6.170 Final Design Document

Team 44 – "The Nihilists"

May 14, 2007

Requirements

Overview

Our system provides a framework for implementing two-player turn-based strategy board games and AIs, as well as interactive textual and graphical interfaces for playing both Chess and a variant of Antichess. It includes an artificial intelligence antichess player.

Both the graphical and textual interfaces allow either a human or the computer to play against either type of opponent. They provide for both timed games, with optionally different time limits for each player, as well as for untimed games.

Human players can make moves by interacting with the graphical board using the mouse. Games can be paused or stopped, and a new game can be started at any time.

User Manual

See `http://lunatique.mit.edu/~nelhage/antichess/manual.html` or `http://lunatique.mit.edu/~nelhage/antichess/manual.pdf`

Revised Specifications

The TextUI has several ambiguities and a few slight modifications. The specifications were ambiguous as to how to handle stalemate situations in the TextUI. A players turn is supposed to be skipped if they have no moves. With the TextUI most things require the user to explicitly tell them to happen. This is not the case with a stalemate scenario. If a player cannot make a move, his turn is automatically skipped. There is an extra '-s' flag for the TextUI which enables a ShowBoard command. The specifications

also mention that the first comman cannot be QuitGame. I think this is a terrible design decision, though lots of the parts of the TextUI are very unusable, but it has been implemented. Later the docs say that after a game has ended it only needs to support the commands available before a game has been loaded. Even though QuitGame cannot be a valid command before the first game has been started, one can still call QuitGame after a game has ended. For loading a game we do check if the board is legal. Because of castling and en passant it's important to know the history of a board. If the board is not legal it will not load properly.

Performance

Our system had to conform to one specified performance requirement: the AI had to be multithreaded and thus capable of running on a multicore computer, and it had to beat a version of itself running on a single core computer 51% of the time. This required us to find a way to improve our AI through the use of threads. Fortunately the typical Minimax search algorithm for game AI's was easily parallelizable, since we could assign different trees for different threads to search simultaneously.

Our AI faced additional non-specified performance requirements in order to work effectively. In the tournament the AI would play against another AI in a timed game, so it had to find and make moves in a timely manner. This meant that the function the AI used to evaluate each board in the game tree had to run quickly, as it would be run many times over, unnecessary game trees would have to be eliminated, and searches of the game tree would have to be optimized to find better moves more quickly and to avoid searching particularly bad moves.

In addition to these concerns, the underlying implementation of the chess board would have to be efficient and compact, since multiple AI threads would be replicating the board, performing and undoing moves on those board copies, and being queried for piece and potential move information frequently. A number of design aspects of the final `ChessBoard` and `AntichessBoard` classes were a result of these performance concerns.

Problem Analysis

The largest abstract model in the system is the Antichess board, encapsulating the current state of a game of antichess, including information such as:

- The set of pieces on the board and their locations

- The current player
- The state of pieces with respect to rules like *en passant* and castling.

In addition, the board must have an understanding of the rules of antichess, and support efficient generation of lists of legal moves. We also need some abstract model for storing the time state of each of the sides in a game of Antichess.

Finally, it must be possible for the AI to search the game tree by looking at the result of making all possible moves from a given position, which will require being able to make a move from a position, but still be able to retain or recover the original position.

It must be possible to maintain a list of all past moves from the game, for saving the game and for displaying in the UI.

We considered many design questions when constructing the board model. For example,

- Should pieces be their own objects, or simply managed entirely by the board they belong to?
- Should the board be immutable, making the game tree search and concurrency questions much easier, or should it be mutable, meaning we have to deal with an "undo" functionality and introducing concurrency issues, but possibly greatly improving performance?
- How generic can we make the "board" and "game" abstractions?
- Should the list of historical moves be maintained in the game abstraction, or as a separate model maintained by another layer of the system?

We ended up opting for a single fairly heavy-weight board class containing all the state of the board and the move history. The board abstraction is mutable, meaning that we had to deal with synchronization and with undoing moves (part of the motivation for storing the move history in the board itself). Since we did not benchmark the alternative (immutable boards), it's hard to say how significant the performance we gained was, but given the depth our AI is able to search to, it seems likely it was worth it. We did, in addition, as much as possible make the smaller component abstractions (pieces, move objects, and the move history) immutable whenever possible.

In addition to this data structure choice, we also designed our abstractions to be as general as possible. Recognizing that chess shares much in

common with many other games (e.g., it is a two-player turn based game with pieces that move or are placed on a rectangular grid), we implemented an abstract `Board` class representing a generic two-player game on a two-dimensional grid populated with pieces, implemented as much of our system as possible in terms of generics subclasses `Board`, and made `ChessBoard` and `AntichessBoard` subclasses of it. This choice means that the AI, for example, is almost entirely independent of the specifics of Antichess, and could be trivially reused to make an AI for some other game. The time state of each of the sides is stored in a separate class `GameClock` so it can be used in any game no matter what the type.

