Pawned Game Execution Environment
Final Documentation

Julián Hernández
Rodrigo Ipince
José Muñiz

May 14, 2007

Contents

Chapter 1

Requirements

1.1 Overview

Pawned 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. Antichess is played between two opponents by moving pieces on a square board. The board is composed of 64 squares. The eight vertical lines of squares are called *columns* and the eight horizontal lines of squares are called *rows*. The squares are colored black and white alternately as seen in Figure 1.1.

The following document assumes familiarity with the rules of *6.170 Antichess*. For a complete description of the *6.170 Antichess* rules, refer to Appendix A.

Pawned is playable as a standalone application. Its basic mode of game is to have a human play against a machine player using a Graphical User Interface (GUI). Additionally, *Pawned* provides a standard interface to engage in multiplayer tournaments with other *6.170 Antichess* players. For a complete specification of this interface, refer to Appendix **??**.

Pawned's computer player does not simply make arbitrary moves, but rather bases them on decision heuristics that enable to play competitively, using multicore computing capabilities, when available.

The GUI has the following characteristics:

- It contains:
  - A graphical representation of the current state of the board
  - The name and color corresponding to each of the players
  - The player that should move the piece next
  - The remaining time on each player's clock
- It allows:
  - Graphical input mechanism. That is, a way for the user to input their moves using a mouse pointer

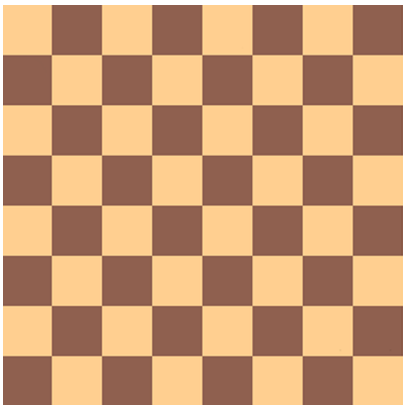

Figure 1.1: A standard board for playing Antichess

- Move validation and real time board status update
- Easy access to information about the last move executed
- Initiation of a new game
- Storage and retrieval of games. For the full documentation of the format used by *Pawned* to save games, please refer to Appendix **??**
- Termination of the current game

Finally, *Pawned* provides a Command Line Interface (CLI) as specified in Appendix B.

1.2 Revised Specification

Pawned supports extensibility in several ways. For that reason, *Pawned* should be seen as a *Game Execution Environment* , and not merely a *6.170 Antichess Player*. The latter is just one game that is runnable within *Pawned*.

1.2.1 Game Execution Environment

Pawned allows the execution of a board game: a sequential activity between two players who can manipulate pieces in an n-dimensional board according to a fixed set of rules. At any given point, the game has one of two states: either running or terminated. This termation can occur as a result of several conditions and may be of different types. For example, in *6.170 Antichess*, an example of a termination could come from a player being checkmated. Additionally, the game has a set of messages. These messages indicate game-specific information about the current state of a game. This information can vary from, for example, a message indicating that a current game of *Chess* is in check, or any other relevant game information

A game can have observers. An observer can see the game dynamics while they happen but cannot participate (i.e. cannot manipulate pieces in the board). A player can be thought of as an obsesrver who has the additional faculty of manipulating pieces.

A game can be timed by a pair of timers, which limits the amount of time each player has throughout the game. If a player runs out of time, the player loses the game.

1.2.2 6.170 Antichess

6.170 Antichess is an example of a game that can be run in the Game Execution Environment. It consists of two variants: *Standard 6.170 Antichess*, and an extended edition of it, *6.170 Antichess with Encastle*, which is similar to the standard version, with additional support for *en passant* and *castling* moves within the game. A complete analysis of the rules and distinctions between both versions of *6.170 Antichess* is included in Section 1.1 and detailed in Appendix A.

The players for *6.170 Antichess* can be either two human players using the same interface, one human player using a playing interface against a computer player, or two computer players being observed by a playing interface.

Playing Interface

Pawned allows the users to play antichess in a Graphical User Interface (GUI) that shows the board and pieces. The revised GUI looks like that on Figure 1.2. It consists of a main panel, a toolbar, four menu bars, and a status bar. The main panel displays the following information:

- A graphical representation of the board and pieces. A numbering appears next to the board, allowing the user to easily reference a cell in the board.
- The name of the players along with the color of the pieces they are using.
- A timer for each player that shows how much time left a player has to complete all its moves. If no time limit is selected, the timers are not displayed.
- A move history displaying all the moves that have been made througout the game. The moves are displayed in extended chess notation format.
- The pieces each player has captured.

The main panel resembles that on Figure 1.3.

The GUI indicates which player has the turn to play by highlighting his name, color and timer (if playing with time limit). The possible moves a piece can perform are highlighted when a user puts the pointer on it.

The four menu bars have the the following menu items (menu items with "?" are still under consideration):

File New Game, Save Game, Load Game, Quit

Figure 1.2: *Pawned*'s Graphical User Interface

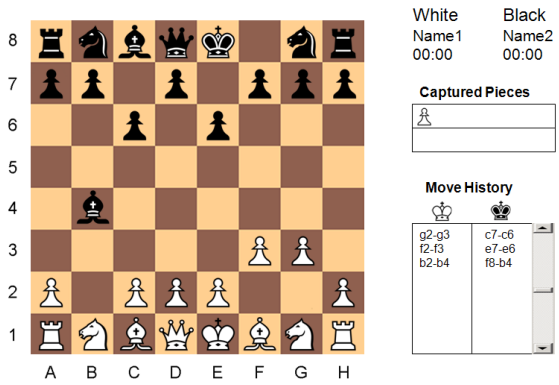

Figure 1.3: *Pawned*'s GUI Main Panel

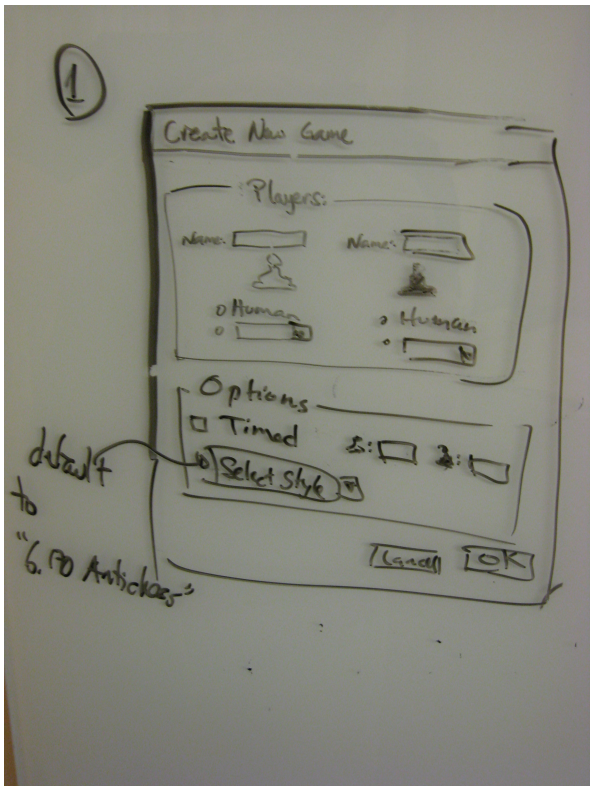

Figure 1.4: *Pawned*'s New Game Window

Game End game

Help User's Manual, About...

The **New Game** option opens a window resembling that on Figure 1.4 where the user has to select the properties of the new game. This window lets the user enter the names of the players, and select if each player is Human or Computer. If the Computer player is chosen, the user can select the intelligence level of the player, from among several options. The user also has to select between a game with "Unlimited Time", or enter the desired time limit for each player. Finally, there is an option to select the type of game (i.e. *Standard 6.170 Antichess*, *6.170 Antichess with Encastle*, etc).

The **Save Game** option opens a file chooser window that lets the user select where the game file is going to be saved.

The **Load Game** option prompts a window similar to that of *New Game*. The user is able to select the players that will resume the game, as well as select a route to a valid *Pawned* game file. After selecting a game to load, the user has to select the players that will resume the game.

The **Quit** option exits *Pawned*.

The **End game** option allows the user to end a game without exiting the application.

The **User's Manual** option opens *Pawned* users' manual.

Finally, the **About...** option shows information about the *cool* creators of *Pawned*.

The operation mechanisms for the Command Line Interface (CLI) to enable the user to select their preferred game variant when a new game is started are detailed in Appendix B.

1.3 Performance

Pawned's *6.170 Antichess* AI player makes use of parallel computing. This player running on a multicore processor beats the same AI player running on a single core processor at least 51% of the time.

1.4 Problem Analysis

1.4.1 Execution Environment

In order to fulfill the requirements outlined above, *Pawned* defines an abstract **player** that can provide a *ply* upon request. A ply is a set of actions, such as moving or adding a piece to the board, that constitutes a player's move for its turn. For example, in regular chess, castling is a ply that consists of two moves.

These players are able to interact with a **controller**, who is in charge of finalizing a ply, and executing it. The controller is also in charge of notifying the appropriate player that it is its turn to play, and notifying the players if a termination condition has been met or if a player has run out of time, in which case this player would effectively lose. Additionally, the controller is responsible for creating, loading, and saving games. The controller can be thought of a referee that mediates communication between the players and the game itself.

Finally, the last component is the actual **game model**, which holds all the relevant information about the game. This includes the status of the game board, the history of the plies, and the *rule set* being used. The rule set contains the set of global rules (i.e. the set of rules that govern the movements of any piece) along with the types of pieces supported and how each piece can locally move. The game model can then validate any move and determine whether there is a winner by using the rule set.

