notes by Sam:

The issue of copy and paste, I am always in conflict about. While copying and pasting code saves a large amount of time in not having to retype all my code, it also lulls you into a false sense of security about the correctness of the code. 99 times out of 100, I will remember to make the correct changes in similar code that I have copied for another field or method. However, when I forget to make a change on that odd hundredth time, it costs me about as much time in debugging as I saved by pasting.

Setting group deadlines is a very important part of completing a project on time. Actually following these deadlines is even more important. Even though our group set deadlines with a two day window cushion, it always seems like a critical bug would pop up right after we thought we had completed our project. Leaving an even bigger cushion for errors and miscellaneous items would have been a good idea.

This time error also led to another problem in our GUI. Because we were at the deadline during the preliminary release, we implemented a few very rough methods in the GUI that strayed from the original GraphicalUI design we had. Many of these methods were removed, but a few persisted. Some of these methods made grew to the point where they were essential, but also hampered the design and flexibility of the rest of the GUI. If we hadn't hacked the initial code, or completely started fresh again, I think we could have had a better working game GUI.

If you are planning on making the framework of a GUI in netbeans, make sure you get it right the first time. We left out the inactive game pieces panels and a few other panels and labels, and it was a pain to have to search through all the netbeans code to add the desired changes without throwing off the layout given to us.
Lessons
The project, in its entirety, was a learning experience. Being part of a team (albeit a three person one), dealing with a large codebase and dealing with deadlines are some that come to mind.

To be more specific, one of the classes we specified initially was a Core class that would allow us to get the number of cores in a given processor and then use that information to the advantage of our AI. At the time we did not realize that Java did not have a direct way of accomplishing this task. When we realized this, we recognized the importance of having good specification, because, even without that information, our system as a whole did not need changes.

Known Bugs and Limitations

Currently our design does not deal with the “chat” aspect of the game. We know that our GUI is expected to implement the chat feature, but we have not specified the methods in our UIHelper class that will handle the Chat option. This problem originates from our confusion over the interfaces and abstract classes provided in the chat module.

Secondly, we became aware of a severe threading issue near the end of our submission deadline. Our GraphicUI move handler, while being threaded, doesn't actually protect against the AI search blocking the GraphicUI control. When a computer searches, the game GUI can become unresponsive until after the computer returns a move. The only way to fix this would have been to significantly alter part of our UI helper which would have altered our TextUI which would have caused us to meet the deadline. However, the problem is certainly fixable.

To get around this, we limit the search of the AI in the online package to a medium level. This means our program can quickly (in a fraction of a second) return AI moves. The move appears about as fast as the board can rerender the board state, which gives the appearance that our game has full responsiveness.

However, the one issue that our game GUI still has a problem with, is the computer vs. computer game play. Since we re-render all parts of the board (both pieces and panels), the board must be repainted every time a move is made. However, a quick AI will return moves faster than the computer can render each new board state. Therefore, when computer vs. computer is played, the board remains a blur until one player either wins or the game ends in a draw. At this point, the final board state is rendered. This problem would also be fixable by adding in some delays when play a computer vs. computer game. We were expecting to overcome this issue by adding the bonus feature of “replay” game, which would have allowed the user to replay the game after its fury of moves, but we didn't have time to implement this feature. However, this issue is still fixable by adding in time delays in between computer moves so that the computer can render the game board.
Appendix A

TextUI User Manual

Here is a list of commands that are supported by the TextUI. (The items in square brackets are the arguments that need to be passed in.)

StartNewGame [player] [time] [player] [time]
Each player is denoted as "human" or "computer." The times specified are the initial "time remaining" for the player, in milliseconds. If a time argument is zero, then the playing time is unlimited. The first player is white and the second player is black.
Sample Use: StartNewGame computer 60000 human 180000
This starts a new game with the computer playing as white and the human playing as black. The computer will have 1 minute to make all of its moves, and the human will have 3 minutes. The system will output “New game started” on its own line.

SaveGame [filename]
The system saves the game to the given filename and reports “Game saved”.
Sample Use: SaveGame AntichessFile
Type this for the game to be saved as AntichessFile.xml (Note: There is no need to include the file extension here.)