The Controller portion of the project presents its own set of problems. And by that I mean, synchronization. The Controller needs to manage asking for and accepting input from a variety sources, a TextUI, a GraphicUI and an AI. It then needs to handle actually performin the moves received.

This was solved by breaking the controller into two large pieces. The `GamePlayer` interface handles getting moves from a variety of sources. And the `GameController`, `ControllerState`, `ControllerMaster` trio handled the actual running of the game. The Controller portion get's moves from the `GamePlayer` by first asking for a move and then being notified by the `GamePlayer` when the `GamePlayer` has a valid move to return. This has the advantage of being asynchronous, so the operations of GamePlayers can be very independent of other and the Controller. This of course improves modularity. With our system, for example, implementing networked play would fit right into our system. Conversely this implementation has the disadvantage of being asynchronous. So with something like the TextUI, where inputs need to be processed by the Controller is a specific order synchronization needs to be set up.

The GUI presents all the problems a GUI typically does of trying to make a clean easy to use GUI that also provides all the functionality a user could want. One of the primary considerations for the GUI is whether to use a drag and drop system for moving pieces or a system where we click on the initial square and then the destination square. Since in real life we need to drag and drop pieces this seems like a more intuitive system. Drag and drop has its share of problems, though. First of all drag and drop is agravating on laptops. Second point and click soon becomes very natural even if immediately it does not seem so. Visual cues such as highlighting pieces can aid in this. Many of our potential users have played computer strategy games before. Nearly every computer strategy game relies on first selecting the piece and then selecting a destination. Mimicking this will actually make the GUI more intuitive for the users.

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 game players, including the AI.

The Board The Board component serves as the model in theMVC 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 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 The Controller accepts requests from the game players 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 Game Player The final major component of this system is the game player, of which there are two types: the human player, whose interaction with the program is facilitated through the View as described above, and the AI. On his/her turn each game player queries the Board for its current state and based off of this state decides on 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 move is illegal and responds accordingly to the game player. Due to its close interaction with the Controller, this component may be referred to as part of the Controller in the rest of this document.

The AI In the case of a human player the View moderates all exchanges between the player and the Controller, while the user does most of the thinking for the game. In order to reflect this thinking behavior

in the absence of a human player, an AI was implemented. The AI performs a Minimax search through the Antichess game tree, while employing a number of optimizations on this basic search to improve efficiency. It interacts symmetrically with the game player component of the system in exchanging information with the Controller.

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 means that while the class names are the same and some of the functionality is the same large sections have been rewritten a couple times.

The actual Controller module is broken up into two main sections which we shall call the Controller and the GamePlayers (sorry about redundant names). The GamePlayers handle getting input from a variety of sources while the Controller manages asking for moves, making moves, and the starting and stopping of games.

First we will discuss the `GameController` and the `GamePlayer` because their interactions drive the runnings of a Chess or Antichess game. The `GameController` is the class that handles requesting moves from the `GamePlayer`, making those moves, while managing time and checking to see if end conditions have been met for the `Board` that it is currently running a game on. The communication between the `GameController` and the `GamePlayer` works as follows: the `GameController` requests a move from the `GamePlayer` if the `GamePlayer` has a valid move it returns that valid move. If it doesn't it immediately notifies the `GameController` that it has no moves. The `GameController` then waits to be notified by the `GamePlayer` that is has a valid move. While this is happening it also checks to see if the `ControllerState` has changed or if end conditions have been met for the `Board`. Once it has a valid move it performs said move on the `Board`. The `GamePlayer` requested for a move is determined by the `Board`. If a `GamePlayer` has no legal moves the `GameController` automatically skips them. There is also the ability to pause the `GameController` loop at any point.

The `GamePlayer` interface represents anything that could be playing a game on the board. We have implemented a `ChessPlayer`, which represents a human player playing chess through the GUI. A `RandomAI` which was used

for debugging. An `AIPlayer` which obviously represents a computer AI. The `TextAIPlayer` and the `TextPlayer` exist as separate classes that both interface the `TextGamePlayer` interface because this simplified the TextUI, and because the communication of the `TextAIPlayer` and the `GameController` is fundementally different than the `AIPlayer`. The `TextAIPlayer` finds the move it would make, but must wait for the TextUI to tell it to give the move to the `GameController`.

