Chess: Final Design Document

Dhron Joshi(d8joshi), Haowei Guo(h48guo), and Rishabh Malhotra(r22malho)

July 26, 2016

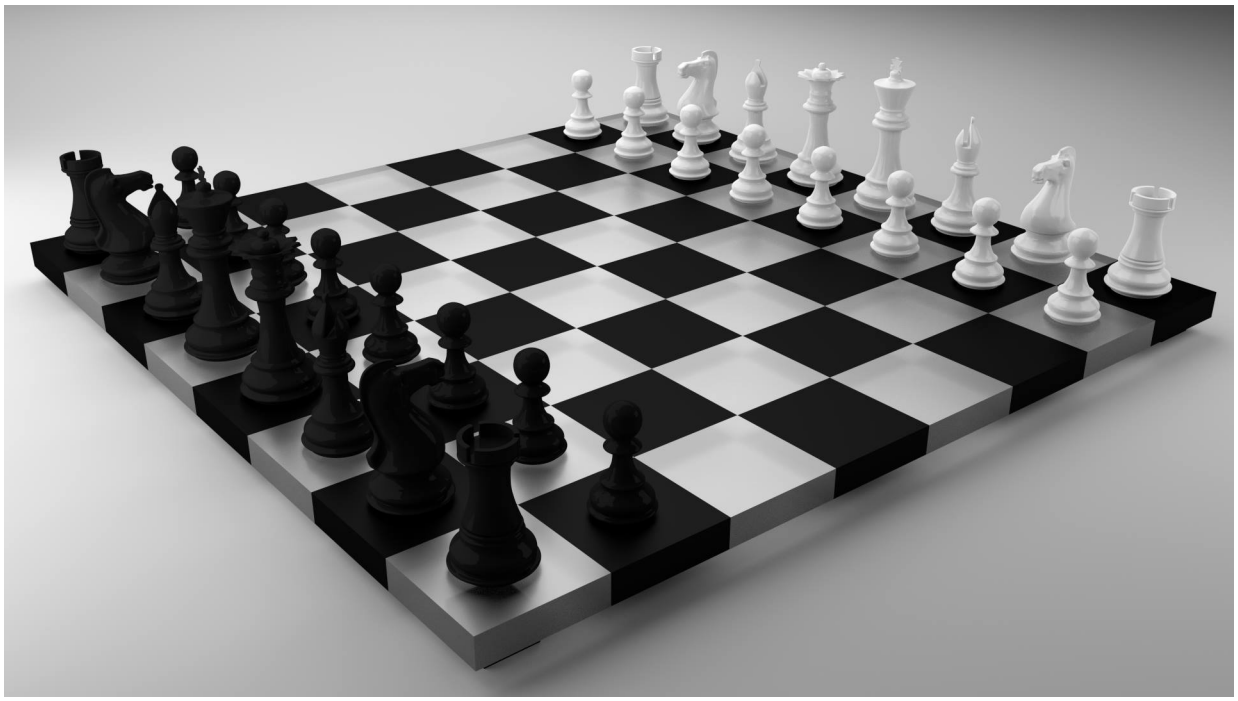


Image retrieved from

http://unconfirmedbreakingnews.com/wp-content/uploads/2014/09/chessboard1.jpg

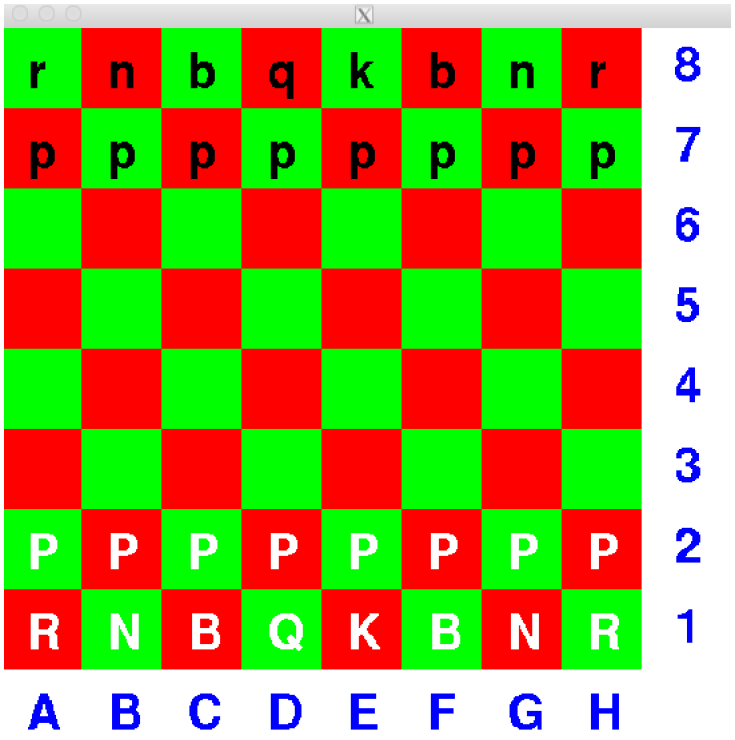
2 Implementation Methods & Use of Design Patterns