A rule set for a game consists of five parts:

- A set of *global rules* that govern the movement of all the pieces in the game. For example, a global rule in *6.170 Antichess* is that if a player is in check, then it *must* perform a ply that will put him out of check.

- A set of *local rules* that specify the valid plies of a set of supported pieces. For example, a local rule in *6.170 Antichess* is that a bishop can move diagonally to

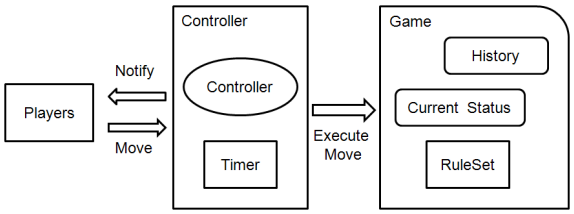

Figure 1.5: Component diagram

an empty cell or one that contains a piece of another color, without jumping any pieces.

- A set of *termination conditions* that specify whether a game has been finished. This termination could be a result of a game having a winner, a game resulting in a draw, or other termination. For example, a termination condition in *6.170 Antichess* is checkmate.
- A set of *game messages* that reflect events in the game. 'Check' is an example of one of such messages in *6.170 Antichess*.
- A *turn management* mechanism to determine the next turn, i.e. the rules governing who should play after a move has been executed by a given player. For example, in *6.170 Antichess*, the players should play in an alternated fashion, except if a player is in stalemate.
- A set of *supported boards and plies* for the particular game specified by the rule set. The rule set possesses information about the initial board configuration for the game. For example *chess board* would be the supported board for *6.170 Antichess*, with the initial board configuration as specified by Appendix A.

See Figure 1.5 for a visual depiction of these components. The *Factories* component refers to the set of all possible plies and boards supported for a given rule set.

1.4.2 Considerations

Handling plies

Even though all the players have access to the board's status at any given point in time, any player that wishes to perform a ply must do so through the controller. This will ensure that the board does not get changed when it should not. If an artificial intelligence (AI) player wants to simulate plies on a board to choose its next ply, then it must do so on copies of the board.

The controller notifies each player when a ply has been executed.

Controlling the game

When the controller decides that it is someone's turn, it notifies the player that a ply should be provided. The controller will then wait for this player to submit a valid ply, prompting it again in case the player submitted an invalid ply.

Once the player submits a valid ply, the controller notifies the model and delegates the turn to the appropriate player. In the case that the move just executed results in a termination state (e.g. in the case that a player has been checkmated in *6.170 Antichess*), the controller's behavior is specified by Section 1.4.2.

Determining a finishing condition

A game finishes when a termination condition has been met, including the possibility that a player has run out of time. In the case of *6.170 Antichess*, these conditions are checkmate, win by depletion of pieces, or double-sided stalemate, as specified by Appendix A. When one such event has occured, the game notifies the controller immediately after the terminating ply has been executed. The controller then notifies each of the players of the termination condition. The controller notifies the players and observers when the time for the current player has been depleted. These in turn act in the way that they deem appropriate (e.g. a GUI might display the termination information, or an AI player might stop computing successive moves, if any).

Since the game model is in charge of holding the status of the game, it will always contain the information regarding whether the game is running or if it has terminated, along with its termination condition.

Ply validity

As was stated before, the rule set for a game consists of five parts. Given a rule set, a ply is valid if it satisfies all the rules (unless a global rule overrides a local rule, in which case compliance with the global rule deems the ply valid).

Local Rules These specify the valid plies of a set of supported pieces. They are piece-specific. A piece has knowledge of its valid plies given its location on a board.

Global Rules Given the union of all the local rules for all pieces, the global rules add or remove valid plies.

Since the controller needs to check a ply for validity before executing it, and the players might want to view the valid moves frequently (for example a GUI determining all the valid pieces for display to the user), *Pawned* provides this validation in constant time.

1.4.3 Architecture

Pawned's game execution environment uses the following abstract modules. See Figure 1.6 for a visual depiction of these modules.

Piece A Piece represents a game piece, whose state and behavior depends on a RuleSet and a Board. A Piece can contain information about its color (black or white), type, initial position as defined by the RuleSet, as well as other information it finds necessary.

A Piece can be created in two ways. In both ways a color for the Piece must be specified. If a Board is also given, then the new Piece will add itself to the Board at a default initial position (defined by the RuleSet). If all the possible initial positions are occupied, an invalid position message is returned to the invoker. If a Board is not specified, then the Piece is created but it is not contained in any Board. It can subsequently be added to a Board at a specified cell position.

A Piece can provide information about its color, type, and the Plies it can perform (given a Board). These Plies are only those which conform with the local rules specified in the RuleSet. Notice that *all* the knowledge about a valid Ply must lie within the Piece itself; the Board will not assume anything except that no two Pieces can occupy the same cell. In this sense, a Piece has to have the information of whether it can capture a Piece of its own color or not (allowing addition of Pieces such as the AntiKing).

Notice that some rules might force the Pieces to be able to determine information about other Pieces in a Board. For example, the rule of castling in Chess requires a King to determine whether a Rook has been moved during the game. To accomplish this, a Piece has a mechanism that allows it to receive a notification every time a Ply has been executed in the Board. It can then retrieve what it needs to update the information it cares about. In some cases no information is really needed, such as for a Bishop, but this functionality allows *Pawned* to easily support any Plies that involve this type of information.

This poses another problem: each Piece now holds important information that needs to be recovered when a game is loaded. Thus, *Pawned* is able to load a game that was saved by reproducing the information by executing all the Plies specified in its move history. A deeper discussion of the alternative mechanisms for holding state information and the ruleset - piece interaction can be found in Section 2.2.1.

Board A Board provides the representation of the gaming board. It consists of a set of cells that may contain at most one Piece. The cells are referenced via a group of n indices. In this sense, a Board represents an n-dimensional board. *6.170 Antichess* uses a Board where $n = 2$.

A Board serves simply as a cell structure that contains Pieces. It makes the follwing assumptions:

- No two Pieces can occupy the same cell.
- A Piece cannot occupy two cells at the same time.
- When a Ply is executed, a Piece occupying the ending cell of the Ply will be captured and removed from the Board.

However, it does not assume anything about the Pieces it contains or about the game that is being played.

It can determine the position of a given Piece, get all the Pieces of a given color that are currently in the Board, execute a Ply and notify the Pieces that are currently in the Board, and clone itself. When executing a Ply, it returns a set of captured Pieces.

Pieces can also be added and removed from a Board outside the context of a game. This allows for Boards to be arbitrarily set in any fashion.

Move A Move represents a transition from one cell to another within a Board. In other words, a move contains information about the starting and ending cell positions in a Board. A Move can be created by specifying a starting cell position and an ending cell position.

Ply A Ply represents a sequence of actions, that can be either Moves, Removal or Additions. A Move is defined previously. A Removal is an action by which a piece previously present in a board is no longer registered as contained by the board. An Addition is an action by which a piece not previously present in a board is registered as contained by the board.

Game A Game constitutes the central mechanism for game control. It contains information about the game rules using a RuleSet. Additionally, it contains a Board representing the current configuration along with captured pieces and ply history, a set of possible Plies and the current Turn, State, and Messages.

As previously specified, States can either be active or terminated (along with its Termination condition), and Messages can be any type of condition that is occuring in the game, such as the *check* condition, in *Standard Chess*.

A Game can be created from an initial Board configuration, along with a RuleSet for the specified game.

A Game provides

- Information on whether a given Ply would be valid considering the current configuration and turn. Because this mechanism is assumed to be used often, it should be as efficient as possible.
- A mechanism that informs the caller about the current Board, Turn, State, and Messages upon request.
- A mechanism for providing the game history (a sequence of Plies in the order they were executed since the beginning of the game).
- A mechanism for retrieving captured pieces for each of the players.
- A mechanism for executing a given Ply and provides information of who should play next. It should also notify the invoker if the Ply resulted in a termination condition.

Controller A Controller is the mediator between the Players, Observers and Game. It holds information about the Timer, as well as the Players and Observers currently engaged in the game.

A Controller can be created by providing the two Players, a RuleSet, and a Board. The Boad will be setup using the list of pieces supported by the game, as provided by the RuleSet. Since each piece knows where to place itself, this process should be unambiguous.

A Controller provides

- An asynchronous messaging system for the suscribed Players and Observers for notification regarding addition of Messages, States, Terminations, and Time Depletion.
- A messaging system for notifying the subscribed Players about their turn
- A system for managing observers in the observer list
- A game saver
- A Controller Factory for creating controllers from previously saved games
- Execute moves on the game

Observer An Observer is the most basic subscriber to a controller. It should contain a mechanism for subscribing to the Controller's messaging system on notifications.

Player A Player is a type of Observer. He should also provide a mechanism for receiving messages from the turn delegation system provided by Controller.

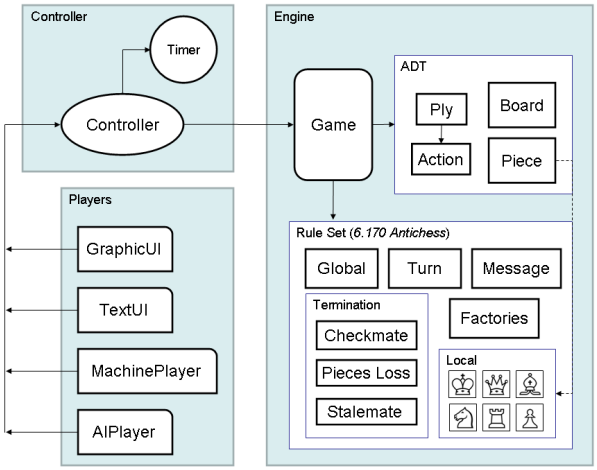

Figure 1.6: Conceptual diagram

Chapter 2

Design

2.1 Overview

Section 1.2 introduced the overall abstract components of the system. The design is based on this principle to provide an extensible gaming platform, fully implementing the requirements for the particular behavior of *6.170 Antichess* with ease.