LoadGame [filename]
The system loads the game from the given filename. Once the file is loaded, “Game loaded” is printed on its own line. “Corrupt file” is reported if the file does not have a correct format. If no game is currently in progress from a previously executed StartNewGame or LoadGame command, then the program assumes a human-human game.
Sample Use: LoadGame AntichessFile
Type this for the game to load a file called AntichessFile.xml in the current directory

GetNextMove
Enter this to make the machine player look for a move.
If this command is called during a human player's turn, the command prints “Human turn” on its own line. If this command is called during a machine player's turn, print on its own line the next move it believes to be the best.

MakeNextMove
If it is a human player's turn, the system prints “Please specify human move”. If it is the computer player's turn, and GetNextMove has not yet been called on this turn, then the system will print “First GetNextMove”. Otherwise, the system performs the move that GetNextMove would return.

MakeMove [move] [time]
Perform the move specified by the move string, in the "standard string format" (see below). The time parameter is specified in milliseconds. This command should only be used by a Human Player. If it is
used during a computer player's turn, nothing will happen to the game state and no response is printed. If the move is not legal, the system prints "**Illegal move**" and not perform the move. If the move is legal, the system should perform the move, decrement the player's time by the amount given, and print the move performed, in proper format, on its own line. If the player's time is unlimited, then the time argument is ignored (but must still be present).

**Sample Use:** MakeMove a2-a4 1000
This move the piece from a2-a4 and tells the game that the move took 1 second to make.

**PrintBoard**
System prints the current "state" of the game to the screen using the same format as if it were being saved to a file.

**IsLegalMove [move]**
System prints either "**legal**" or "**illegal**" to specify if the move is a legal next move.

**Sample Use:** IsLegalMove a2-a4

**PrintAllMoves**
System prints, in alphanumeric order, all legal moves for the next player.

**GetTime [player]**
The system prints the time remaining in milliseconds for the player specified.

**Sample Use:** GetTime white
The system will print 3000 to indicate 3 seconds left for the white player. If the time for the player is unlimited, the system prints "unlimited".

**QuitGame**
Prints “**Exiting game**” and terminates the present game and application. Note that, QuitGame cannot be the first command.

[Note: All commands and arguments are case sensitive.]
Appendix B
GraphicUI User Manual

Main Screen:

When you open the Shallow Red program, you are immediately given an assortment of choices.

New Game – If you click the “New Game” text area, you will be transitioned to an Options page where you may select the game you would like to play and customize other game play options.

Load Game – If you click the “Load Game” text area, the main chessboard page opens up along with a directory finder that you may select a previously saved game file (in .xml format) to load and play.

Instructions – If you click the “Instructions” text area, you will be directed to this user manual. Explanations for all game options and game play are provided here.

View Scores – If you click the “View Scores” text area, you will be directed to a table of scores for previous players that are ranked from first to last place. If you have played any ranked games, you can see how well you stack up against other players.

Exit – If you click the “Exit” text area, you will exit completely from the Shallow Red program.

Game Options:

The games options menu is an essential part of setting up customized game play. You, or your opponent, (if you are playing human vs. human) are allowed to set various game fields to your own preference(s).

Game Type – The game type combo box selects the current game type to be played for the game. You may select from any of the five game types offered:

Standard Antichess
Antichess + Ammendment
Standard Chess
Variant Antichess

Antichess is a variation of chess. While all pieces still possess the same directional movements, the rules for constraining these moves have been altered. While checkmating the opponent still results in a win for you, you may also win by timing out your opponent or by losing all of your pieces (except for your King) first. More detailed instructions on the rules of Antichess can be found at http://en.wikipedia.org/wiki/Antichess.

Chess is a universally known and played board game. While there are slight variations of chess, the general objective is to mate your opponents King before timing out. This has to be done before your opponent does this to you or you lose. More detailed instructions on the rules of Chess can be found at http://en.wikipedia.org/wiki/Chess.

All five games do include certain rules of antichess and chess that aren't always used in some games. These fringe rules consist of:

* **En Passant**: If an opposing pawn has crossed the dividing row (white to row 5 or black to row 4), and you move a pawn, that hasn't moved been moved previously, two spaces ahead, and your pawn's path crosses the capturing path of the opponent's pawn, then your opponent may (or in antichess – must) capture your pawn, moving its pawn to the square one row in front of the starting position of your pawn. Based on the number of conditional statements used in simply describing this scenario, it should be apparent that this move is rarely used in chess, but occasionally becomes significant in chess due to the must capture clause. More detailed instructions on the rules of En Passant can be found at http://en.wikipedia.org/wiki/En_passant.