The `ControllerMaster` is quite literally the master of the entire system. It handles the creation of new games, saving, loading and setting the state of the `GameController`. (This is handled through the `ControllerState` class, but we will get to that later. The `ControllerMaster` saves games by using the `GameWriter` class. It loads games by using the `GameReader` class. All of its methods are specific for the game it is currently running because their treatment of a few things necessitate different code. The `ControllerMaster` also provides a way for the View to control the state of the game or to start a new game, save a game, or load a game. Since during these operations it's important that the `GameController` state is controlled by the `ControllerMaster` (we wouldn't want the `GameController` making moves while the user was trying to save a game, there needs to be some sort of communication system between the `ControllerMaster` and the `GameController`. This is handled by the `ControllerState` class.

The `GameController` has a `ControllerState` that it notifies whenever it changes state. A `GameController` has three states, running, paused, and stopped. Stopped cannot be started again, paused can. Besides just monitoring the state of the `GameController` the `ControllerState` also can set the state of the `GameController` Because the `GameController` has its own thread this is handled by having the `ControllerState` set flags that the `GameController` ruitinely checks for. When the `GameController` notices that a flag has been set it does what the flag is requesting and then notifies the `ControllerState` that it has changed state. This allows the `ControllerState` to wait on a `GameController` until the `GameController` has entered the state that the `ControllerMaster` requested.

The Board

The `Board` and associated classses store the current state of a game (Chess or Antichess in our case), and is responsible for updating the state of the board in response to moves by players, and for evaluating the legality of moves and generating lists of legal moves to the AI and UI.

`Board`s contain `Piece`s, which are immutable and store state including coordinates on the board. `Board`s take action in terms of `Move`s. `Move` itself is abstract, containing only the `Piece` making the move. `ChessMove` extends `Move` for chess and antichess, and contains a destination square for the move, and the piece being captured (usually, but not always, the piece formerly located on the destination square).

The `Board` handles move legality by maintaining `MoveGenerator` classes for each type of `Piece`. A `MoveGenerator` knows only about the legal *types* of move for a piece – e.g. rows and columns, jumps for knights, whether a piece can capture or not, and so on. `MoveGenerator` deals with generating and/or testing move validity by this criteria, and then the `ChessBoard` or `AntichessBoard` ensures compliance with game-wide concerns, such as

- The King cannot move into check
- The King can't castle through check
- In Antichess, captures are mandatory

`AntichessBoard` is implemented as a subclass of `ChessBoard`, since most of the rules of the games are equivalent.

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

The View can be broken up into two distinct sections, the GraphicUI and the TextUI.

All of the operations for the TextUI are handled by the `TextUI` class and the two implementations of the `TextGamePlayer` interface, an extension of the `GamePlayer` interface. These two `GamePlayers` are the `TextPlayer` and the `TextAIPlayer`. The `TextUI` supplies the moves that the current `TextGamePlayer`, that the `TextGamePlayer` the reports to the `GameController` after the `TextUI` has deemed it appropriate. Because each operation of the `TextUI` needs to be done in order the `GameController` needs to notify the `TextUI` when changes have been made to it. The `TextUI` implements the `GameControllerObserver` method in order to do this.

The GraphicUI consists of a main class `ChessGUI` and several other classes which are components of the `GraphicUI`. The `ChessGUI` communicates with the master for saving, loading and starting new games. The

component that accepts inputs for moves and displays the board is the `BoardView`, for chess this is specifically the `ChessBoardView`. The `BoardView` is set to accept input by a `HumanPlayer`. A `HumanPlayer` is an extension of the `GamePlayer` interface. 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 `BoardView` handles loading and getting images through the `ImageMap` class. Selecting options for starting a new game and loading an Antichess game are handled by the `NewGameWindow` and `LoadGameWindow` classes respectively. The history of the current game is displayed in the `MoveHistoryView`. The times for each of the players are displayed in a `TimerLabel`.

The AI

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.

The AI is implemented by a generic `MinimaxAI` class, which implements minimax with $\alpha-\beta$ pruning. A `MinimaxAI` is created with a `BoardEvaluator`, which takes care of implementing the game-specific static evaluation at the leaves of search trees. Different AIs could be developed simply by using instances of different board evalation objects.

The Antichess AI uses a standard minimax search algorithm to search the space of potential moves. It implements the $\alpha - \beta$ pruning, 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.

In order to further improve move ordering (critical for efficient performance of the $\alpha - \beta$ cutoffs), it implements two more ordering heuristics:

The "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.

History tables Each AI maintains a map from moves to a numeric heuristic evaluation of its "goodness." Every time a move is found to be

"good," by either causing a β-cutoff or by being returned as a principal variation, its goodness is increased. In addition, all goodness values are scaled back at the start of each turn, to prevent stale, irrelevant values from dominating as the game goes on.

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.

