Preliminary Design

Julián Hernández
Rodrigo Ipince
José Muñiz

April 18, 2007

Contents

Chapter 1

Preliminary Design

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. From this point onward, the document uses the form *Pawned does* when referring to the form *Pawned should do*.

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 B.

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:

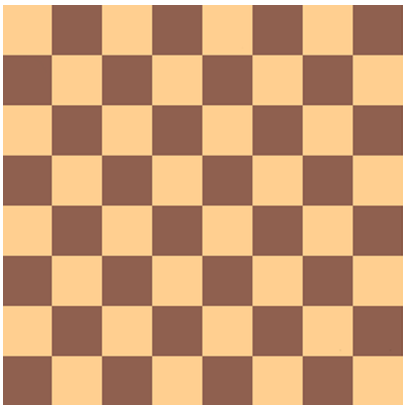

Figure 1.1: A standard board for playing Antichess

- Graphical input mechanism. That is, a way for the user to input their moves using a mouse pointer
- 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 C
- Termination of the current game

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

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 game: an activity between two players that can manipulate pieces in an n-dimensional board according to a fix 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 accessible to

the players. For example, in *6.170 Antichess*, 'check' or 'stalemate' are two possible messages.

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 the ruleset specified 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

Game Resign, Undo?, Hint?, Display Options?

Figure 1.2: *Pawned*'s Graphical User Interface

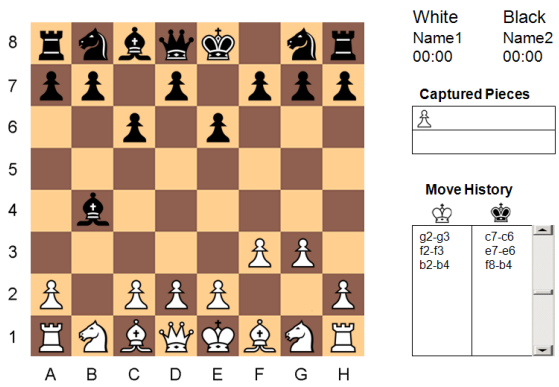

Figure 1.3: *Pawned*'s GUI Main Panel

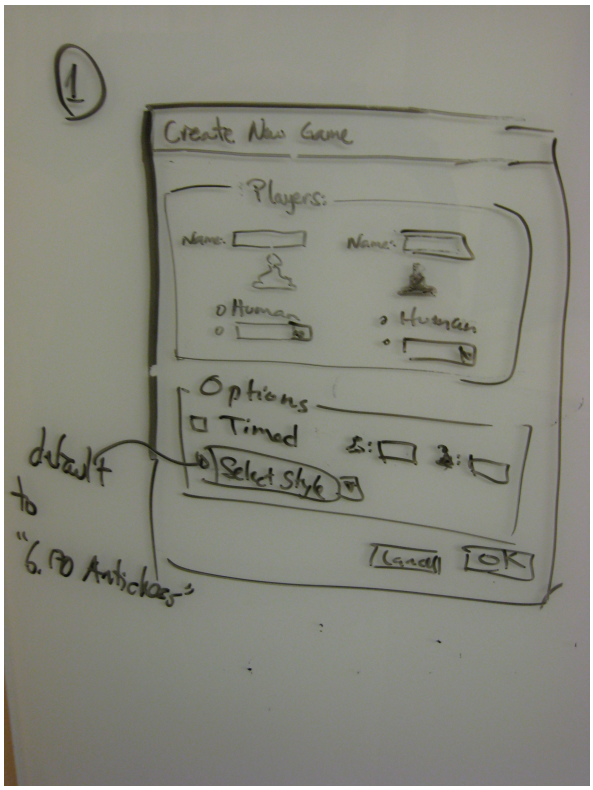

Figure 1.4: *Pawned*'s New Game Window

Help User's Manual, About...