* **Castling**: Castling occurs between a king and a castle only if they both have not moved. If both the king and castle are in their original starting positions, and the King is not in check in its current position, in its transition position (one column to the left or right), or in its final position (two columns to the left or right of its starting position), then it is allowed to move two columns towards the direction of the castle. In turn, the castle skips over the King, and finally ends up in between the King's final position and its starting position. More detailed instructions on the rules of Castling can be found at http://en.wikipedia.org/wiki/Castling.

* **Pawn to Queen**: If you one of your pawns move all the way over to your opponents side (row 8 if you are playing as white, row 1 if you are playing as black), then the pawn is redeemed for a queen of the same color.

**Pieces in Game:**
Standard Antichess: Possesses all the same pieces as standard chess: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king per player.

Antichess + Ammendment: Possesses all the same pieces as standard chess: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king per player. Also, this game possesses three different power ups: upgrade, destroy, spawn. The game has the option of having power-ups placed on the board. When customization of powerup placement is not selected, the game automatically assigns 2 upgrade, 2 destroy and 2 spawn power ups on the board in both a random and symmetric fashion.

Standard Chess: Possesses all the same pieces as standard chess: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king per player.

Variant Antichess: Possesses all the same pieces as standard chess: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king per player. Also, this game possesses nine different power ups: upgrade, destroy, spawn, traitor, berlin pawn, snake, chameleon, gryphon, dragon king. Five of these powerups, when captured, result in the creation of in a particular variant piece of the same name: berlin pawn, snake, chameleon, gryphon, dragon king. The game has the option of having power-ups placed on the board. When customization of powerup placement is not selected, the game automatically assigns any combination of 2 upgrade, 2 destroy, 2 spawn, 2 berlin pawn, 2 snake, 2 chameleon, 2 gryphon or 2 dragon king power ups on the board in both a random and symmetric fashion, so that they add up to six total power ups.

Variant Chess: Possesses all the same pieces as standard chess: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king per player. Also, this game possesses nine different power ups: upgrade, destroy, spawn, traitor, berlin pawn, snake, chameleon, gryphon, dragon king. Five of these powerups, when captured, result in the creation of in a particular variant piece of the same name: berlin pawn, snake, chameleon, gryphon, dragon king. The game has the option of having power-ups placed on the board. When customization of powerup placement is not selected, the game automatically assigns any combination of 2 upgrade, 2 destroy, 2 spawn, 2 berlin pawn, 2 snake, 2 chameleon, 2 gryphon or 2 dragon king power ups on the board in both a random and symmetric fashion, so that they add up to six total power ups.

Piece Movement:

Pawn:
Moves: A pawn may move forward in the direction away from its starting side of the board (black pawn towards row 1 and white pawn towards row 8). If they have not moved in the current game,
and no pieces obstruct their path, they may leap two rows forward. Otherwise, they may only move one row per move.

Captures: A pawn captures diagonally in a direction that leads them away from their starting row on the board.

Bishop:
Moves: A bishop moves in all four diagonal directions. It may not leap over pieces.
Captures: A bishop captures by moving in any of the four diagonal directions.

Knight:
Moves: A knight leaps to any space that can be reached by moving one space orthogonally, followed by another space diagonally outward, or vice versa.
Captures: A knight captures in the same way that it moves.

Rook:
Moves: A rook may only move orthogonally.
Captures: A rook captures orthogonally.

Queen:
Moves: A queen may move in any diagonal or orthogonal direction available to it.
Captures: A queen captures in the same way that it moves.

King:
Moves: A king may move one step in any diagonal or orthogonal direction available to it, with the condition that after the move is made, the King is not in check by the opposing piece.
Captures: A king captures under the same conditions as it moves.
**Variant Pieces:** These pieces aren't found in the standard antichess game and take on peculiar behaviors. They are only found in “Variant Chess” and “Variant Antichess” in the Game Type option in the Options Menu. For a better description of variant pieces found in chess and their associated properties, please see: [http://www.chessvariants.org/index/mainquery.php?type=Piececlopedia&orderby=LinkText&displayauthor=1&displayinventor=1&usethisheading=Piececlopedia](http://www.chessvariants.org/index/mainquery.php?type=Piececlopedia&orderby=LinkText&displayauthor=1&displayinventor=1&usethisheading=Piececlopedia).

Enjoy...

**Berlin Pawn:**
Moves: A berlin pawn moves one step diagonally forward (or two steps from the initial pawn row, subject to e.p. capture)
Captures: A berlin pawn captures by moving one step directly forward.

