6.170 Preliminary Design Document

Team 44 – "The Nihilists"

May 1, 2007

System Design

Overview

The overall design of our Antichess program follows an MVC pattern. Our system consists of four major components, each of which is subdivided into classes as appropriate: the Board, the View, the Controller, and the AI. The Board component serves as the model in the MVC pattern. In Antichess it acts as programmatic model of the chess board, equipped with chess pieces and the logic to move those pieces on the board and distinguish between legal and illegal moves. The View is the only component that directly interacts with the user. The View represents the board to the user in some manner (graphically or textually) and accepts input from the user of the moves he/she would like to make. While the View can read the current status of the Board directly, it can only modify the Board through the Controller. The Controller accepts requests from the user and modifies the state of the Board appropriately and safely as a result of those requests, then pushes the result of these changes back to the View. The final major component of this system are the game players. On his/her turn each game player queries the Board for its current state and based off of this state makes a decision concerning the next move it would like to make. The game player then submits this desired move to the Controller, which either determines that the requested move is legal and updates the Board accordingly by applying that move to the Board, or determines that the requested move is illegal and responds accordingly to the game player. To support both human and AI play, two game players are implemented, one for each of these different types of players. The human player simply takes input passed from the view concerning the moves the human would like to make, while the AI will ultimately perform a multithreaded minimax search of future possible gamespaces and decide upon a desired move based on the results of this search.

Module overviews

The Controller

Due to the inherent complexity of the system, and the crucial role of the controller coupled with a relatively low amount of actual code needed to actually write the controller, we chose to use a prototyping model for development with the controller. This current iteration has a lot of nice things, but some failings. The controller can easily be divided into two separate parts, the ControllerMaster and the GameController. The ControllerMaster handles creating new games, saving games, loading games and setting the state of the GameController. The GameController handles the actual running of a Chess or Antichess Game.

First we will discuss the GameController class. The GameController class handles communication with GamePlayers, the Board and the GameClock. A GamePlayer is an interface which represents any entity the GameController can ask for legal moves. The Board and the GameClock are part of the model. The cycle of communication goes as follows: The GameController asks the Board whose turn it is. It then asks who ever turns it is for a move. If that GamePlayer has a move they return it, if they do not they return null and the GameController waits for the GamePlayer to notify that it has a move. The GameController asks the Board to do the move and then checks to see if anyone has won from time conditions or from conditions that the Board determines. Interspersed in these actions are checks that the time has not run out and if the ControllerMaster wants the GameController to halt it's operations.

The GameController communicates to the View and the AI through the GamePlayer interface. A GamePlayer, as mentioned before, represents an entity that the GameController can ask for moves. This includes RandomAI, AIPlayer, and HumanPlayer. HumanPlayer is an interface for communicating between the GameController and the View. This was implemented for Chess and Antichess with the class ChessPlayer. The HumanPlayer player communicates to the View through anyting that extends BoardView.

The ControllerMaster saves games by using the GameWriter class. It loads games by using the GameReader class. When it wants the GameController to stop the ControllerMaster sets an internal value requesting a pause. It is the responsibility of the GameController to check if the ControllerMaster wants a pause. This also requires that the ControllerMaster knows whether the GameController is running or not. This is handled by having the GameController inform the ControllerMaster whenever it is switching from running to not running or vice versa. To summarize, the ControllerMaster sets flags, the GameController must check the state of the ControllerMaster and inform it how its doing. The newGame, saveGame, and loadGame methods are called directly from the View.

Things that will improve in later iterations of the the Controller: The communication between the GameController and the ControllerMaster needs to be improved. Pausing and then restarting the GameController requires that the ControllerMaster spawns new threads. There is a better way. Also rather late in development (last Friday) the Board was made to be generic. The GameController does not take this into account and it would be best if it did.