The AI uses the *PVSplit* algorithm to split the game tree into multiple subtrees to be searched in parallel. It searches the first branch of the game tree serially, and then searches each remaining move in parallel. Search tasks are represented by `AIJob` objects, which are placed in a queue and then read out by worker threads. Each node being searched in parallel generates its own copy of the board, in order to avoid synchronization issues, and then atomically updates a copy of a shared `AIMove` object with the results of the search.

Module Structure

MDDs for the overall system and for the individual large components are attached in the appendix.

Testing

Strategy

We employed two types of tests in our testing strategy: automated tests and interactive tests.

Automated Tests

Our automated tests consisted of unit tests and predefined game files with known outcomes. We defined unit tests for classes in which there were easily testable non-trivial methods; these classes included the model and some of the view and AI components.

The model was easy and logical to test with unit tests, since the behaviors for these classes were necessarily predictable and it was critical to the system that they work just as expected. As a result most of the unit tests we have test the classes in the model. Other classes that implemented specific

features of our program, such as the GameReader and GameWriter classes, were similarly tested with unit tests.

This was less true for the other components of our system, since they either had relatively unpredictable behavior, tied parts of the system together, or could be tested by visual inspection of the working system. Thus, only a few additional components, including some of the view and Controller components and the AI board evaluation function, were tested using unit testing.

In order to test other aspects of our system, such as the functionality of the AI or the system as a whole, we used predefined game files with known outcomes. We ran these files with the TextUI in order to verify basic system functionality. We also used these predefined game files to verify some corner-case specifications of the Antichess rules, such as the precedence of the checkmate win condition over the losing all pieces win condition. Together, these automated tests were effective in covering most if not all of the specifications of our system's functionality.

Interactive Tests

A substantial portion of our testing strategy involved so called, "Angry Russian Testing," or user testing. From before the preliminary release, once we had a decent AI against which users could play, we had users play against our AI using our GUI. We refrained from commenting while the users used our programs and listened to their comments on both the UI and the AI. We then used these comments to make improvements and bug fixes to the UI and, to some extent, the AI. However we used another interactive testing strategy to make more AI improvements.

Our primary method of finding ways to improve our AI involved playing it against other AIs. Using both our own program and the provided Antichess program provided, we would run our AI against other AIs and our own AI and look for conditions that would hinder our AI from winning the game, such as blocked pawns. Using these results we implemented improved board evaluation functions, including functions that favored against blocked pawns and evaluated the end game differently than the early game, and better handling of the given time. While these interactive tests did reveal several bugs in our program, their primary use was to make incremental improvements to both our AI and UI.

Time Profiling

In order to improve the efficiency of the board we used time profiling on our board program to examine what methods were taking especially long and how we could improve these speeds. We used the Extensible Java Profiler on our ChessBoard and AntichessBoard test suites, which contained a self-consistency check that resembled the AI's usage of the AI and examined the resultant hierarchical time profile of the Board, which resulted in a number of critical efficiency improvements to the Board and, therefore, the AI. This step was not performed until after the Board had been modified for the ammendment and was passing all of our updated tests.

Test Results

As described above our tests resulted in a number of bug fixes and feature verifications in our code. Our unit tests verified most of the features specified in the specifications, although a few specification bugs were found later in user testing, so at this point we are confident that our program meets all of the specifications. (The specification bugs found through user testing were later tested thuroughly by new unit tests.)

There is still much room for improvement of our AI, although we have managed to make noticeable improvements from the AI we had for our preliminary release. We are still confident, however, that we have written a very decent AI that will provide the user with an enjoyable game of Antichess.

Due to time constraints we were unable to test the extensibility of our system as much as we would have liked. Our goal was to demonstrate this extensibility through the implementation of another turn based board game, such as checkers, with minimal additional classes.

Reflection

Evaluation

On the whole, our design worked out fairly well. The mutable boards caused a nontrivial amount of annoyance with synchronization issues between threads, and hassle with implementing undoing moves, but they let us make some fairly significant performance improvements, allowing our AI to perform often impressively deep searches of the game tree.

The MVC model, as expected, allowed for excellent modularity of components; After the preliminary release, we essentially completely rewrote the

controller portion of the system, without requiring any modifications to the rest of the layers (the AI, board, or view).

The Board's move generation and legality checking architecture was not an equally resounding success. While it did a reasonable job of encapsulating logic for the normal moves into the `MoveGenerator` classes, there was an occasionally annoying amount of code duplication and separation of logic when dealing with *en passant* and castling. In particular, the logic for accommodating the fact that in *en passant*, the pawn captures a piece on a square other than the one it's moving to is mixed between the `ChessBoard` and `PawnMoveGenerator`.

The use of generics to make as much of our system as possible independent from the specific game we are playing worked out well. Although we have not had time to do so, we could implement another two-player strategy game, such as checkers or Othello, complete with a UI and an AI, essentially only by implementing a `Board` subclass encapsulating the game rules, a `BoardEvaluator` to implement a static evaluation heuristic for the AI, and a `BoardView` that rendered it. The specific use of generics, however, resulted in some annoyingly ugly code in places, which could have been improved slightly.