**Snake:**
Moves: A snake moves like a traditional pawn, except it may recoil one move backwards as well.
Captures: A snake captures diagonally, like a traditional pawn.

**Chameleon:**
Moves: The chameleon takes on the movement of the last piece moved by the opponent. If the last move was null, then the chameleon has no valid movements. Otherwise it mimics the other piece.
Captures: A chameleon may capture in the same way that it may move.

**Gryphon:**
Moves: The Gryphon moves one square diagonally in any direction, the continues away from the starting square in an orthogonal direction. Its path must be unoccupied; it can not jump over pieces.
Capture: It captures an enemy piece by moving onto the square occupied by that piece.

**Dragon King:**
Moves: A dragon king moves any number of spaces orthogonally, or it may step one space diagonally.
Captures: A dragon king may capture in the same way that it may move.

**Power Ups:** When captured, power ups always transform the game state in some way that can be both beneficial or detrimental.
depending on the piece type or the game type you are playing. When moving, pieces can move through or over power ups. Power ups are only captured when a piece lands directly on the square they are located at. When a power up is captured, it transforms the capturing piece and is removed from the game. Kings are not affected by power ups. For more information on the three original power ups (upgrade, spawn and destroy), please see http://www.mit.edu/~6.170/assignments/antichess/ft07-amendment.html.

Upgrade:
An upgrade turns the capturing piece into a queen piece of the same color.

Spawn:
A spawn creates a copy of the capturing piece and places it to the right of the capturing piece (with respect to White's viewpoint). If no space is available, the piece is added to the next open space in a counter clockwise direction. If no spaces are open adjacent to the power up, the copy of the spawned piece is not added to the board.

Destroy:
Destroy removes the capturing piece from the board.

Traitor:
Traitor switches the alliance/color of the capturing piece to the opponents.

Berlin Pawn:
The Berlin Pawn Power Up transforms the capturing piece into a berlin pawn piece with the same color as the capturing piece.

Snake:
The Snake Power Up transforms the capturing piece into a snake piece with the same color as the capturing piece.
Chameleon:
The Chameleon Power Up transforms the capturing piece into a chameleon piece with the same color as the capturing piece.

Gryphon:
The Gryphon Power Up transforms the capturing piece into a gryphon pawn piece with the same color as the capturing piece.

Dragon King:
The dragon Power Up transforms the capturing piece into a dragon piece with the same color as the capturing piece.

**Game Theme:** The “Game Theme” option allows you to customize the look and feel of the pieces used in the game. Available piece themes are:

- **3D**: choose this to play a game using 3D pieces (Colors: white, red)
- **Metallic**: choose this to play a game using metallic looking pieces with Indian styled figures (Colors: white, black)
- **Minimalist**: choose this to play game using very basic 2D pieces (Colors: white, blue)

**Ranked Game:** Check this box to play a ranked game. To help standardize game results, you are only allowed to play human vs. computer games for a ranked game. The computers' level of difficulty is automatically set to “expert”, which is only surpassed by “grandmaster” in difficulty. Additionally time for a ranked game is set to 10 minutes for both you and the computer. The game only records scores for Standard Antichess, so you may only play this variation for a ranked game.
**White:** The white player is allowed to set various settings of the game.

**Name:** If you choose to play as white, enter your name, or unique game name in the name text field. It is essential to have a unique game name in order to receive your own ranking if you choose to play a ranked game.

**Computer:** If you wish to have the computer play as White, select the computer button.

**AI Level:** AI Level ranges from easy difficulty level ("novice") to increasing difficulty ("grandmaster"). Depending on how much of a challenge you desire, or your experience, you should select this level accordingly.

**Mortal:** If you decide to play as White, select this button.

**Timed:** Select the Timed button area to play a timed game for White. Enter the desired time length for white. The first text field represents hours, the second minutes, and the last box seconds. Enter any desired duration of game time, though we suggest between 5 and 10 minutes.

**Untimed:** An untimed game gives you unlimited time to decide on a move.

**Black:** The black player is allowed to set various settings of the game.

**Name:** If you choose to play as black, enter your name, or unique game name in the name text field. It is essential to have a unique game name in order to receive your own ranking if you choose to play a ranked game.

**Computer:** If you wish to have the computer play as Black, select the computer button.

**AI Level:** AI Level ranges from easy difficulty level ("novice") to increasing difficulty ("grandmaster"). Depending on how much of a challenge you desire, or your experience, you should select this level accordingly.