The Board

The antichess Board will necessarily need to remember the arrangement of chess pieces on the current board and what moves those pieces can legally make. In order to handle a range of move and rule complexities we have divided the design of our Board into a hierarchy of abstract classes and interfaces. At the base of this hierarchy are the Pieces that go on the board. Pieces follow a flyweight pattern, where each piece on the board is a singleton implementation of an abstract piece type associated with some state information, including its location on the board. Pieces are immutable, so changes in a piece's location will result in a new piece object, a new set of state information associated with an existing piece type implementation. Pieces possess Move Generators based on their piece type, which determine what moves are legal on the single piece level. These Move Generators return Moves, which simply contain information about an action some piece takes during a single turn of the game. The Board containing all of the pieces for a given game then looks at all of the moves these Move Generators deam as possible and examines overarching consequences of the moves to finally determine their legality in the game; for example, the board determines if some legal move would leave the current player's king in check, which would render it illegal in the game. In this way, the Move Generators for the Pieces feed piece-level legal moves up to the Board, which then looks at the greater consequences of these moves to determine if they truely are legal.

Our particular implementation of the Board allows for a large amount of flexibility. In our implementation, while Moves fit into a general framework of a Piece and some pair of coordinates that that piece effects in a given turn, this design is subclassed to contain more specific data based on the game. In the case of Chess, therefore, the afforementioned pair of coordinates depict where that piece is moving to in that turn, and ChessMoves can optionally remember a captured Piece which the Piece captures in some way as a result of its movement on the Board. The general abstract Board class is similarly subclassed depending on the game in question, allowing this abstract class to cater to a variety of Games and for data that the Board needs to manage but is unrelated to the game of Chess or Antichess to be handled outside of the specific implementation of the Board for Chess or Antichess. Furthermore, since many of the rules for Antichess are very similar to those for Chess, we chose to implement the Antichess Board as a sublcass of the Chess Board, allowing the Antichess Board to be solely concerned with those details of the game that differed from Chess and allowing our system to be sufficiently flexible to handle either of these games.

Timing is handled in two classes. A GameTimer represents one timer ticking down. This has start and stop methods. A GameTimer also contains a list of A GameClock represents many GameTimers with at most one timer running at one time.

The View

A BoardView is a class that displays a Board and accepts input from the user. The BoardView is set to accept input by a HumanPlayer. From the user input, the BoardView generates sets of coordinates that it passes to the HumanPlayer who then converts the sets of coordinates to moves, checks if they are legal and then notifies the GameController that there is a move.

The main class in the view when playing Chess or Antichess is the ChessGUI. The ChessGUI contains, creates and displays all of the elements of the View. The ChessGUI communicates directly with the ControllerMaster class when requesting the creation of new games, saving games, and loading games. Loading and savig are handled in through dialogs initiated by menu items. Creating a new game is handled by the NewGameWindow class. For displaying the state of the Board and the Clock the ChessGUI contains: a ChessBoardView which extends BoardView, MoveHistoryView, and TimerLabel. BoardView is described above. MoveHistoryView displays the moves that have been performed in this game. A TimerLabel displays a timer that is ticking down during a persons move. A TimerLabel is a display for a GameTimer.

The AI

The Antichess AI uses a standard minimax search algorithm to search the space of potential moves. It implements the alpha-beta heuristic to prune the game tree, and uses iterative deepening to search the move space in a breadth-first fashion. In addition, when searching at a deeper level, the AI searches the principal variation discovered at a shallower search depth in order to hopefull achieve an alpha-beta cutoff sooner.

The AI also implements the so-called "Killer Heuristic" – when searching a given level of the game tree, the move that last caused a cutoff at the same level is searched first, again with the hope of causing a quick cutoff.

While searching, the AI module makes use of the Board's ability to perform and then undo moves to search, by performing moves, then recursively doing an evaluation of the move, then undoing the move. All searching is executed on a local copy, in order to ensure that the copy referenced by the View and Controller remains up-to-date and consistent.

At present, the AI is not yet multithreaded; Concurrent search of the game tree will be implemented for the final release.

The AI is abstracted into the `GameAI` hierarchy of classes; There is also a `AIPlayer` class that implements `GamePlayer` that allows the AI to be used by the controller during interactive game play. The MachinePlayer class, for interacting with other computer implementations, uses a `GameAI` object directly.