In order to fully appreciate the extensibility provided by *Pawned*, it is useful to consider the execution environment as separated from the implementation of *6.170 Antichess*. With this environment, it is possible to describe a broad range of sequential games using the execution environment as a platform.

2.2 Game Execution Environment

The game execution environment publishes a set of interfaces that enables a Java programmer to fully implement a sequential board game by indicating the rules and pieces for their game. Based on the functionality of these implemented interfaces, the *Game Execution Environment* is then capable to execute a game successfully. The structure of this platform can be divided into four independent layers.

Controller Layer The controller layer provides the asynchronous messaging turn request and notification system. The abstract *Controller* defined in Section 1.4 is contained within this layer, along with the timers and asynchronous messaging utilities required to query and inform users of incoming events.

Game Layer The game layer provides the mechanism for board-player interaction under the rules and regulations of the rule set. The abstract *Game Model* defined in Section 1.4 is contained within this layer

Programmatic Interface This layer contains the interfaces and abstract classes that need to be subclassed in order to implement a new game. They describe the basic rules, boards, and pieces that are allowed in the game, and provide a contract with the *Game Execution Environment* that permit the operation of

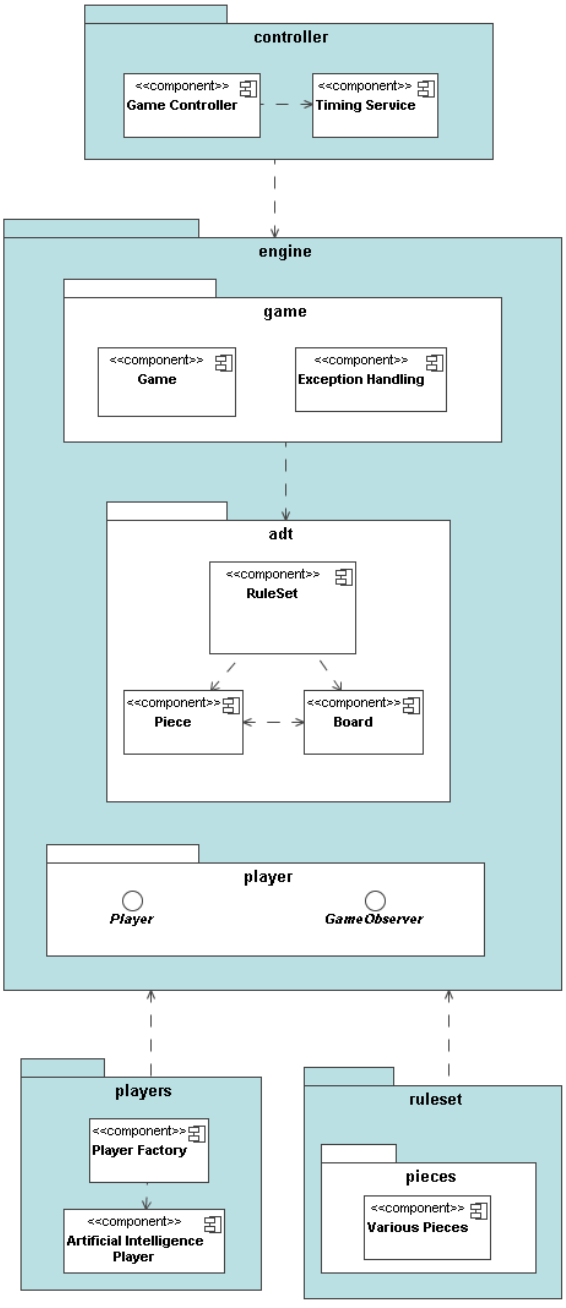

Figure 2.1: *Pawned*'s Implementation Diagram

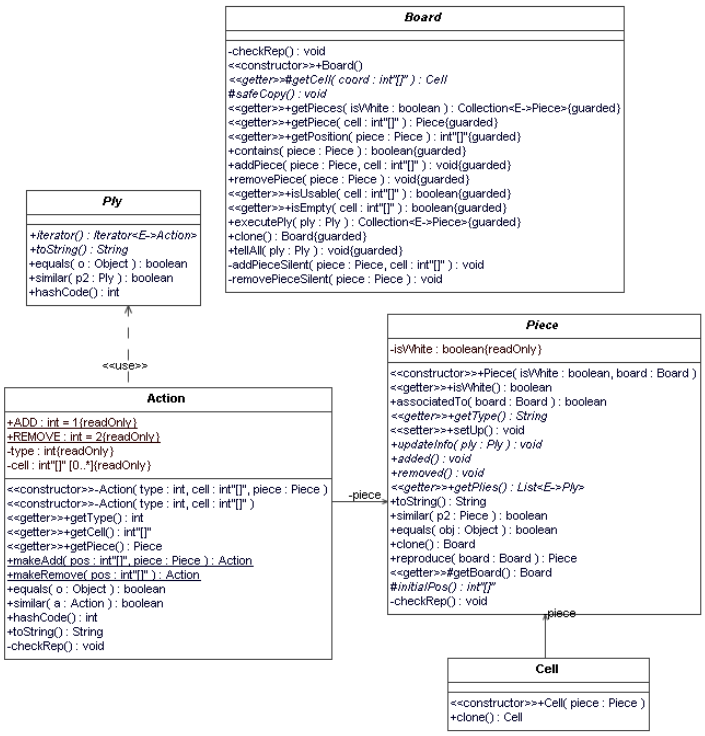

Figure 2.2: Board-Piece-Action-Ply interaction

the top layers, along with the specification of the contracts that objects have to fulfill in order to participate in games, via the **Player** and **GameObserver** interface.

Developer Utilities A set of prefabricated pieces, boards, and players that conform with the specification of the published *API* and that can be used in the development of games.

These layers, along with their general modular dependencies, are presented as an implementation diagram, in Figure 2.1, along with the modular structure of the application.

2.2.1 Programmatic Interface

Contents The programmatic interface layer is found in the `engine.adt` package. It defines the contract between the game developer and the execution environment. Its components classes are `Action`, `Board`, `Parser`, `Player`, `Ply`, `Piece`, and `RuleSet`. See Section 2.2.3 for a discussion of `Player` and `GameObserver`

Board-Piece interaction for sequential games

There exist several different ways to implement a piece manipulation scheme within a board. For example, an implementation for `Board` could contain methods such as `makeMove(int[] from, int[] to)`, `castle()`, etc. The board would then be in charge of determining whether the moves are legit, and executing them. However, this would mean that the board would be forced to be eternally coupled with a particular rule set, and it would be unpractical for any game-neutral agent to translate generic moves into game-specific method calls to a particular board. For that reason, such an implementation would greatly harm extensibility.

Another possibility could be for each piece to contain the board they are currently in, and then to ask each piece for the information that is relevant to them. For example, a king in *6.170 Antichess with Encastling* would query a rook for information on whether he has moved. Then, based on this information, the king would determine whether he can or cannot castle. However, since a board that holds different types of pieces can only hold a general object of type `Piece`, and not particular subclasses, each object of type `Piece` would require, in this example, to contain an implementation for `boolean hasMoved()`. Once more, this solution is impractical for a generic game environment, where each piece could require arbitrary types of information about the other pieces and their movements.

Yet another alternative would consist of having only a global ruleset, in charge of computing all the possible moves given a board configuration. This option brings the advantage that only one instance of each piece type would be required, since pieces would not be required to hold any game-specific state. However, the desired states for any given type of piece would still be held inside the rule set. Then, all the information regarding each piece would be stored in a single location, making it difficult to maintain and extend. During development, a benchmarking scheme was devised, in which several pieces with certain information were instantiated. Afterwards, the performance in time and memory was compared with the creation of a single object storing all the information. The result threw no significant difference between both methods.

For these reasons, *Pawned* requires the pieces to be aware of the state of the board, and then compute their valid moves, in the way introduced by Section 1.4. The design and implementation details for these scheme are presented in the following sections.

Board

Description An object of type `Board` represents an N-dimensional game board. That is, a set of cells arranged in some order. A `Cell` is a simple data type that in a board may contain at most one `Piece`. To refer to a cell within the board, its position must be specified. A cell's position is determined by n integers, which specify the coordinates of the cell in an n-dimensional board.

It should be noted that not all possible combination of n integers are necessarily associated to a cell. If a sequence of n integers $(k_1, k_2, \cdots, k_n)$ is unusable.

A board contains a set of pieces, each inside of some distinct cell. A usable cell

might thus be empty or occupied by a piece. If a cell is usable and empty, then a piece can be added to this cell.

A board can execute valid `Ply`, a sequence of additions and removals of pieces within a board. (See Section 2.2.1). A ply is valid with respect to a `Board` if the set of actions it defines can possibly be executed given the current board configuration.

Class type choice `Board` is an almost fully implemented abstract class. Most of the operations can be implemented in terms of the `Cell getCell(int[] coord)`. A distinct board can be built by implementing the previously mentioned method. In this way, an appropriate cell fetching mechanism can be used to retrieve a particular cell. For example, in the case of a chess rectangular board, the representation for a board could consist of an eight by eight array of cells, whereas a very large board could use other mechanisms for storage, such as a hash table. With this design choice, boards can also have any arbitrary shape, dependent on the implementatino of the particular `Cell getCell(int[] coord)` method.