**Mortal:** If you decide to play as Black, select this button.
Timed: Select the Timed button area to play a timed game for Black. Enter the desired time length for black. The first text field represents hours, the second minutes, and the last box seconds. Enter any desired duration of game time, though we suggest between 5 and 10 minutes.

Untimed: An untimed game gives you unlimited time to decide on a move.

Main: This button, when clicked, will transfer you back to the Main page where you may choose to Exit, View Scores, etc...

Play: This button, when clicked, will take you to the Game page. Until this point you may update any of the options fields, but afterwards the page reads these current selections and creates a game based on your changes.

Game Board: The game page is where the fun begins. Here you may play the current game, start a new game, chat, and perform many other cool features.
**File:** This menu contains all functionality of the Main page.

**New game:** When New Game is selected from the File menu, the page reverts back to the Options page where you may set up a new game.

**Load game:** When Load Game is selected from the File Menu, a directory finder pops and allows you to search through your directories and load a previously saved game.

**Save game:** When Save Game is selected from the File Menu, a save file directory pops up and allows you to save a game. All saved games must be saved in the form “filename.xml”.

**View Scores:** When View Scores is selected from the File Menu, the scores page pops up, where you can look up your score (or rivals scores) to see where you stand in ranking.

**Quit:** When Quit is selected from the File Menu, a message pops up that gives you the option to save the current game. If you select “yes” from the popup, you can save the current game. If you select “no”, the game page closes and you return the Main page. If you click “cancel” you simply close the popup and return to the current game you are playing.

**Options:** The Options Menu contains some of the must important functions that can by used during game play.

**Undo:** When Undo is selected from the Options Menu, and the current player is a mortal/human, the last two moves are undone on the board. This allows the current player to go back and redo the previous move. If the current player is a computer, than only one move is undone, which returns the move control to the human player. Note: The time of the current game is not updated to the previous game times of the players. Also, this option is discontinued during a ranked game, so make sure you make good choices the first time around.

**Pause:** When Pause is selected from the Options Menu, the times for each player (if they are playing a timed game) are halted. This allows either player to take a break from the game. A message is shown to remind players that the game has been pause, and the message must be exited to resume game play.

**Note:** This option is discontinued during a ranked game, so you should be ready to play up to 20 consecutive minutes
**Hint:** When Hint is selected from the Options Menu, a move, that for the next few immediate moves will be good for you, is returned. You may choose to make this move on the board, or ignore it. Selecting Hint multiple times may result in alternative moves. Note: This option is discontinued during a ranked game, since you are solely ranked on your own personal skill as an antichess player.

**Theme:** When Theme is selected from the Options Menu, you are given the option of selecting any of the three themes (described earlier under the Options Page section).

**Animation:** When Animation is selected from the Options Menu, you are given the option of turning Animation on or off. More specifically, this either turns on or off the coloring of legal moves and moveable pieces. The default for a new game is with the Animation turned on, since this allows you to see movable pieces during an antichess game. However, you may prefer to turn off animation while playing a variation of chess.

**Help:** This menu simply allows you to select this user manual

**Players table:** In the upper right corner of the game, a player table is provided. The “player” column of the table provides you with player colors. Also, the current player (whose turn it is to move) is denoted by a “>” next to the start of color. For example “> WHITE” would indicate that white now must move. The “name” column of the table pairs the player color with the player's name. The last column represents the time remaining for each player. If the player is playing an untimed game, the cell will read as “untimed”, otherwise, the players remaining time is displayed in the form minutes:seconds (ex. 4:53). When the player is expected to make a move, the time will decrement every second and be updated on the screen for the players convenience.

**Previous Moves Table:** The previous moves table displays the previous moves made by each player throughout the current game. The left column displays all moves made by white, and the right column represents black moves.

**Message Board:** The message board displays messages from the computer regarding the current game status, your game playing style, your personal hygiene, etc...

**Move Text Field:** The move text field may be used to enter moves manually. By entering a move in the form “a2-a3” (first board position, hyphen, second board position) you can update the board. After entering the desired move, you must click the “Move” button to the left of the text field to actually enter the move and update the board.
Options Buttons: At the bottom of the board there are five buttons that you may access quickly for convenience.

Pass: The “Pass” button may by the most obscure button displayed for game play. It is only activated when the current player has no legal moves and must “pass” and allow the opponent to make a move. A message should appear in message board to alert the current player that they have no legal moves and must pass.