Testing

Our overall testing strategy was composed of essentially two parts. First, we wrote unit tests, black box and glass box alike, to test specific methods inside of our classes. In some cases we would write test cases for classes that we did not write ourselves, in order to provide greater detachment from the code we were testing while we were designing our tests. In a number of cases, however, simple unit testing of individual methods was simply not a practical means of testing our system. In the Board, for example, it was impractical to attempt to test several of the methods on a roughly individual basis; more integrated tests, such as those between determining what legal moves existed on the Board and whether the Board could properly execute those moves, were simply more practical to test together, even though the final result was that the successful completion of those tests depended on a number of Board methods all working properly together. This feature of our system was found to be highly prevalent, so integration tests, both automated as just described and non-automated through 3rd party user interaction with the working system, were a crucial component of our testing strategy. Every member of the team tested the working system in its various stages (as a Chess program and as an Antichess program; with a respectable AI and with an AI that made purely random moves) and contributed to finding additional desirable features and bugs in the system. The program was also passed to 3rd parties (fulfilling our quota of jokingly named ”Angry Russian testing”) who had no previous experience with our system in order to provide a different perspecitve on the functionality of our system. This variety of testing approaches to our system created a highly comprehensive set of tests for our system.

Appendix

Figures and Diagrams

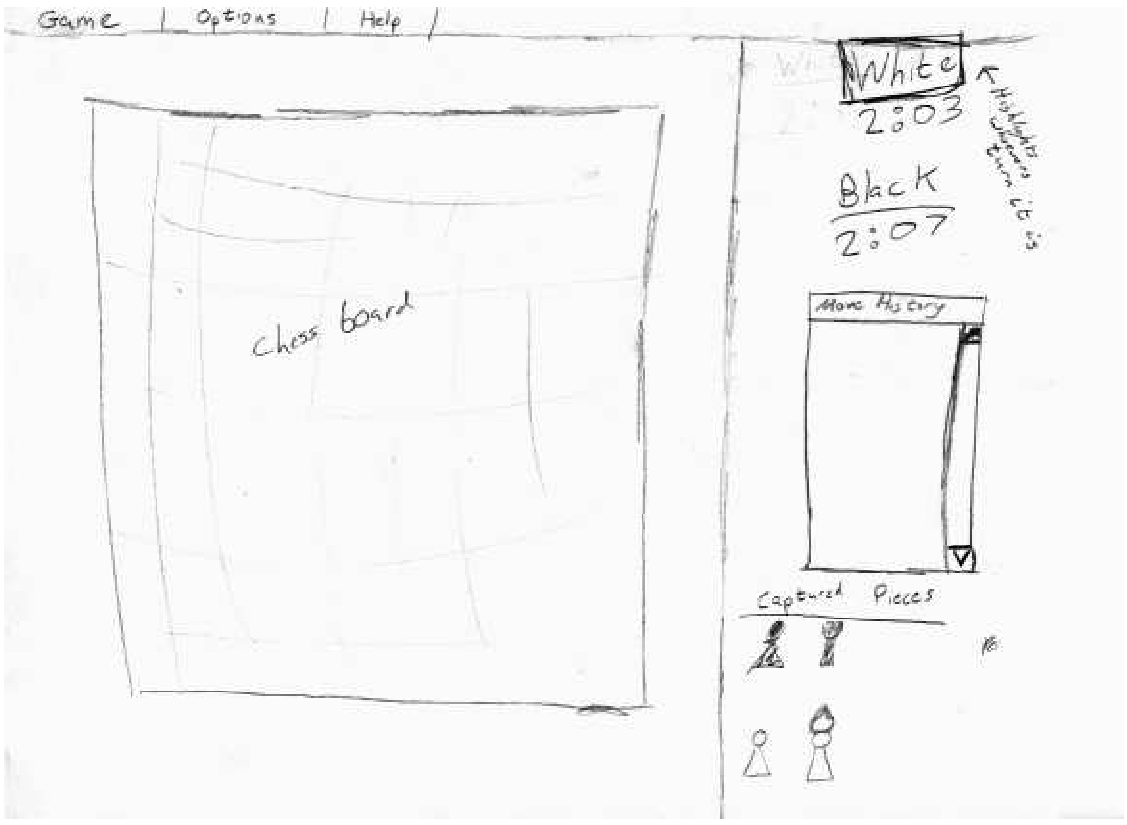

Figure 1: Mockup of the game UI

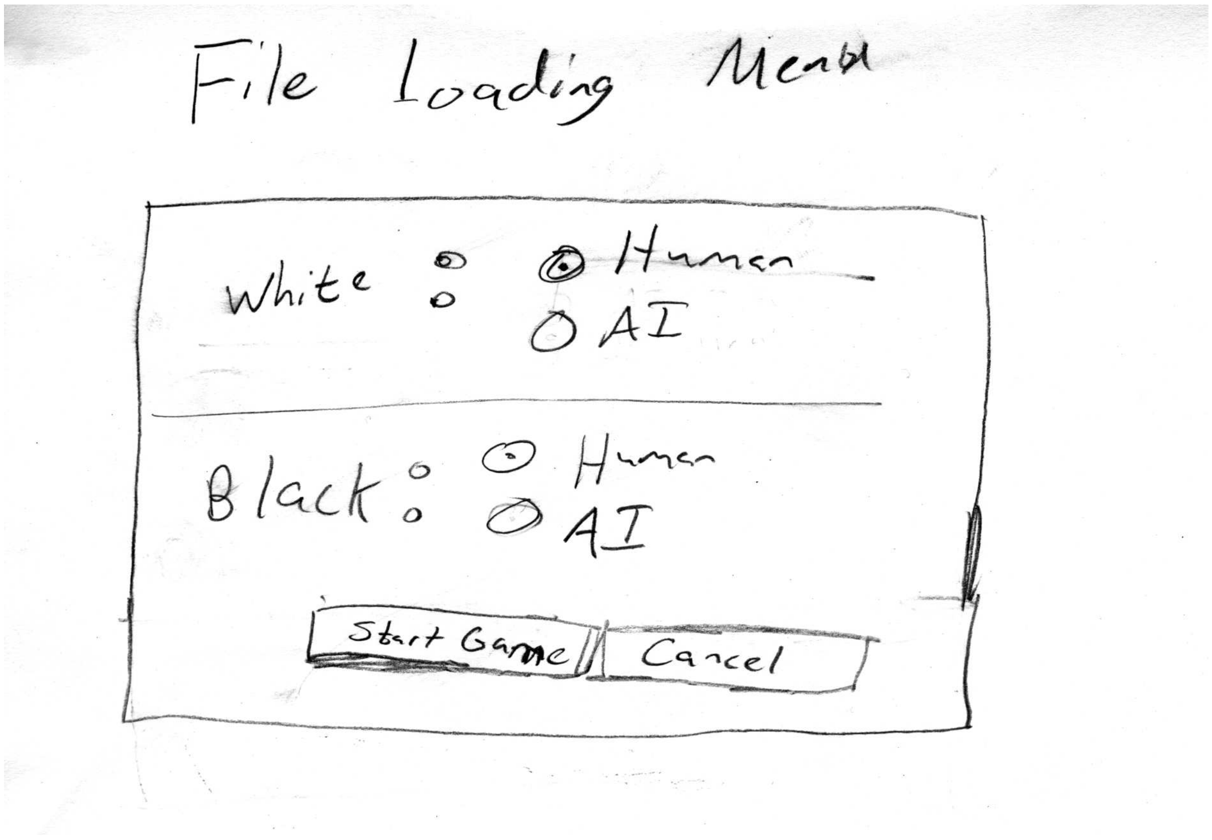

Figure 2: Mockup of the new game dialog

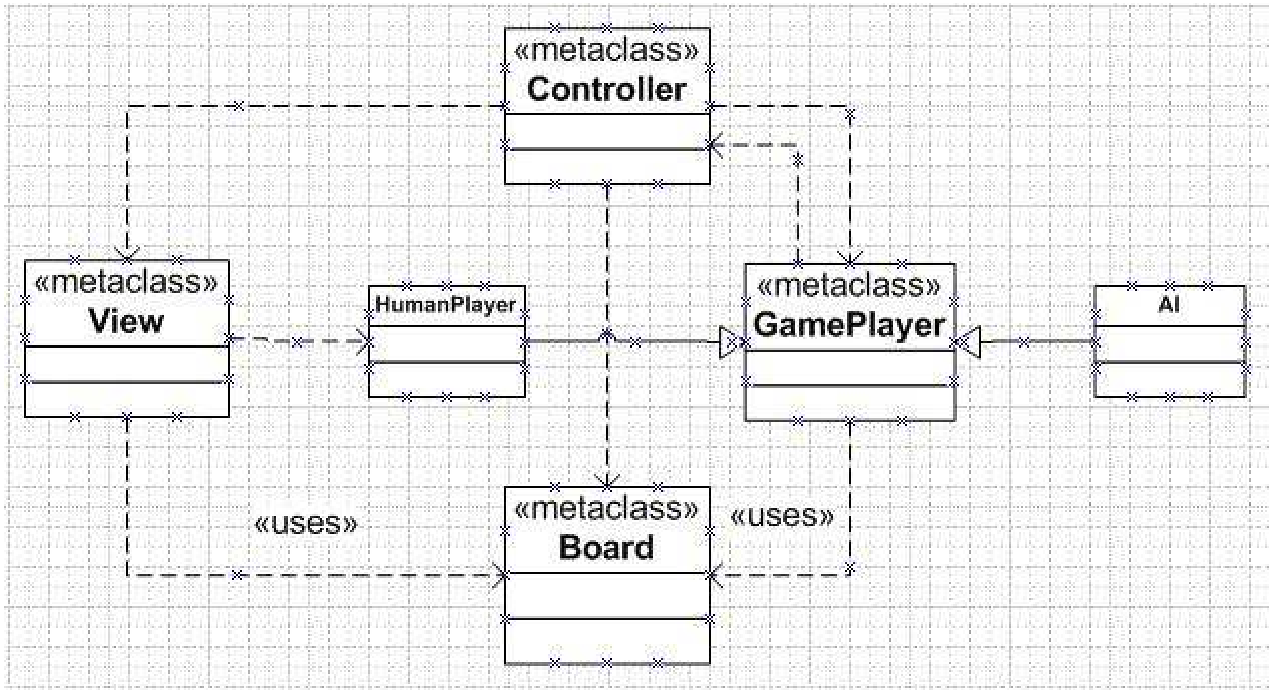

Figure 3: High-level MDD of the Antichess system

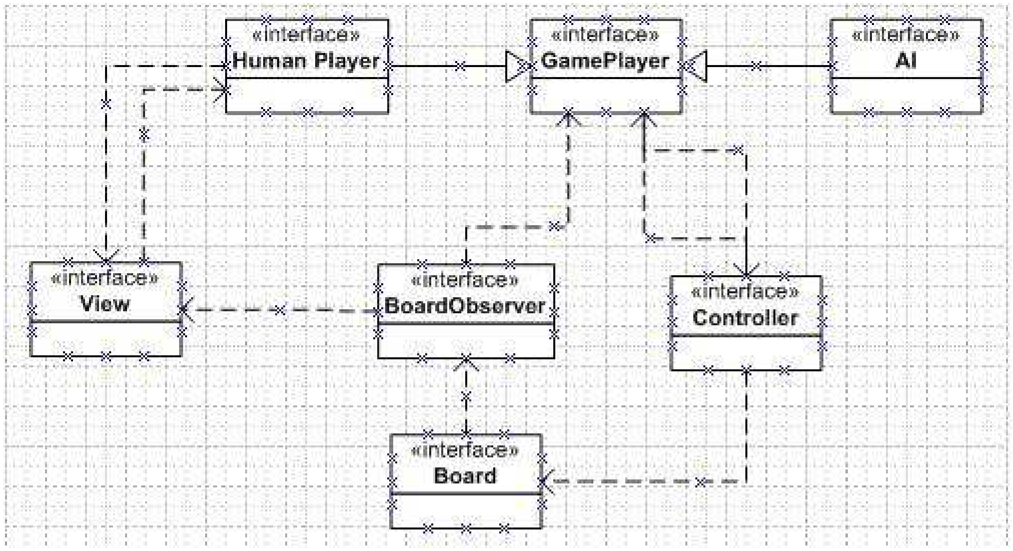

Figure 4: Dataflow between high-level Antichess interfaces