Lessons

Having the `ChessMove` class store the captured piece for a move, and requiring that clients of the board correctly populate that field before making moves was a mistake. If clients had only passed in a move containing a start piece and an end square, the board could have asked the `MoveGenerator` if the move captured a piece, allowing for piece movement logic to be more entirely localized to the `MoveGenerator`s, and saving a lot of redundant checks for captured piece validity.

We had a fair number of issues with threading and synchronization of our threads. Our system would probably have been a lot more robust if we had sat down from the start and decided what should run in which thread, and what needed synchronization, instead of essentially each of us just adding threads to our portion of the project as we needed. Some issues with synchronization caused weird dependencies. The TextUI needed to be synchronized with the GameController so that it does not try to perform more moves before the previous move has been completed and the Controller has asked the next GamePlayer for a move.

While our generic system is really cool for its flexbility, it has some warts that could be improved. Many classes have to be generic on both the type of

the board, and the type of the moves legal for that board, even though the two are generally closely linked. Figuring out a means to fix that problem might have been nice.

In addition, we originally made the `MoveGenerator`s a field of the `PieceType`, since that conceptually made sense. However, doing so would have required `Piece` to be generic, or to use a specialized subclass of piece for everything, which would have seemed awkward. Hence, we moved the `MoveGenerators` into a map in the `Board`, which means that `PieceType` is now essentially just an enum; We could have either found a way to fix the generics issue so that having `MoveGenerators` in the `PieceType` was easier, or else have made the `PieceType` into a true `enum`, and eliminated the now not-very-useful `PieceTypeFactory`.

Known Bugs and Limitations

Our system currently meets all specifications, to the best of our knowledge.

Appendix

Formats

Module Dependency Diagrams

There are two types of arrows drawn in the MDDs. A solid line that terminates in a closed arrow head indicates that the class at the tail of this arrow is an implementation or extension of the class at the head of this arrow. A dashed line that terminates in an open arrow head indicates that the class at the tail of that arrow depends on the class at the head of that arrow. An optional tag above a dashed line describes the relationship between the class at the tail and the class at the head of the dependency arrow. In addition a description tag inside of a class block describes what that class is or what that class's function is.

All arrows are unidirectional. In some cases multiple arrows will overlap each other, such as if some class near the top of an MDD depends on a number of classes below it in the diagram. (See the `ControllerMaster` class in the 3. In some cases this may cause the dashed to overlap and form a solid looking line rather than a dashed line. In these cases it is easier to distinguish the type of line it is by the arrow head.

Module Specifications

See `http://lunatique.mit.edu/ nelhage/antichess/api/`

Team Evaluation

We also had a major success outside of our code. We realized the night before our final project was due that, relative to many of the other groups, we had very little necessary work to do on our project and were able to spend substantial time refining components of our existing system while completing the last parts of our final project submission. When we asked ourselved what caused this relatively unpaniced state of our project, we discovered this success.

Throughout the course of this project our team got along remarkably well and were motivated to work on this project. We formulated our overall design before the official specifications for the project were released and began structuring interfaces for our program in order to help us visualize how the final system would work together. In addition to this early design, we established a repository system, a mailing list, a zephyr class, and a regular time that our group would attempt to meet each week to work on the project before the TA's provided or requested these things from us. We delegated who would work on what components and when we wanted to have our system at various stages of functionality at this first design meeting, and communicated set backs and successes freely and frequently with everyone in the group, since we were friendly enough with eachother to do so. This early initiative and sustained momentum accounted for much of our team's success.

We also discovered that, although we were a relatively small group, none of us felt like we individually did that much work for this project, no matter how many lines of code the system reported that we wrote or how much work other team members would comment that each of us did. This testifies to the effectiveness of our group due to the friendly relationship of the team members with each other and the effective delegation of work and communication of problems with our team as a whole.

We also employed a number of methods for maintaining group morale during particularly difficult weeks or tooling sessions that were close to deadlines.

Acknowledgements

We would like to thank the following for their contributions to the success of our project:

dvp for providing crazy russian testing.

jmcquaw for bringing us food when we were hungry but wouldn't admit it.

jscohen for being an excellent game tester and exposing a bug in our code.

kpugh for raising our spirits.

Margret Thatcher

mitchb for introducing us to the concept of "Angry Russian Testing."

Oreo it's a kitty!

Thailand Cafe and Sicilia's for bringing us food when we were hungry and would admit it.