A mapping from pieces to its coordinate is kept in the abstract class in order to implement the `int[] getPosition(Piece piece)` method efficiently. This optimization is very useful, since a game ruleset and the pieces themselves are expected to query the board often for this information.

Important features and design choices A `Board` is synchronized. This allows the board to be accessed in an asynchronous fashion. For *Pawned*, this is a necessary condition, since players and observers could potentially request and read information from the board while the controller is executing moves on it. Additionally, the set of pieces of a color that can be retrieved by the board's `Set<Piece> getPieces(boolean colorIsWhite)` will not change even a ply is executed, in order to ensure that a caller can safely iterate over it.

A `Board` is clonable, in a deep copy sense. A deep copy is required in order to ensure that board manipulations in a copied board are independent of the initial board, so the boards cannot share a reference to a same object, since in doing so, any modification in one board would have side effects on the other. The implementation of this functionality is non trivial at the abstract class level, since in order to deep copy, a board would require information about its representation, which is not readily available in this case.

A possible solution for this situation could be to utilize *Java Reflection* to dynamically obtain the fields from the subclasses, and set the copy's fields to be clones of the initial fields. However, this would require all of the components of the extending classes to be clonable, and would also require a very stringent contract between this class and its implementing classes. Therefore, in order to solve this issue, each board that extends `Board` must clone its representation on request. Since each implementation should know how to deep copy itself, the clone method of the generic board can rely on these side effects to successfully deep copy the board. After it has done so, the board can be safely cloned. Since the latter method is less error prone and requires less processing, this is the method chosen for *Pawned*. Although the method is protected, therefore exposing it to potential external unwanted calls, the

effects on a board would not disrupt its normal functioning, since the representation held after the method call would represent the exact same state as before. Therefore, there is no danger in exposing this method.

The board implements a notification mechanism for its subscribed pieces. This notification mechanism informs the pieces whenever they are added or removed from the board, and also whenever a ply is executed in the board the pieces currently populate. With the use of the *Observer* pattern in this case, pieces can gracefully receive notifications that they can use in order to process their corresponding valid plies. For example, a king in *6.170 Antichess with Encastling* would be able to monitor its rooks, keeping a record on whether it is allowed to castle or not. With the ply notification system, the king can notice when the rook (or the king itself) has moved, and then update its state to no longer publish castling as an allowed move, in case all the other castling conditions apply.

In order to achieve this functionality, pieces must be associated with a single board. There are several reasons for this requirement. On one hand, a piece can only be playing a game in one place at a time. If no association were required, the piece could potentially be added to more than one board, getting notifications from both and not being able to determine which call came from what board, and thus disrupting its normal functionality. Additionally, pieces are notified whenever they are added or removed, for situations where in-game addition of pieces is allowed.

Boards make use of the exceptions provided in `engine.exception` to indicate illegal calls or requests to the board. Since these errors could occur at several different points, and because of the need to distinguish them from other types of runtime exceptions, `InitialPositionException`, `InvalidBoardException`, and `UnusableCellException` provide more clarity and specificity when each error is thrown.

Piece

Description A `Piece` represents an object that is placeable in a cell within a specific `Board`. `Pieces` can be asked to return a list of the valid `Ply` that they can perform. It also contains information about its color, along with its initial position in a specific type of board according to the local rules of the game.

Class type choice Piece is an abstract class. This is because, depending on the `RuleSet`, each piece needs to be aware of what moves are valid given their position. However, some methods are common to all possible pieces, such as `boolean isWhite()`, `void setUp()`, `void associateTo(Board board)`, which specify common mechanisms for determining a piece's color, setting it up given a concrete subclass-provided initial position, and an board association procedure.

Important features and design choices Once a piece is set up in a board, it automatically subscribes to the board's messaging service, engaging in a doubly coupled observer-broadcaster relation. This mechanism allows the pieces to receive information about the game, for the reasons specified in Section 2.2.1.

An initial consideration regarding pieces is the constant assumption that only two types of piece exist, either black-colored pieces or white-colored pieces. By convention, "white" is the name of the principal player in a two-player sequential game, while "black" is the name of the secondary player. For example, in chess, the white player can be considered the principal player since it moves first. A game such as a four-player chess would violate the assumption of dichromatic pieces. In order to solve this situation, a player number (as opposed to `boolean isWhite`) would be required on each of the player number descriptions. However, this decision would hurt readability of the code. In the case that such an extension were required, a simple code refactoring would allow a transition from a 2 player sequential game to an n player sequential game.

Since a piece has to be associated to a particular instance of a board, it is unfeasible to clone a piece, since it would be impossible to set the cloned piece to be associated with a different board, so pieces would not be clonable and placeable on different boards. For that reason, `Piece reproduce(Board board)` acts as a cloning mechanism, except that it provides the functionality of changing the associated board to the specified board, thus making piece "cloning" feasible.

Since several pieces in a board could be identical (e.g. two black pawns in a chess board), there is no way to distinguish them by just observing their characteristics and specfield representations. Therefore, the concept of referenciaql equality for comparison is required in order to ensure correct behavior of the board. However, a mechanism to compare whether two pieces are of the same type and the same color is provided under the name of `boolean similar(Piece)`. This method allows the comparison of two pieces, even when they are associated to distinct boards, and the comparison of plies that represent similar end results, but contain a referencially unequal piece in them. See Section 2.2.1 for further discussion on ply-action-board interactions.

In order for the piece to contain all of the information related to itself, a piece holds information on the location it should set itself up when a game is started. Although the global rule set could contain this information, this centralized storage of information would bring difficulties in maintaining code, for different parts of the piece behavior would be maintained on different places. The piece therefore has information about the set of possible cells that it could be placed on. Since these cells can be more than one, this information cannot only be stored in a field. The piece provides a `void setUp()` method on which the piece automatically detects where it can be placed, or throws an exception when it is incapable of doing so. With this information, all of the pieces of a game can be setup by an agent that has no knowledge about the initial positions of the pieces.

A mechanism is needed to communicate the piece type to the game (for saving and piece generating purposes, for example). For this reason, each piece provides a unique identifier of itself via its type. Since it might still be possible that more information wants to be conveyed about the pieces in a string representation (e.g. "a black pawn located at e8", the `String toString()` method is not necessarily equal to a `String getType()` call, which can be useful for development or message display purposes.

Finally, `List<Ply> getPlies()` is one of the central functions of the piece.

Possessing all the necessary information from the board and its movements, via the inform mechanisms discussed above, a piece object can make a decision regarding which plies could be executed on it. These returned plies are filtered only regarding the local rules for the piece, and not with the general state of the game. For example, the restriction on *6.170 Antichess* that no piece can move when a player is in chess unless such a move releases the king from check is not a local rule, so it is not enforced by the local `List<Ply> getPlies()` call.

Ply and Action

Description An `Action` represents an atomic change in a board. There are only two types of actions: add, and remove. An add action adds a piece to a cell in a board. A remove action removes the piece from a cell. An action therefore includes information about its type, the cell, and the piece (if any, it refers to)

Any possible ply in a sequential game can be expressed as a sequence of actions. For that reason, `Ply` is simply a sequence of actions that represent a player's turn in a sequential game.

Class type choice Action is a concrete class, since *Pawned* assumes that the atomic game movement units are add and remove from the board. Although there are some other possibilities for atomic actions, such as "destroy cell", which renders the cell in question unusable, no direct support is provided for this type of action. However, its implementation would only require a modification of the specification and implementation of `Action` and `Board`.

Important features and design choices As discussed previously, *Pawned* uses plies to express any type of board manipulation necessary, thereby ensuring extensibility.

One of the issues that arise while implementing this data type is the concept of ply equality. A natural implementation of ply equality would be that a ply is equal to another whenever, if both plies were applied to a board with the same configuration, it would yield the same final configuration. However, some complicated issues arise from this definition. Since the ply is applicable for any given board, the most naive implementation of the algorithm would require executing both plies to be compared on every possible board, applying both plies to board clones and then determining whether the new boards are equal.

For example, consider the following two plies (assuming a 2D board for clarity in this document)

Ply 1
1. Remove piece from $b5$.
2. Remove piece from $a1$.
3. Add last piece removed.

Ply 2
1. Remove piece from $b5$.
2. Remove piece from $a1$.
3. Add last piece removed to $b5$.

4. Remove piece from $a1$.
5. Add piece x to $a1$.
6. Remove piece from $a1$.

In this case, these plies are not equal, since the final boards will be equal only if $a1$ is initially empty. Therefore, an algorithm would require to verify each action where the board at the relevant cell is both empty and full, thus resulting in an exponetial explosion. Additionalyl, the pieces making use of a particular ply would be required by the specification not to assume a particular way to represent an equal ply, thus forcing them to provide code generic enough that could work for any possible expression of a given ply. In order to control this problem, the pieces (as members of a rule set) can make assumptions about the predefined structure of the plies within that rule set. Then, by assuming the structure of the plies in the game, the code in the piece is greatly simplified.

In the same way as the pieces, a test for similarity (and not just strict equality) is provided.

Finally, the ply must provide a string representation of itself, in order to be reproducible by the factories in the rule set.

Parser

Description and Type A parser is convertion from a string representation of a coordinate to to and from the coordinate it represents. This mechanism is useful for saving and loading purposes, along with the `String getType()` and `String toString()` methods in `Piece` and `Ply`, respectively.

The parser is an interface, since the conversions are fully game-dependent.

RuleSet

Description A `RuleSet` represents a set of local and global rules upon which a game can be fully operated. That is, a `RuleSet` is a mechanism for determining and creating all of possible plies, pieces and boards supported by a particular game, as well as to indicate the set of possible plies and `GameTermination` that can happen given a board configuration. Since this method is fully dependent on each game, `RuleSet` is an interface. The decoupling between the rules and the game execution comes mainly from the treatment of any given ruleset through this interface. Since a `RuleSet` is defined almost exclusively in terms of its relationship with game, it is fully discussed in Section 2.2.2.

