Final Design Document – Chess

By: Wenchao Du and Hao Xue

UML Class Design

See UML in uml.pdf. Most changes to the UML from the previous plan of attack are minor changes to method. All inheritance, composition, aggregation relationships remained the same. The only addition to the UML is a MoveTree class for computer player level 4.

Outline of source files

All classes depicted in the UML have their own respective *.cc and *.h files. In addition to these files, we have main.cc, which will call Game to initialize the game, *.xbm files which are images files for the chessboard and chess pieces, and a Makefile, and ai.cc/h that contains some helper functions for AI Level 1-3.

Overview of Features

This section will highlight the main features of the chess game and describe their implementation. This is in an order similar to the requirements specified in the chess.pdf instructions document for the project.

Rules of Chess

All rules of chess are implemented in this program, including proper castling, en-passant, pawn promotion, check and checkmate detection. The implementation details of all rules with regard to the movement or capture of a piece are put inside each individual piece subclasses, such as castling in King (king.cc) and promotion in Pawn (pawn.cc). When a Player registers a move with ChessBoard, ChessBoard will determine if the move is legal by calling the moving piece's move method, which follows a template method pattern, and registers the move if it is legal, or reject if illegal. (See piece.cc:move() for template method pattern)

Displays

There are graphics and text displays in the program. The TextDisplay (textdisplay.cc) is straightforward, and print out messages on checks and checkmates, as well as AI behaviour for AI level 3. The GraphDisplay (graphdisplay.cc) is a rewritten version of the provided Xwindow class to handle .xbm image loading. Information on image loading using <X11/Xlib.h> was scarce to find, therefore I used a lone example of implementation using Pixmaps to implement this class. It did not integrate well with the existing Xwindow class, so the Xwindow class was removed because it was redundant. In essence, GraphDisplay loads images of chess pieces in to a window and the individual Cell objects will notify the graphical display when they are changed. This is an example of the Observer pattern.

Players

We have all player features described in the instructions document. In other words, the game supports human vs human, computer vs human, computer vs computer. HumanPlayer is straightforward. The class's move method takes moves from stdin until it receives a legal move or "resign". ComputerPlayer (computer.cc) has 4 difficulty levels. A computer player with a higher difficulty level is usually able to defeat a player of lower level.

<u>Level 1</u> – Randomly select a move

<u>Level 2</u> – Prefers capture and checking moves

<u>Level 3</u> – Prefers avoiding capture, capture, and check moves. This is the best AI that computes its move nearly instantly. It is implemented using statistics kept track by the game, in particular, whether a cell is a reachable by each piece on the board. Using this information, it is able to efficiently compute how it can safely capture a piece or perform a check.

<u>Level 4</u> (aitree.cc) – Make moves using concepts from game theory (game tree and the minimax algorithm).

1. The AI first builds a game tree by playing out a fixed number (12) of best moves chosen by a heuristics algorithm for 4 consecutive turns (by making moves and undoing them). In essence, the AI first adds to the tree some best possible moves for itself during every turn, and then after each of these moves, attaches the opponent's best possible countermoves as children of these nodes in the game tree, and so on for 4 turns. This tree building process is obviously slow because it's exponential ($O(m^n)$, where m is the number of "best moves" and n is the depth of the search). The AI compensates for this by limiting m and n, at the expense of not evaluating moves that appear bad but may lead to better results several turns down the road.
2. After the tree is built, I do a post-order traversal on the tree that uses a minimax algorithm to find the move gives the greatest net gain. Leafs are evaluated using standard chess piece values and a modifier for special moves.

This version of the ComputerPlayer is significantly slower than all previous counterparts because it thinks 4 moves (plies) ahead, but will defeat all AI of previous difficulty levels on a consistent basis, especially when it has fewer pieces left on the board (which allows for a more complete game tree). How much slower depends on the system. When ran locally on an i7 CPU, it takes 2 to 4 seconds for it to compute each move. However, I did notice over ssh on school servers, it can take well over 10 seconds.