The Dude

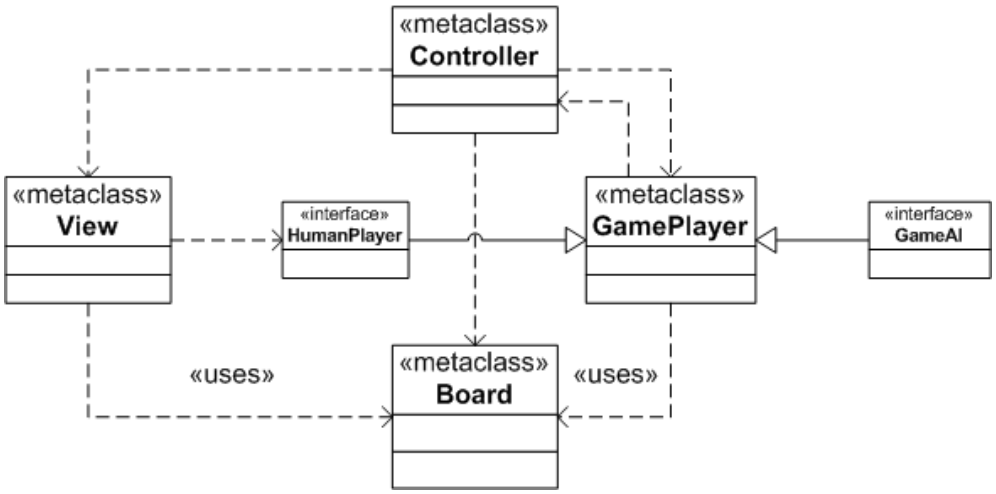

Figure 1: Overall System Dependency Diagram

Figures and Diagrams