Important features and design choices The `RuleSet` can be thought of as the pluggable brain for the *Game Execution Environment*. For that reason, it must provide all the information the platform requires in order to correctly carry out a game.

A `RuleSet` provides a set of factories, namely a `PieceFactory`, a `BoardFactory`, and a `PlyFactory`. This factories are used by the

For a discussion of the interactions between the game layer and this interface, please refer to Section 2.2.2.

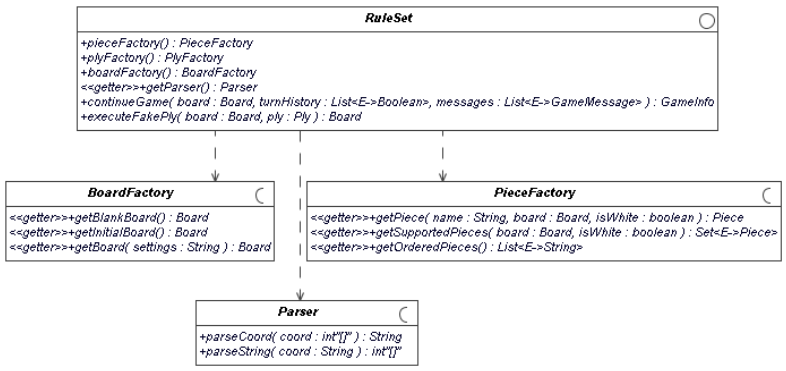

Figure 2.3: RuleSet and its factories

2.2.2 Game Layer

Contents The game layer is found in the `engine.game` package. It is the mechanism under which a board can be manipulated according to the rules specified by a ruleset. Further, it also allows the storage of the game state. Its components classes are `Game`, `GameInfo`, `GameMessage`, and `GameTermination`.

Game-RuleSet interaction for sequential games

The `Game` is in charge to keep the state in the game. However, the ruleset can be called from a static context. Although `RuleSet` itself is an interface, so it therefore must provide non-static methods, its behavior is not expected to change depending on its state. In general, a single ruleset can service several games of the same type happening at the same time. For example, when an Artificial Intelligence player is pondering moves, he does not need to create a ruleset for each of the possibilities explored, but rather call the same ruleset that the game is associated with.

By keeping a static communication with the `RuleSet`, resources can be optimized, by not requiring several instantiated objects to be present in order to use the game. However, this design choice is not strictly enforced. A rule set can be instantiated and can keep a game dependent state, if it considers necessary. In this way, rule sets can be optimized in the case that they are very resource intensive. Regardless of this, *Pawned* provides solutions for some amount of information storage, even with static methods, via its `GameMessage` system.

The `RuleSet` and the `Game` synchronize information every time a move is executed. The `RuleSet` receives a call to its `continueGame(Board board, List<Boolean> turns, List<GameMessage> messages)` method. With this information, the `RuleSet` returns a `GameInfo` bean, which contains information about the current state of the game.

GameInfo is a bean that contains information about a `Game` at a particular state. This information includes the list of valid plies for the current board configuration, the player who should execute the next move, and a list of game messages that are associated with the new state. However, a `GameInfo` object cannot represent a terminated state. Therefore, `GameTermination` objects are the representatives of the state.

GameTermination This is a `Throwable` object used to indicate that a `Game` has terminated, due to a termination condition. It contains information about who won the <tt>Game</tt> (or if it was drawn), and why.

Whenever a game has finalized, a `GameTermination` is thrown instead of returning a `GameInfo` object. This structure allows game terminations to be easily detected and propagated automatically along several method invocations, if necessary; it also allows the separation of termination and active states, for a cleaner implementation.

GameMessage objects describe additional information about a current game state. This information can be used to either notify users of relevant states in which they are involved, or for use in the ruleset for optimization purposes. For example, in a *Connect 4* game, the ruleset might be interested in knowing the previous plies that were executable, in order to check those pieces for path formations, as opposed to the complete board. In the same way, a game of *Chess* implementing draw conditions could make use of this messaging service to persist some string representation of previous board states, in order to validate whether a given board configuration has been repeated or not.

Game

Description As specified previously, `Game` provides the central mechanism for game control. It contains information about the game rules using a rule set. Additionally, it maintains a board representing the current board configuration, and a set of collected pieces. A piece p is collected if, after a ply has been executed, whenever p was contained in the board before the ply was executed and p is not contained in the board when the ply was executed, p is a collected piece by the ply. The state for a game can be either active or terminated. Game is a concrete class.

Important features and design choices The `Game` can be thought of as a proxy between a synchronous ply executor and a board, ensuring that the board is manipulated according to the rules, and keeping the relevant information about the game history and status.

Since a `Game` possesses no prior knowledge of the game being played, it must depend entirely on the rule set for game specific decisions. All of the executed plies result in a call to the ruleset, from which the game updates its state. All of the methods provided by this class are obtained in this fashion. However, `void executePlies(List<Ply> plies)` builds a game without consulting with the

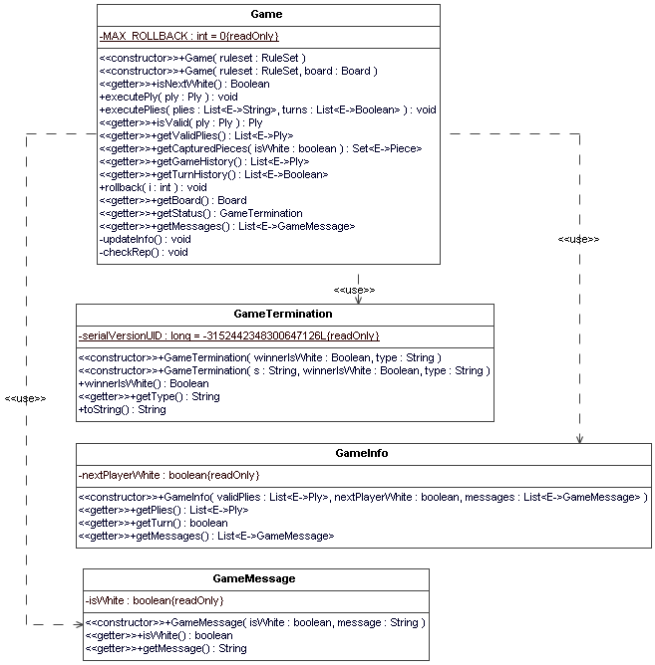

Figure 2.4: Game and its supporting classes

ruleset. This method can be used when a set of plies that are known to be valid need to be executed without the overhead produced by the constant calls to the ruleset. The pieces, due to the board's notification mechanism, would still be able to hold all the information they require for their proper functionality.

In order to guarantee that a game runs adequately, it would be necessary to query the ruleset after every move is made. This condition does not impose much additional overhead for a chess game of regular length. However, since the average length for all types of game is not known, and neither is the amount of processing required after each move for each particular ruleset, it is impossible to predict how much overhead would result in general. An extension on this program could provide a boolean as a parameter for this method, so that the caller can decide whether he desires to guarantee that all the plies are valid.

All the collections returned by this class are unmodifiable, in order to prevent modifications. See 5.3.4 for a discussion of the protection against undesired mutations in *Pawned*.

2.2.3 Controller Layer

Contents

The controller layer contains the classes `Controller`,`Stopwatch`, and `XmlFactory`.

Player and GameObserver - Controller Relation

Description The controller provides asynchronous communication between the players and the game. In other words, the `Controller` is in charge of providing a guarantee of synchronized calling to game, as well as handles the ply retrieval and information to and from the players. In order for the `Controller` to achieve this goal, two interfaces: `Player` and `GameObserver` are defined.

A `GameObserver` is an entity that can be subscribed to the listening message queues in a controller. An observer receives notification regarding the plies that are executed on a given gamem, along with information regarding the termination of the game.

A `Player` is a type of game observer that, apart from being informed about the game status, it gets queried for moves to be executed within the game.

Design Decisions The use of the observer pattern allows an easy solution to the problem of notifying observers of modifications in the shared game. There are various other alternatives that were considered before the constructions of these interfaces.

For example, the controller could work in a synchronized fashion, providing methods that allow another agent to determine when to make moves and when to obtain moves from computer players. In this case, an interface (such as the GUI) would be in charge of managing the workflow in the application. An immediate advantage to this approach is the reduced number of threads that are required. However, this approach must limit the application to a simple desktop human-computer

game. This is because the user interface would be empowered to input its own moves, and to ask the controller for the AI Player's moves, implicitly assuming that the only possible players are these.

Under *Pawned's* archiecture, a variety of player types and interfaces can be easily implemented. For example, any class implementing `GameObserver` can participate as a game listener. A large amount of services can then be added to games in this fashion, such as logging services, game analyzers, artificial learners, etc.

Similarly, different types of players can easily be added to the architecture without any modification. For example, the notion of an Internet Player, a player that connects to the controller via Internet in order to play a game, or a plugin for connection with existing chess servers are all easy to implement under the structure provided by *Pawned*.

Controller

Description A controller is a the communication link between the players and a game. It is also responsible for saving the game using the information contained in the Game, as well as managing the time in the game.

Threading issues A controller consists of at most four distinct threads. Two of these threads are devoted to keeping the time remaining for each of the players, and ending the game as soon as the time expires. A third thread constantly executes the `TurnCycle`, the ply dispatcher in charge of retrieving and executing plies for the players, as well as informing each of the observers of an executed move.