Undo: This button possesses the same functionality as the “undo” option in the Options menu at the top of the game page.

Hint: This button possesses the same functionality as the “hint” option in the Options menu at the top of the game page.

Pause: This button possesses the same functionality as the “pause” option in the Options menu at the top of the game page.

The Board:

The Board itself is checkered in black and white. The pieces for the White player are placed at the bottom of the page (rows 1 and 2) while the Black player pieces are initially placed at the top of the page (rows 7 and 8). White starts the game (if it is new, and not a loaded game).
The most convenient, and quickest method, of moving pieces on the board, is by clicking with your mouse. One click on a piece (you may click anywhere within the boundaries of the square the piece currently rests on), selects the piece. A second click on the square unselects that piece position. However, if you click on another square, the game will attempt to move the piece there. If the resulting move is illegal for the current game state, no updates will occur, and you must try to move the piece to a valid square. If the move was legal, then the board will move the selected piece to the requested board position.

A useful functionality of the game is that you may choose to view animation of pieces. You may select this from the “Options” menu. When animation is turned on, and you are the player whose turn it is to make a move, the following animation will occur:

The pieces that you can move will be indicated by turning the board square they rest on green. If you scroll over any of the movable pieces, the moveable piece position that your mouse currently is placed over will turn yellow to indicate that it has been selected. All the legal moves this piece can make will be indicated by turning their board squares blue. If the piece has no legal moves, it will not be highlighted in yellow, and instead, the pieces that do have moves will reappear as green. If you actually click on a piece that has valid moves, it will become highlighted as yellow as before, but the scrolling property will be unenabled. If you click on any square that isn't indicated as a valid move, the movable pieces will reappear as green. However, if the move is valid (from yellow square to blue square) the board will be updated with this move and move control to your opponent so that he/she/it may make it's next move. The last player's move is also always shown as purple (only the starting and ending squares are colored purple – not the complete path traveled by the piece).

Directly above and below the game board are spaces that hold the captured pieces. Captured black pieces are shown at the top of the board, on the black side of the board, and white pieces are likewise shown at the bottom of the board, underneath the white side of the board. Each time a piece is captured and removed from the board, it is placed on the corresponding color's side.

If you are curious about the actual rules of the current game and the possible piece movement, please see the earlier information given about these subjects under the “Options Page” section, subsection: game themes.
Scoring:

As explained earlier, you can choose to have your game ranked. If you choose this option, a new score profile will be created under your name. From then on, whenever you use your profile to play a ranked game, your score will be updated in our database.

Here is the formula for your score:

\[
\text{Score} = \text{Total Wins} + \left( \frac{\text{Total Draws}}{2} \right) + \left( \frac{\text{Number of Games Played}}{5} \right)
\]

Additionally, as described earlier in the user manual, by click on the View Score button at the main menu, you can check the ranking of all players that have ever played a ranked game against Shallow Red. Here is how the ranking page looks like:
Appendix C

Specification/Implementation level comments

Please refer to the supplied Java docs and code base for specification level comments (Abstraction Functions, Representation Invariants).
Appendix D
Module Dependency Diagram (MDD)

Figure 1.4 Module Dependency Diagram
Appendix E

Additional Features in the GUI

Apart from the basic required functionality of the GUI, it has been designed to implement the following features (time permitting):

- Ability to choose between Chess and Anti-Chess and variants of the same
  There is an option to choose either a Chess game or an Anti-Chess game in the Options pane.
- A Scoring system (based on the International Chess Ranking System)
  The Scoring system is name based and lays constraints on the options that the user has control over in the Options Menu. For example, if the user checks the Ranked Game checkbox then he/she cannot set the values for the timer or the intelligence level of the AI.
  Each win is 1 point, a loss is 0 points while a tie is half a point and this sum is normalized over the number of games played.
- Piece themes
  There are multiple sets of themed images for chess pieces that the user can choose from.
- Hint Section
  Clicking the Hint button in the Main GUI gives the user some good moves that he/she can make without giving the advantage to the AI.
- Pause
  The ability to pause a game. The actually Panel with the board gets painted over so that the user cannot access the game when it is paused.
- Undo
  The ability to undo a set of moves after the AI has made its move. This is disabled when the user plays a ranked game.
- Animation
  Simple movement of the piece from old position to the new position when the user clicks the desired positions on the board.
- A semi-intelligent chat box