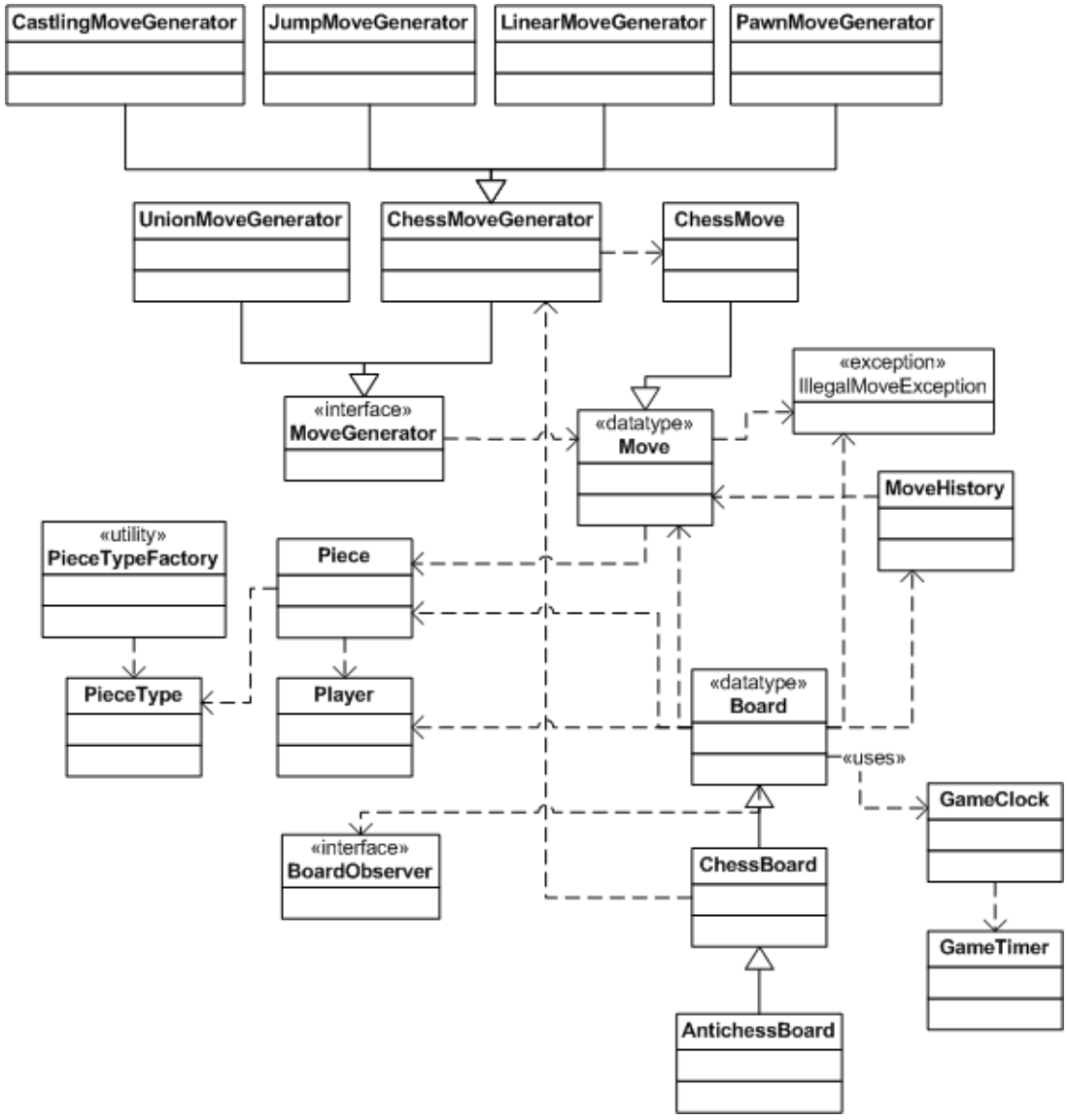

Figure 2: Model MDD

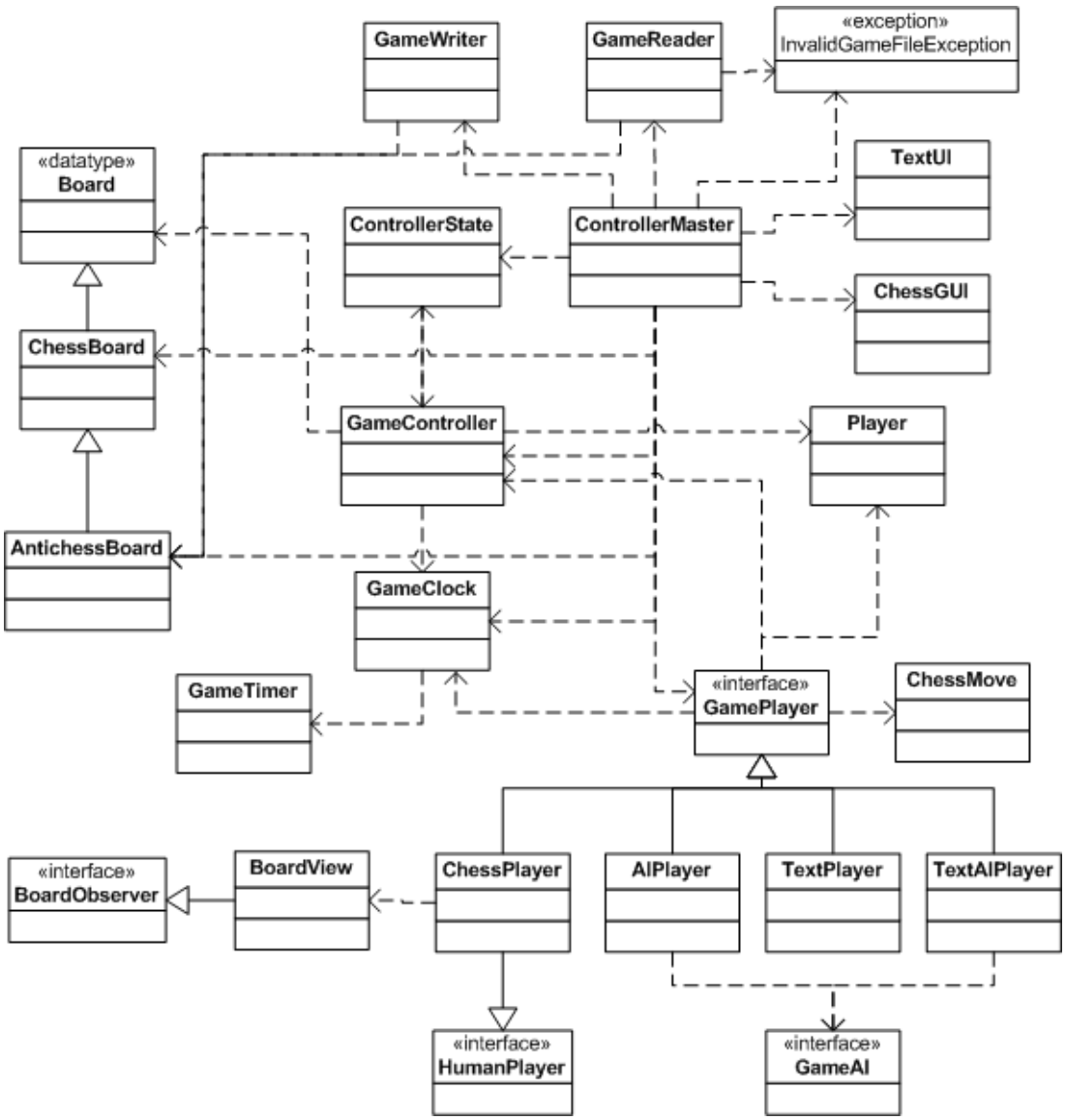

Figure 3: Controller MDD