There are, once more, several implementation possibilities for the controller regarding its interaction with the players, assuming an asynchronous communication. For example, each call to an `inform(String ply)` could come from a pool of threads. This mechanism would be efficient, especially in the case where several observers are expected. Under these conditions, a loop that waits for each of the observer's `inform(String ply)` method to terminate could lead to very long wait times, or even to an unrecoverable state. However, there is a tradeoff with using this approach. By separating the calls to `inform(String ply)` from the `submitPly()` dispatcher – that is, from the mechanism in charge for retrieving the moves from the players –, then it is no longer possible to guarantee that a player whose turn it is to play will get notified of all the plies that were previously executed before it is his turn to move.

Since the lack of this guarantee would require more complicated information handling, the inform mechanism is carried out in a single thread. However, a change to another type of implementation would be very easy to obtain.

XML Saving and Loading In order to save an XML file, the controller requires to provide information about the piece configuration, the move history, the turn history, and the time at which each ply was executed. This information is all contained in the game. Since each of the plies and the pieces in the board have a particular string representation, this is used to persist them into the XML file. Further, the

ruleset provided parser is used to parse the cell coordinates into the game-specific representation of the cells. Additionally, whenever a game is terminated, this information is also readily available as a `GameTermination`.Finally, the ruleset provides its own String representation, completing the requirements for the information to be saved in the XML file. In this way, the XML file can be generated with no ambiguity.

In order to load a game from XML file, it is necessary to create a new controller using a constructor that takes an XML file as a parameter. Once the controller has knowledge about the ruleset to be used, each of the pieces can be generated from the ruleset's piece factory. Similarly, each of the ply representations can be translated into actual plies using the ruleset-provided ply factory.

StopWatch

Description A `StopWatch` represents a scheduling device that hsa the ability of being paused and resumed at any point during its execution. After its time expires, a `StopWatch` executes a `Runnable` task.

Important features and design choices The `StopWatch` class was designed in order to provide the asynchronous notification to the players when the game has terminated due to time depletion.

An alternative to this implementation would be a non-threaded watch which verifies the remaining times only when moves have been executed. However, such a system would be confusing and would not allow for an immediate notification of loss by time depletion.

Further, the timer could be implemented at the User Interface level, only notifying the controller about the time spent on each move. However, since all the players would require to implement their own timing mechanism, this decision would lead to code repetition. Additionally, the responsibility delegated to the players to provide the time consumption would require the strong assumption that all of the players that could possibly be implemented will implement this functionality correctly. These complications are removed when the controller itself handles the time. There are, however, some considerations to be made when the communication time between the controller and the player is expected to be long (e.g. using an Internet connection). These conditions could lead to a biased disadvantage against a player with a slower connection.

It should be noted that the the `StopWatch` contains an inner `CheckRep` class. This is not only a static method as the regular checkReps, but rather a representation invariant checker that keeps a state in order to determine whether the timewatch is preserving its state.

Since a `StopWatch` implementation relies on a constant update of the amount of time remaining, we assert that, if T_0 is the initial time at which the clock was started and T_f is the actual time, then let $\{r\}_n$ be the sequence of values set for remaining. Then $S = \sum_{k=2}^{n}(r_k - r_{k-1})$ is the total time that has been consumed from the stopwatch.

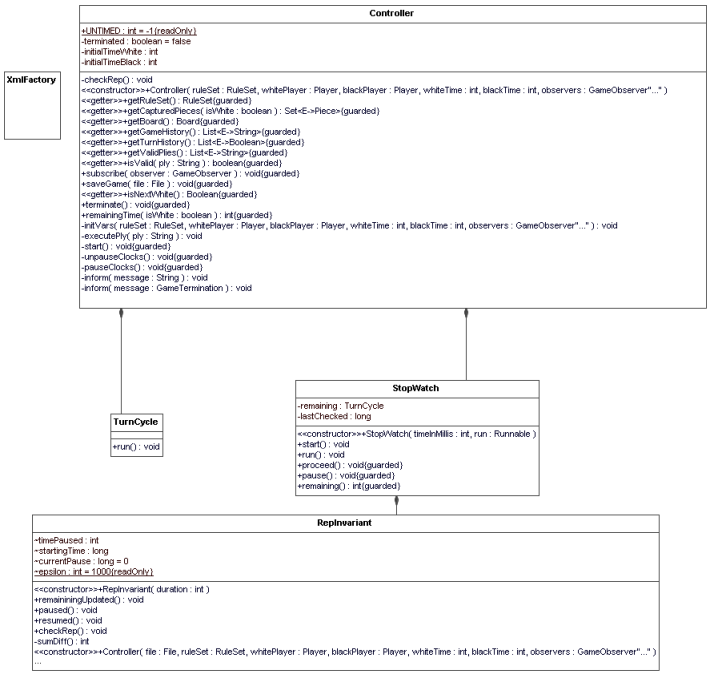

Figure 2.5: Controller and its supporting classes

$$T_f = T_0 + S + P$$

P represents the total sum of the time when the stopwatch was paused.

2.3 Standard Antichess

Along with the *Game Execution Environment, Pawned* provides the implementations of a *Standard 6.170 Antichess* and *6.170 Antichess with EnCastle*. The pieces do not generally require the listener mechanisms inside of board. This ruleset is provided under the `ruleset.antichess` packages. The set of pieces are contained in `ruleset.piece`, whereas the plies are contained in `ruleset.plies`. All of the previous are simple implementations of the required specifications detailed above.

A generic antichess ruleset is extended by the encastling and the standard versions, each implementing *Standard 6.170 Antichess* and 6.170 Antichess with Encastling, respectively. This is because the only change resides in the `PieceFactory`, which includes a different type of pawn and king, which have the faculty to castle and eat en passant.

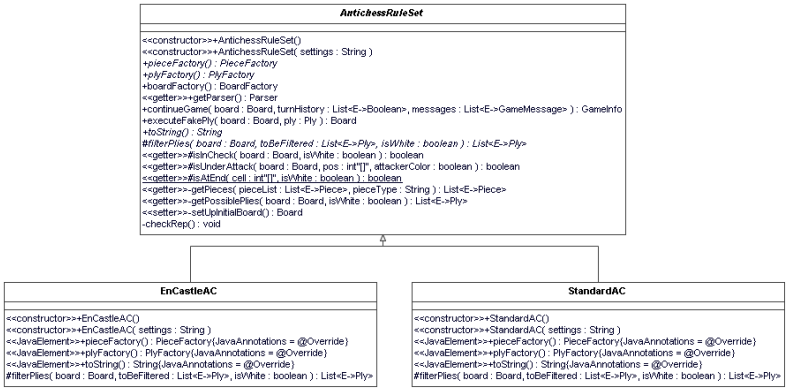

Figure 2.6: Antichess Ruleset

Chapter 3

Outstanding Assignment Observations

3.1 AiPlayer

The specification for AiPlayer dictates that the AiPlayer should submit an empty string whenever it is not allowed to make a move. From the specification for **AiPlayer**:

> Returns: the move to make. Should return an empty string if this player cannot make a move (because the opponent's move ends the game, or for some other reason).

The fulfillment of this requirement brings several issues into consideration. The choice of overall architecture, in which players are exclusively asked to make a move when they are required to do so, is incompatible with the nature of this specification. An implementation that could trivially support it would require direct involvement in the turn management and knowledge of the specific game rules regarding termination and conditions such as *stalemate*. Such a direct involvement with the game requires a significant amount of coupling that can hinder the possibility to implement further players or extensions to the game. Therefore, *Pawned* uses a notification system in which players are asked to submit a move only when it is necessary for them to do so. In order to fulfill the specification, *Pawned*'s implementation of **AiPlayer** acts as a representative for the other player in the tournament, playing in the **Controller** against the actual **AIPlayer**. Since **AiPlayer** is not asked to submit empty strings by *Pawned*, **AiPlayer** uses a queuing system for the actual **AIPlayer**'s moves as they are notified from the system, in order to detect and send these through it's `makeMove(move, time)` method.

For example, in a game such as *N-move* chess, where each player has to play N moves during his turns, the turn delegation becomes problematic. In this case, there are two options either allow the user to input all of his moves in a single string after he receives a `makeMove(move, time)` call, or query him N times for a move. In the first case, the other player will be forced to submit twenty empty strings to

the referee, thereby forcing the player to constantly report to the referee when it has no necessity to do so. Not only does this result in a significant design flaw, but could also have performance repercussions in cases such as calls to the players via the Internet, such as *RMI* invocations. On the other hand, if all the moves are input at once, then the sequential nature of the `makeMove(move,time)` calls is broken, making the return of an empty string seem arbitrary.

In addition, the choice to request an empty string for invalid moves makes the application more prone to failure and makes it harder to recover from erroneous input. Since a player can return an invalid move when it is not his turn, *makeMove(move,time)*, there is a chance that the player will submit some type of ply. Further, there is no way to determine the time remaining for the other player, so the player will be unable to submit an empty string when the game has finished as a result of time depletion, thus making the player unable to completely determine whether the game is still being played or not.

it is apparent that due to this design choice, several computations are bound to be repeated. Since the referee has to verify whether a move is valid or not, then he needs to determine whose turn it is, in order to predict whether a given player should have returned an empty string. On the other side, each player is required to carry out this computation, since it has to determine whether it has to return the empty string or not.

3.2 TextUI

TextUI does not make use of the asynchronous information messaging system provided by *Pawned*. In fact, to an extent, the requirements for TextUI and the expected functionality of `Controller` are incompatible. For example, the TextUI is expected to be able to control the artificial intelligence player's movements, whereas the `Controller` assumes the role of ply retriever in the execution environment. For this reason, TextUI was implemented at the `Game` level, and not the `Controller` level.