Command Interpreter

The game supports all commands described in the instructions document (game, resign, move, setup, etc). In addition, the game supports "**undo**", which will undo a player's move. The human player can undo an unlimited number of times, but he or she must use caution because he or she must manually undo twice in order to return control back him or herself because otherwise, it is the opponent's turn and the player will have input control. The command interpreter will check for input validity and the program does not break when an invalid command is issued. However, this does not apply to setup mode in that only the rules stated in the instructions document are implemented.

Scoring

Scoring is implemented in the Game class.

Additional Non-Required Features

In additional to requirements described in chess.pdf, which were all implemented in the program, several other features have been implemented. Undo has been mentioned before as a feature that is not required for the project. Undo is actually central to the development of the AI because the AI will think several turns ahead by actually making a move on the board and undoing them. Graphics with loaded images are also an additional feature, as opposed to graphics with letters of pieces. AI level 4 is of course the final additional feature.

Object Oriented Programming Design

Classes

The purpose and responsibilities of all classes are unchanged from those described in the previous plan of attack. Ideas from OOP including inheritance, polymorphism and encapsulation, are all exhibited in the class design. For instance, the inheritance and polymorphism of Pieces allow for much simpler code when working chess pieces, and this is similar for Players. For encapsulation, all data are kept private if possible, with the exception of the AI having friend access to ChessBoard simply because it would be far more efficient if the AI could have access to these data directly. In fact, only Players and Game have public access to ChessBoard. Access to Piece and Cells are also limited to ChessBoard only.

Design Patterns

The Observer Pattern is the most common pattern employed in this program. Both the TextDisplay and GraphDisplay are observers to the game board, and more specifically, TextDisplay to the ChessBoard, and GraphDisplay to each Cell in the game. The Cells will individually notify the GraphDisplay when they are changed, so this means that cells are redrawn only if they are changed. The TextDisplay does not need this level detail, so it listens to the ChessBoard.

The Template Method Pattern is used for legal move checking for Piece and its subclasses. Specifically, since all moves must not exposed the king to a check and must not result in a player capturing his own piece, the methods for these conditions are common to all types of chess pieces and thus made non-virtual. However, all Piece subclasses move differently, so this checking is made virtual and the subclasses must provide implementation for legal move checking.

Iterators on STL classes were used heavily in the program, and this is an example of the Iterator pattern.

Coupling and Cohesion

Many classes in the program share high cohesion because they share a similar purpose. For instance, Piece and its subclasses share the same purpose, which is to enforce the rules of chess. The same occurs for the Player classes and the AI classes, which are to compute the best move possible, the Display classes, which are to display the chessboard, and the ChessBoard and Cell classes, which both store data related to the game that the other classes may want to use. Therefore, most the program is quite cohesive.

However, there does exist high coupling between a few classes in the program. In particular, ChessBoard, Cell, Piece, and the sub-Piece classes do have somewhat high coupling. This is because the ChessBoard, which owns Pieces and Cells, updates each Cell with information regarding what piece is kept on the cell and whether the cell is reachable by every piece in the game (mostly for AI computations and check detection). In addition, the Cells update the Pieces themselves with their current positions and whether it is threatened or covered by other pieces. This is an example of high coupling because these objects all depend on each other. This is done with a purpose because otherwise, ChessBoard would contain far too much data with regard to the state of the game, and it would need to handle an overly large proportion of the game's responsibilities.

Nevertheless, most modules in the program are loosely coupled, that is, a change in implementation or even interface of a class does not affect the operations of another module. In particular, the display classes, the player classes (with exception of certain AI functions) and the game class are all very independent modules because they do not depend on implementation details of other classes.

Answers to Questions Posed in chess.pdf

The answers for all questions remain the same. Of these three questions, only undo is implemented in the program.

1. Chess programs usually come with a book of standard opening move sequences, which lists 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.*

Since ComputerPlayer already has friend functions to ChessBoard that will use ChessBoard's members to calculate what the next move it should make, it will have access to the list of Moves that each Player has made. So, a new Openings class could be created, which would simply be a trie of moves, where each unique path from the root node would represent an opening in chess. For instance, the Ruy Lopez opening, which is one of the most popular and powerful openings in chess, would have a path with nodes root-e4-e5-Nf3-Nc6-Bb5. The ComputerPlayer would have a new method that would traverse the trie based on the current list of Moves and see if a move from a chess opening is possible. The Openings class would also have an initialization method that will read from a file that will contain the openings and insert the items in to the trie. Such a file would likely have to be created manually.