Pawn (surprise features that will not be revealed yet)

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. *6.170 Antichess*, *AntiKingChess*, *GreekChess*, *SpaghettiChess*, *MonkeyChess*, 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 also opens a file chooser window to select 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 **Resign** option allows the user to end a game without exiting the application.

The **Hint** option gives the Human player a hint of what to play next.

The **Display Options** option lets the user select between different graphical options, such as different piece images, and different boards.

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

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

1.3 User Manual

```
throw new RuntimeException(``Not yet implemented!'');
```

1.4 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.

1.5 Problem Analysis

1.5.1 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 consistss 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.

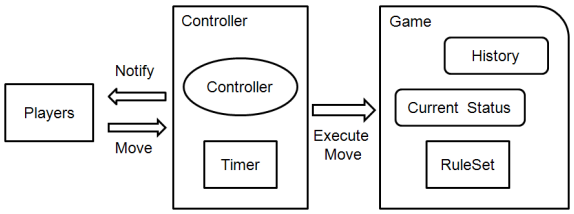

Figure 1.5: Component diagram

- 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 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.

See Figure 1.5 for a visual depiction of these components.

1.5.2 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 1.5.2.

Determining a finishing condition

A game finishes when a termination condition has been met or when one player has run out of time. In the case of *6.170 Antichess*, these conditions are checkmate, or win by depletion of pieces, 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 should notify 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.

1.5.3 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 be 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* uses a modified XML schema to save this information. All the information can then be reproduced from the saved game and the game can continue. However, *Pawned* is still able to load a game that was saved by a different application using the standard XML schema, in which case the information can be reproduced by simply executing all the Plies specified in its move history.

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 indeces. 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 *6.170 Antichess*.

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.

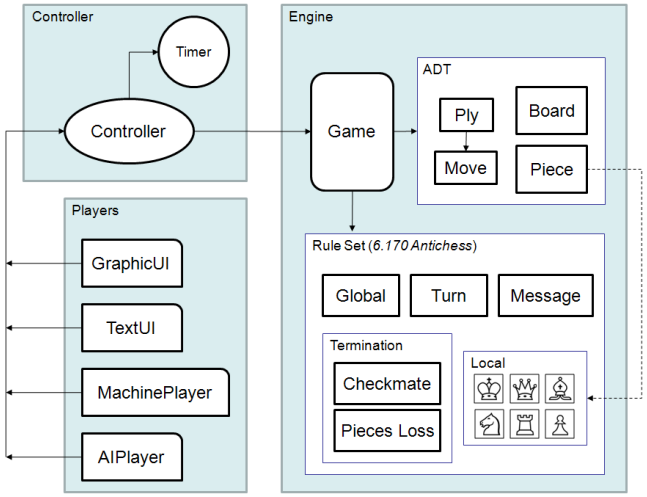

Figure 1.6: Conceptual diagram

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.
- An asynchronous messaging system for notifying the subscribed Players about their turn
- A system for adding and removing observers from 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 basicmost 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.

Chapter 2

Plan

2.1 Individual To-Do Tasks

- Appendices A-D
- Spellcheck documentation

2.2 Road map

Documentation, Implementation, and Testing:

Week 1 Finalize Design Board.java, Piece.java, RectangularBoard.java, StdACPiece.java, Move.java, Ply.java Timer.java implementation and unit testing.

Week 2 RuleSet.java, Game.java, Controller.java implementation and unit testing, TextUI.java, DumbAIPlayer.java, MachinePlayer.java, Integration testing.

Week 3 AIPlayer, GUI.

Appendix A

Antichess Rules

```
throw new RuntimeException(``Not yet implemented!'');
```

Appendix B

Machine Player Specification

```
throw new RuntimeException(``Not yet implemented!'');
```

Appendix C

XML Schema

```
throw new RuntimeException(‘‘Not yet implemented!’’);
```

Appendix D

TextUI Specification

```
throw new RuntimeException(``Not yet implemented!'');
```

Index