The TextUI was extensively used as a testing mechanism for `Game`, since it is a wrapper for it, without the additional functionality provided by the `Controller`. However, the TextUI lacks the extensibility supported by the rest of the execution environment, since it is tightly coupled with assumptions regarding its role in the game process. For example, consider the problem of allowing players compete against each other over Internet. Since the command line interface heavily relies on synchronous communication and

Further, the requirement that the players themselves submit the time they took to submit the move makes it hard to possess a reliable time measurement, since an accurate measuring cannot be achieved. On the other hand, `Controller` could easily allow an implementation of such an addition, by implementing a particular version of `Player` which is capable of communicating across the Internet.

Chapter 4

Testing

4.1 Strategy

4.1.1 Testing of `engine.adt`

Testing `engine.adt` posed the problem that the majority of its classes are not fully implemented (some methods are abstract). In order to solve this problem, basic types of each abstract class were created and those classes were unit tested.

`CountingPiece` and `SimplePiece` are two pieces that were developed to test board operations and functionality of the board's messaging systems. Special attention was paid to ensuring that the order of the messages behaved in a manner consistent with the specification, and also that the pieces were successfully added, removed, and cloned within the board, ensuring that the specifications are met on all the different possible states of a board position (unusable, empty, and occupied).

In order to provide a board concrete class, `UselessBoard` and `LineBoard`, 0 and 1 dimensional boards, respectively, were created in order to place the implemented pieces for testing.

4.1.2 Testing of `RectangularBoard`

In order to test the rectangular board, a test driver was developed. This test driver is capable of creating boards and adding pieces to it. The addition and removal of pieces was carried out under different circumpstances (on unusable, empty, and unoccupied cells) and the behavior was tested for compliance with the specification.

4.1.3 Testing of `StdACRuleSet`

Several unit tests were implemented for each piece in *Standard 6.170 Antichess* and *6.170 Antichess with Castling*. A rectangular board was set up with different configurations of pieces, and each piece was queried for its valid plies, comparing them against their expected results.

Additionally, the ruleset's global filtering was tested using a similar mechanism.

4.1.4 Testing of `controller.*`

The components of the controller package (StopWatch and TurnCycle) were tested separately. Controller was tested along with the GUI, both in regular and in stress testing. The XML Factories were tested using unit tests as well.

4.1.5 Testing of `Game` and `TextUI`

`TextUI` and `Game` were tested together. Since `TextUI` is a wrapper directly built on top of `Game`.

4.1.6 Testing of the GUI

Since the development of the Graphical User Interface could not wait until the remainder of the game was finished, a temporary controller lookalike: `Kontroller`, was developed to simulate calls by controller to the GUI. This top-down approach to the GUI development allowed for an early set of initial tests on the user interface.

The GraphicUI tests consist of hand made tests carefully chosen and recorded to prove the functionality of *Pawned*. An scenario will be described in detail, and then we will check that the GraphicUI behaves as expected.

4.1.7 Opening *Pawned*

Three menus are enabled, with the following submenus:

- File: *New Game*, *Save Game*, *Load Game*, and *Quit*
- Game: *End Game* and *Display Options*
- Help: *User Manual* and *About*

The Save Game and End Game options in the menus are disabled. Each submenu except *Display Options* will open a new window. *Display Options* contains a submenu with three options that can be checked individualy (Highlight Movable Pieces, Hightlight Possible Moves, Highlight Last Move). These options are disable from user selection before starting a game.

There is a ToolBar with one button for each of the following options: New Game, Load Game, Save Game, and End Game. The Save Game and End Game buttons are disabled.

The window can be minimized and closed using the small buttons in the upper corner of the window. Maximize is disabled. The main panel should be empty.

The following hot keys are enabled:

- New Game - Ctrl + N
- Load Game - Ctrl + O
- User Manual - Ctrl + U
- Quit - Ctrl + Q

New Game

The New Game window can be accessed by either the New Game option in the File menu of the button in the toolbar. For each of the players the New Game window has:

- Text field to input the player's name (max 8 char)
- Radio buttons to select between Human or Computer
- Combo box to select the intelligence level of the Computer player (Baby, Kid, Adolescence)
- Checkbox to select Timed game
- Spinners to input the time (one spinner for minutes and another for seconds)

If the Computer radio button is selected, the computer level combo box is enabled. Otherwise it is disabled. If the Timed checkbox is selected, then the Time spinners are enabled and can receive an input. Otherwise they are disabled. The spinners are set to receive only a number from 0 to 60 for minutes and from 0 to 59 for seconds.

The New Game window also contains a combo box where the user can select the type of game desired (Standard Antichess, EnCastle Antichess, and Connect N). When this window is opened, the default game selected is Standard Antichess and two pawn images are displayed, one pawn of each color, to identify the players. If EnCastle Antichess is selected everything stays the same. If Connect N is selected, the pawns change to a red chip and a black chip. A spinner appears to the right of the game type combo where the user can select the value *n* for the Connect N game. The spinner will let you to enter values from 2 to 7.

4.1.8 Load Game

This window is very similar to the New Game window. The differences are:

- There is a read-only text field and a Browse button. The Browse button opens a file dialog window that is set to filter XML files. The text field is updated after the file is selected from the file dialog.
- The Timed checkbox is disable, along with the Time spinner fields.
- The game type combo box is disabled.

After selecting a file in the File dialog, the Load Window will refresh and show some information about the file selected (assuming the file is valid). If the saved game was untimed, then the Timed checkboxs will be unchecked and the Time fields disabled. If one of the players was set up with limited time, then its Timed checkbox will be checked and the Time fields will display the remaining time it has for playing. If both players were timed, then this information will be updated for both. At this point *Pawned* will allow the user to change these options.

The game type combo box will also be updated with the type of game of the saved file. However, this field will always remain disabled. The spinner for selecting the *n* value for Connect N also remains diabled, but it refreshes an shows the n value of a saved Connect N game.

Cases to test for each type of game:

- Untimed game
- Only one player untimed.
- Two timed players.

Other things to test:

- If a non existing file is chosen in the File dialog, display an error message box.
- If 'OK' is clicked without selecting a file display an error message box.
- If a corrupted file is tried to be loaded, display an error message box.

4.1.9 Starting new games and changing from game type to game type

Pawned allows you to change from two game environments that are further apart from each other than two variations of Chess. The GraphicUI was tested to assure the ease of this change.

4.1.10 Controller-GUI stress testing

In order to test a correct threading and concurrency implementation, several matches between extremely fast random players were run, which return their moves immediately after they are queried. At the controller level, it was tested that

- A controller is successfully initialized, both starting a new game or loading an XML game, even when the players will immediately respond to a request.
- A game can be saved when players are engaged in a very fast gaming activity.
- A controller successfully terminates a game when players are engaged in very fast playing.
- The sets of captured pieces and plies do not become inconsistent when the players are exercising moves very fast.

The previous items were run with delays on different parts of the code (e.g. the `RunCycle` in charge of turn management, the `inform(Ply ply)` method in `Controller`, in charge of notifying the players of the plies as they are executed by the players.

The `StopWatch` objects were tested for accuracy to within 5 ms under regular and heavy CPU load. Additionally, tests were run to ensure that the pauses and starts

are responsive to within 1 millisecond in a *Centrino Duo, 1.66 GHz*, as well as a *Centrino, 1.5 GHz* machine. Finally, a stopwatch test on an interval of `Integer.MAXVALUE` was carried out to test border cases for the stop watch.

All these tests were successful.

In the same way, the GUI was tested under similar conditions, in order to ensure appropriate behavior, even under extreme conditions. Games were saved and loaded one after another, and while games with the fast random player were being carried out. Although several tests were successful, some race conditions were detected, as well as some deadlocks under certain stringent conditions. For example, when a game is being played by extremely fast random players, and the termination condition is launched exactly at the time in which a person is choosing a file in the second dialog of the GUI, the main window comes to a halt. This situation is solved whenever the players take longer than 10 ms to make a move.

Chapter 5

Reflection

5.1 Evaluation

5.2 Lessons

5.3 Known bugs and limitations

5.3.1 Captured pieces

The definition of a captured piece in a board is the following: When a ply is executed on a board, if a piece that was inside a board before the ply was executed is no longer in the board when the ply finished, then the ply is considered 'captured'. However, this interpretation cannot fully incorporate particular requirements for the games.

For example, in chess, it is not necessarily true that a pawn that has been coronated is 'captured'. However, according to the definition, this piece would actually be captured. Further, some games do not consider the notion of captured pieces (e.g. connect four), so such functionality is not particularly useful.

A possible solution for this problem could be to include this information in the set of objects of type `GameMessage`. However, further analysis of this problem is required to provide a full solution.

5.3.2 GUI Threading

As mentioned before, the GUI behaves poorly under stress testing. Although the situation in which players move almost instantaneously is very unlikely, this situation could bring undefined results in a slow computer under heavy load. However, the GUI must be inspected to ensure proper thread management.

5.3.3 AI Player

Currently, the AI Player uses a minimax algorithm in order to fulfill the parallelizability requirement. Because this algorithm is very easily transofmred into a concurrent algorithm, performance increases almost by about as much as the increase in the number of processors.

However, in doing so, the AI Player fails to use any optimization for the minimax algorithm, such as $\alpha\beta$ pruning. Since $\alpha\beta$ pruning requires a sequential depth-first search on the game tree, it is hard to obtain satisfactory results while parallelizing, since a naive parallelization would require computation of more nodes than what would be required in a sequential algorithm.