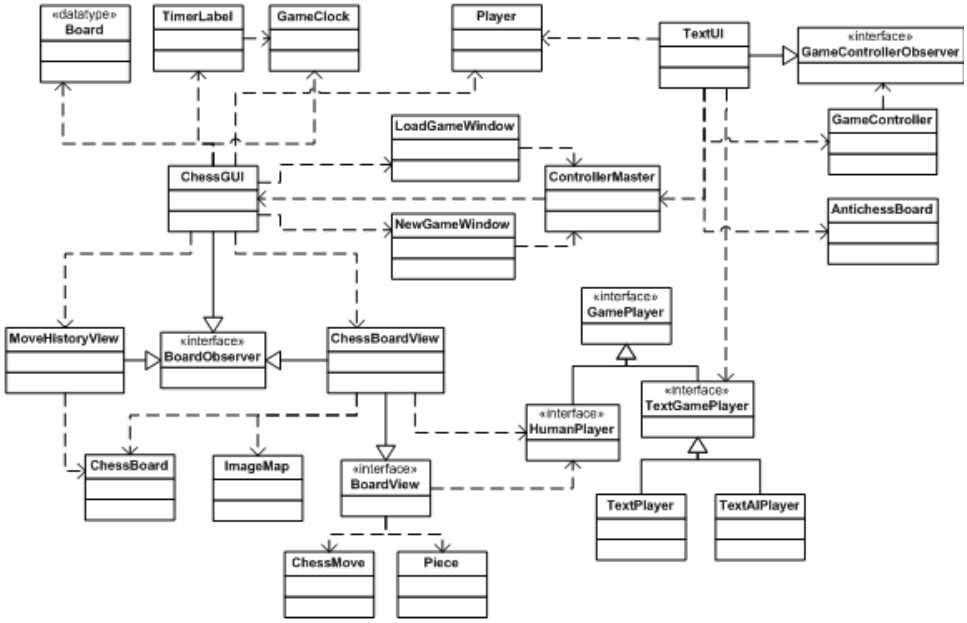

Figure 4: View MDD

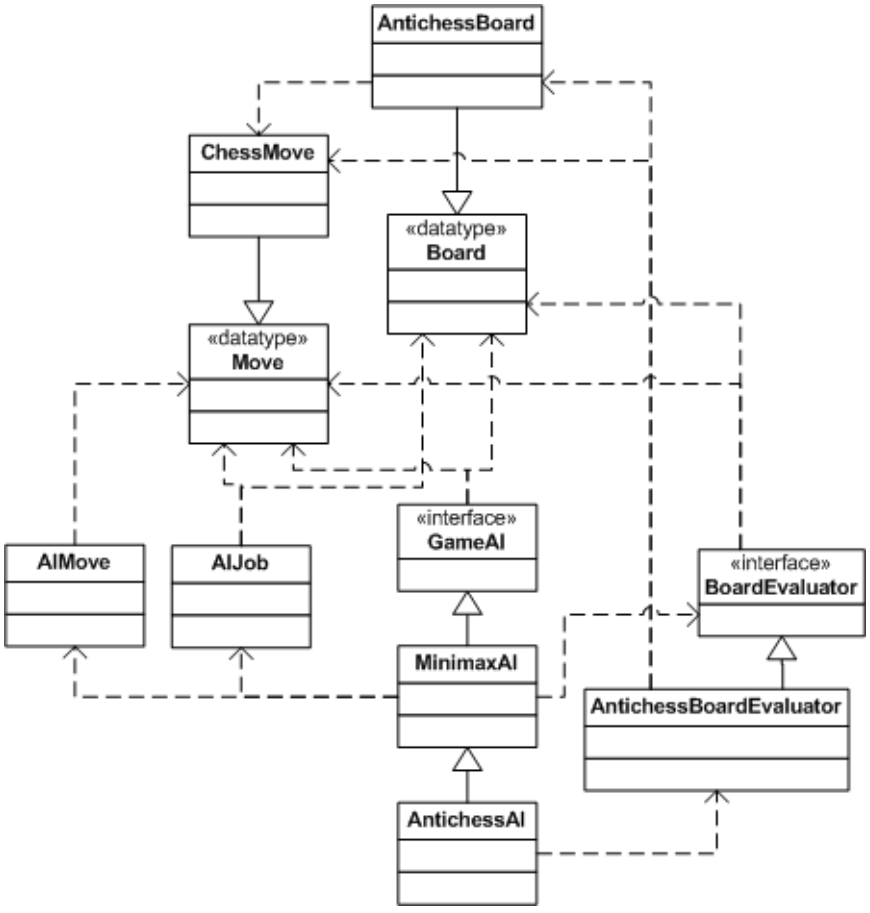

Figure 5: AI MDD