2. How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of moves?

The main idea here is not destroy or remove a Piece when it is captured or promoted. It will simply be removed from Cell, ie the Piece pointer in Cell will be set to NULL, but the Piece itself will remain in ChessBoard. When an undo command is issued, the ChessBoard will use its list of Moves to undo the most recent move. Such an operation would require resetting the Cells, changing the state the Piece in case it was promoted or captured, and updating the displays. Reverting a basic move is straight-forward. Reverting a capture would simply require adding back the Piece to the Cell to which it belongs. "Unpromoting" would require the promoted piece pointer in Pawn to be reset to NULL. An unlimited number of moves would not be a problem because we keep the entire list of Moves.

3. 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.

This requires a substantially more modifications to each class than the previous two features. Here are some changes that would be necessary to implement four-handed chess:

- There are four instances of Players, which would require modifications to the Game class
- The coordinate system in ChessBoard and Cell would require modification. The move methods in ChessBoard would now check for illegal moves differently due to the "holes" in the corners of the board
- Both display classes would be changed to draw the new board accordingly
- The legal move checking mechanisms and the AI would now have to account for the possible moves of 3 other players, not 1
- Need to account for the team system in four-handed chess (more below)

Some things that we will do to make the transition easier include:

- Not including any code that depend on the colour of the pieces
- One way to circumvent rewriting all the code with regard to capture and checking is to have a Team superclass from which Player is inherited. This way, Player will still overload the move methods, but now the Game and ChessBoard classes will know which Team each Player is on.

Answers to Questions Posed in Guidelines

1. What lessons did this project teach you about developing software in teams?

Continuous integration and revision control is crucial for developing software in teams. Since we realized this at the beginning of the project, we set up a git repository for revision control (svn was not working at that point in time). Frequent pulling and pushing to and from the repository is necessary in order to prevent merging problems that did arise at times during the project. In general, it is best to try to divide work in a manner such that at no times, two persons worked on the same methods or classes at the same time.

In addition, frequent communication is needed for both person to understand where each other is at, and what parts of the program requires additional work. A good initial class design is obviously useful because it allows for the whole team to have the same understanding of how the program will be built. Also, commenting becomes very important when multiple people are working on the same code. Shared test files are also useful because they inform the entire team of what bugs are present in the program and allow whoever is responsible for it or whoever knows how to fix it to fix it.

2. What would you have done differently if you had the chance to start over?

We do not feel that a drastic or radical change is needed if we were to start over because in the end, the program works well and satisfies all requirements of the chess projects. However, certain changes to our approaches could help:

- King could be made a singleton because only one is allowed on the board at all times. This enforces this restriction, but it also makes it special. We have several methods that checks if the king would be checked if certain conditions were to occur, so the distinction of king from the rest of the pieces may actually be a good idea because we would not have to find index for the king in the polymorphic array of pieces for each of these methods. Note that the alternative, which is used in the program, is to specify that the king would always occupy the first position in the array of pieces.
- We could definitely try to loosen the coupling between certain classes in the program. In particular, ChessBoard should probably handle less work with regard to initializing and updating the Cells and Pieces. In addition, there are some methods in Cell and Piece with similar intent that could probably be simplified.

- On a similar note to the previous point, certain classes, like ChessBoard, have somewhat long methods and complicated implementation. Adding more functions and methods could help in certain cases. However, because the AI (in particular level 4) does use some of these long methods a huge number of times, adding many helper calls will likely slow down the program due to the millions of stack copying.
- Computer player level 4 could be better optimized if we had more time. For instance, as opposed to simply evaluating all possible and taking the best moves and form the game tree based on these best moves, we could probably use more advanced ideas described online, e.g. see Computer Chess on Wikipedia). We could also incorporate the functions used by level 3 to get these moves (this though would not make the AI any smarter, but perhaps faster).
- The computer players have trouble checkmating an opponent because it simply does not think ahead enough to make a smart choice. Thus, some kind of end game specific algorithm could help.