Project Planning – Pawn Pusher 9000

By Jui-Chi Liu and Ray Adrian

1. Project Breakdown

We are going to use the Model-View-Controller (MVC) pattern that is similar to the Flood It question we did for Assignment 4 to implement this project.

Model: The model will be divided into three classes, Game, Piece, and Player. The Piece class is an abstract class inherited by six child classes representing the six types of chess pieces (King, Queen, Bishop, Rook, Knight, and Pawn), which each has their own rules of movement. The Player class is also an abstract class that is inherited by two types of player, Computer and Human.

View: The View is consisted of two types of display, TextDisplay and GraphicDisplay.

Controller: The Controller is the class that receives user (or computer) inputs for moves and connects Model and View.

We will put Model as the top priority of our implementation to make sure that we have enough time to correctly implement and fix the game logics. However, since the Computer AI part is much more complicated than other parts. We will start other parts of the program while still implementing it. Ray will start on implementing the six chess classes and Jui-Chi will be implementing the two player classes. Ray will be responsible for implementing the movement rules for the six types of chess pieces that can be used for the course of the game. After the Player (excluding the Computer class) and Piece classes are done, the next step is to build the game board, which is the Game class that represents the two-dimensional board responsible for storing game status. The board will be built by Jui-Chi.

After the Model of this project is built, Adrian will work on Controller and TextDisplay, which includes the command interpreter and the output function. Finally, after we have a program that works (or at least partially works), we will finish the graphic display. After all parts of the program are implemented, we will start to test and debug the program.

2. Work Distribution

The planned work distribution and schedule will be as follow:

Part	Group Member	Completion Date
UML	Liu, Adrian	Nov 26
Plan of Attack	Liu, Adrian	Nov 26

Player Class-Computer (AI)	Liu	Dec 02
Player Class-Human	Liu	Nov 28
Game Class	Liu	Nov 29
Six Pieces classes	Adrian	Nov 29
Controller Class	Adrian	Nov 30
View Class-TextDisplay	Adrian	Dec 01
View Class-GraphicDisplay	Liu, Adrian	Dec 03
Makefile	Adrian	Nov 28
Testing and Debugging	Liu, Adrian	Dec 03

3. Project Specification Questions

Q1: Chess programs usually come with a book of standard opening move sequences. These are moves from classical strategies that many chess players have memorized. A standard opening move sequence is typically a list of accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

A1: We will store all the opening move sequences in a sorted large file (or categorize them into different files by putting all the sequences that have the same first move in the same file). Every time a move is registered, the program will add it to a stringstream and try to match it to one of the opening move sequences in the file. After all the possible sequences are matched, the program will register the next move by randomly choosing a move from the matched sequences of moves.

Q2: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

A2: First, we have to create a linked list that records the history of the moves made by the players. This linked list will record all the new row, new columns, previous rows, previous columns, and also the opponent's pieces that were captured. We can get these data when the Piece class is sending a notification to the Display class. If the player wants to undo, the program can read the last history, and move the player's piece from new row and new column back to the previous row and previous column. If there is an opponent's piece that is captured in the previous action, the program will read from the history, and initialize the opponent's piece back to the new row and new column.

The history of the moves is also available for unlimited number of undos until the board return to the original state before the game.

Q3: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

A3: The changes include:

- generate four players of human or computer class instead of just two
- modify the Controller class to have four turns for each player
- if this is a 2v2 game, then the players have to specify which team they are in within their class, if this is singles, all players are in different teams
- modify move function so that a player can't attack his/her own teammate
- create a board of the size 14*14, the 3*3 part on each corner of the board is unavailable for the players, the same goes to board on Display class
- during the initialization of the board, players within same team must have their pieces face against each other (the players sit across their teammate)
- the number of chess pieces that can be contained within the board is now 2 times larger (i.e. 64 pieces instead of 32 pieces)
- make modification so that the pawn can be promoted in any "King's row" of the other three players
- for 2v2 game, we need two checkmates instead of one in order for a team to win the game; for singles, three checkmates is necessary and the last player is the winner

Q4: Assuming you have already implemented all the commands specified in the **Command Interpreter**, how would you implement loading a pre-saved game with maximal code reuse?

A4: To load a pre-saved game from a file, we will reuse the code implemented for the setup function. We will create a load() method in the Controller that can use filestream to read the pre-saved game progress from a file. The load() method will loop through the file and keep calling setup() to restore the board status. The load() method will loop until all 64 grids of the board are visited and set and the turn is stored, which reaches the end of the file.