In order to produce a satisfactory AI Player, the AI Player should build a transposition table that is concurently available and modifiable by all the concurrent threads. However, time constraints did not allow the implementation of such a system. Although the minimax algorithm does fulfill the requirement of using the power of parallel processors, its expected performance is poor.

5.3.4 A final note on extensibility

Pawned constanatly attempts to provide an extensible environment for game execution. The controller will always ensure that the game is played following the rules from the rule set. However, to do so, the objects that hold a reference to the instantiated controller keep an implicit promise not to modify the board in play. However, for a publishable API, this behavior might be very dangerous. For example, consider an Internet enabled version of the controller, such that external developers can create `GameObservers` and `Players`. Malicious attacks would be easily carried out by modifying the board in play with arbitrary plies, thus rendering the game unplayable.

However, this situation is easily solvable by cloning the board whenever it is requested, instead of providing the player with the actual board. This feature was not included on this version of *Pawned* since it can be guaranteed that none of the UIs will ever modify the board directly.

Appendix A

Antichess Rules

A.1 Standard Antichess

Antichess is a variant of chess in which the goal is to either lose all of your pieces (except your king) or checkmate your opponent.

A.1.1 The Chessboard

Antichess is played between two opponents by moving pieces on a square board. The board is composed of 64 equal squares. The eight vertical lines of squares are called columns. The eight horizontal lines of squares are called rows. The squares are colored black and white alternately. The lines of squares of the same color, touching corner to corner, are called diagonals. The chessboard is placed between the players in such a way that the near corner to the right of each player is white. The columns are labeled a to h from left to right. The rows are numbered 1 to 8 from bottom to top.

A.1.2 The Pieces

At the beginning of the game, one player ("white") has 16 white pieces, and the other ("black") has 16 black pieces. The white player pieces are: one King (e1), one Queen (d1), two bishops (c1 and f1), two knights (b1 and g1), two rooks (a1 and h1), and eight pawns (row 2). The black player pieces are: one King (e8), one Queen (d8), two bishops (c8 and f8), two knights (b8 and g8), two rooks (a8 and h8), and eight pawns (row 7). The initial position of the pieces on the chessboard is shown in Figure ().

A.1.3 The moves

A move is defined by the following rules:

1. White moves first. The players alternate in making one move at a time until the game is completed.

Figure A.1: Initial position of the pieces

2. A move is the transfer by a player of one of his pieces from one square to another square, which is either vacant or occupied by an opponent's piece.

3. No piece except the knight may cross a square occupied by another piece. That is, only the knight may jump over other pieces.

4. A piece played to a square occupied by an opponent's piece captures that piece as part of the same move. The captured piece is immediately removed from the board.

A player's moves are limited by the following fact: A player is forced to capture an opponent's piece whenever possible. If a player can take several of the opponent's pieces, he/she is free to choose which piece to take. This limitation does not exists in regular chess. The only exception to this rule is being in check.

All the pieces move exactly as they do in standard chess. An excellent description of how each piece moves and captures is at

`http://www.princeton.edu/~jedwards/cif/chess.html`

In the Standard Antichess rules:

- Castling and en passant are **not** allowed.
- When a pawn moves to the last row on the opposing side, it turns into a queen. (In standard chess, such a pawn can be turned into any piece of the player's choice.)

A.1.4 Handling Check Situations

In short, each move of player A must observe the following:

1. If A's king is under check, A must move the king out of check.
2. A cannot move in a way that causes the king to come into check.
3. If A can take one of B's pieces, then it must (unless disallowed by the previous rule).
4. If A's king is under check and A can move in such a way that the king is out of check by either taking B's piece or in some other manner, A must take B's piece.

The king is in check when the square it occupies is attackable by one or more of the opponent's pieces; in this case, the latter is/are said to be checking the king. A player may not make a move which leaves his king on a square attackable by any of his opponent's pieces; e.g., the player cannot move the king into check. Check must be resolved by the move immediately following. If any check cannot be parried, the king is said to be checkmated or mated.

It is the foremost obligation of each player to move the king out of a check. This overrides the rule that you must take an opponent's piece. For example, in the figure below to the left, it is black's turn and black must move its king out of check even though it can take white's bishop on c6 with its rook.

If it is possible for a player to remove the king from check as well take a piece of the opponent, then the player must do so. For example, suppose the black player's king is under check from white's rook. Further, suppose black has two choices to move away from check Ů remove the check with or without taking a white piece. In that case, black must take white's piece and remove the check. In the figure below to the right, black's king must take the white bishop with the king in the next move (it cannot simply move the king away from check without taking the bishop).

A.1.5 End of Game

Player A wins the game against player B if:

1. all pieces of A except for the king are taken, or
2. player A checkmates player B, or
3. player B's timer runs to 0.

If player A checkmates player B and on the same turn takes the last of player B's non-king pieces, player A wins (ie, the checkmate prevails).

The game is stalemated if the king of the player who has the move is not in check, and this player cannot make any legal move. In the example on the right, black is stalemated on their turn, since neither their pawns nor king can move. In antichess, the stalemated player loses their turn, and the opposing player may continue to take turns until the stalemate is broken or the game is won.

If the two players are stalemated, then the game ends in a draw (double-stalemate)

A.2 EnCastle Antichess

Castling is a special move that allows a player to move both their king and rook in one turn under certain conditions. It is described towards the end of

`http://www.princeton.edu/~jedwards/cif/basics2.html`

En passant is a special move that pawns can make. It is described towards the end of

`http://www.princeton.edu/~jedwards/cif/basics7.html`

All other rules of standard antichess apply.

A.3 Connect N

Connect N is a two players game which takes place on a rectangular board placed vertically between them. Chips of two different colors are used, red for the first player and black for the second player. During a turn, a player drops a chip at the top of the board in one of the columns; the chip falls down and fills the lower unoccupied square. A player cannot drop a chip in a column that is already full.

A value n is chosen at the beggining of the game (n has to be greater than 1 and less than the largest of the side dimensions of the board). The red player makes the first move. The object of the game is to connect n chips vertically, horizzontally or diagonally. If the board is filled and no one has alligned n chips then the game is drawn.[1]

[1] Rules for the popular *Connect Four* can be found in `http://www.ce.unipr.it/~gbe/cn4rules.html`

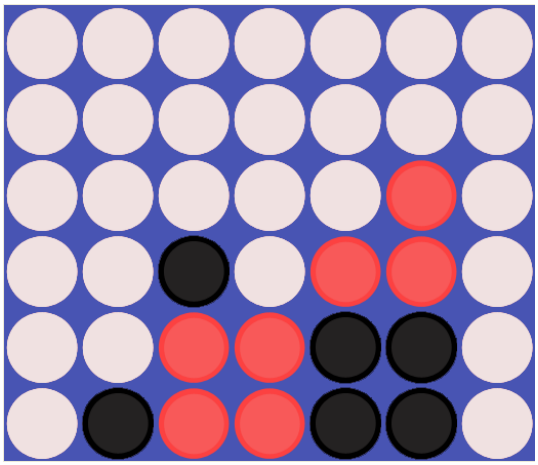

Figure A.2: Winning position for red player for n=4

Appendix B

TextUI Specification

Main is an interactive loop between a human and a computer. The `TextUI` should wait for user input. Valid user input is in the form of the commands shown in the requires clause. Parameters to commands are enclosed in square brackets. Outputs as a result of a command should be terminated with a newline. If multiple lines are output, the last line should also be terminated with a new line.

When main is executed with no command-line arguments, supported commands are:

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. For example, StartNewGame computer 60000 human 180000 should start 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 should output New game started on its own line.

SaveGame `[filename]`

The system should save the game to the given filename and report Game saved on its own line.

LoadGame `[filename]`

The system should load the game from the given filename. Once the files is loaded, print Game loaded on its own line. You should report Corrupt file if the file does not have a correct format. We will not require you to determine if the board is legal. If no game is currently in progress from a previously executed StartNewGame or LoadGame command, then assume a human-human game.

GetNextMove

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. The printed move should be in the 'standard string format' described in the assignment. The time aken to compute the move should be subtracted from the computer player's game clock.

If called repeatedly, this should return the same move over and over without further decrementing the computer's time remaining.

MakeNextMove

If it is a human player's turn, the system should print Please specify human move on its own line. If it is the computer player's turn, and GetNextMove has not yet been called on this turn, then the system should 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' described in the assignment. 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 should be printed. If the move is not legal, the system should print, on its own line, 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).

PrintBoard

System should print the current 'state' of the game to the screen using the same format as if it were being saved to a file. The output should end with (at least one) a newline.

IsLegalMove `[move]`

System should print, on its own line, either 'legal' or 'illegal' to specify if the move is a legal next move.

PrintAllMoves System should, in alphanumeric order, print all legal moves for the next player. Each move should each appear on its own line.

GetTime `[player]`

On its own line, the system should print the time remaining in milliseconds for the player specified. For example GetTime white, should print 3000 to indicate 3 seconds left for the white player. If the time for the player is unlimited, the system should print 'unlimited'.

QuitGame

Prints (on its own line) Exiting game and terminates the present game and application. QuitGame cannot be the first command.

For each command other than GetNextMove the specified behavior completes within 10 seconds. GetNextMove may take no more than ten seconds more than the player's time remaining to complete; if it exceeds the player's time remaining it must report a human victory in the format described below.

If the user input does not match one of these commands, output Input error alone on one line. Also, the first valid command entered must be either StartNewGame or LoadGame, or else Input error is printed.

When a player has won the game, output on its own line: [Player color] Player has won. For example, if the black player has won, output: Black Player has won. At this point, you can assume that the Antichess program has just been started and therefore, you only need to support the subset of commands.

The behavior of the TextUI is unspecified when main is run with one or more command-line arguments.