- Observer Design Pattern
  - The chess game functionality was entirely done by using the observer design pattern. This is because we felt the majority of the chess game is updating the board based on certain events, and we could encompass player movement, text and graphics display, and updating cells all in one design pattern. We decided that there was no need to include any other design patterns besides the observer. Part of designing and implementing good software is knowing what types of design practices best fit the goal you are trying to achieve; we believe only the observer design pattern was needed to accomplish our goal.

- Player Movement and Legal Piece Movement
  - The player movement is implemented by having each subclass of the `Piece` class to contain a vector of it's legal moves. When a piece moves, there is a loop that checks all legal moves of the piece and attaches it to it's vector of legal moves. Then when a move command is issued, we check whether that position exists within our vector, and if it does, we move the piece. Each legal move for each piece is determined by the freedom of the cell on the board, and the unique movibility of each piece.

- Drawing and Updating Display
  - The text display belongs to the `TextDisplay` class. The character components of the text display are stored in char vectors and are initialized in a standard chess board format within our TextDisplay constructor. The display is updated through our `comeNotify(), (line 58, display.cc)` and `leaveNotify(), (line 88, display.cc)` functions which alert the display to update whenever a piece is moved to a different coordinate on the board. We also overloaded the bitshift operator `<<` to output our TextDisplay in our main function as we needed.

    As the text display runs, we also implemented the `X11/XWindow` library to create a graphical window that represents the pieces on the board.



  - A graphical view of our board using X11.
  - Two loops are used to draw the axes and are drawn during the creation of the window. The green and red squares are also drawn upon creation. The pieces are continuously redrawn as they move around the board upon the calls of `comeNotify()` and `leaveNotify()` as discussed above.

- Pawn Promotion
  - This was accomplished by having an if statement check whether the pawn of the opposite colour reached the opposite side of the board. If the player is a human, we give the option to choose what piece you want the pawn to be; if the player is an AI, it chooses the queen by default. When the piece transforms, we remove the current piece and create a new piece from the subclass of the desired type, and so it possesses the correct properties of that piece.

- En Passant
  - We have a condition that checks for the availability of the en passant move, and requires that the attacking pawn was moved into the position before the attack one move earlier. If these conditions are met, the en passant move is added to the legal moves of the pawn that wishes to attack, and then it may move in the position the pass the opponent pawn. Once that pawn moves, the en passant move is completed and the opponent pawn is removed from the board.

- Castling

  We check whether the king and the rooks have been moved before using our `stepCount()` method and that they're in their original position. If they haven't been moved and are in their initial position, and there are no pieces between the king and the rook; if all these conditions are met, we attach the coordinate for castling in our `legalMoves` vector for our king, and then the king is able to castle. After the king has moved, we also move the rook to its corresponding place on the grid.

- AI Superclass

  - The AI is the superclass that controls all the computer played levels. It has Level1, Level2 and Level3.
  - The AI specifies protected fields like objects of the Grid class, the colour that the AI is playing as and a vector of all the pieces that the AI has on the board at any given state of the game.
  - Also, the AI superclass specifies important methods like the pure virtual `makeMove()` function, a function that generates a random number in a given range and another function that gets a random valid piece and supplies it to the AI to consider for movement.

- Level1 AI

  - The level1 is a subclass of the AI superclass. It is the level where, the computer just makes legal random moves and doesnt exactly analyze whether it is making an intelligent move or not.
  - The Level1 class has `makeMove()` method that generates a vector of vectors which it then passes to the `generateMove(<vector <vector> >)` function and then the `generateMove()` function generates a move.
  - If this generated move is legal (valid; obeying the rules of the game), then the `popCell()` method of Level1 pops the small vector (one of the vectors inside the bigger vector) corresponding to this move from big vector.

- Level2 AI & Level3 AI

  - These levels are also subclasses of the AI superclass. Level2 is the level where, the computer tries to capture the opponent pieces and prefers capturing moves over just legal random moves like in Level1 so that it is a bit higher in difficulty.
  - Level3 is the level, where the difficulty actually rises because the AI in level3 doesnt just prefer capturing, rather, it plays defensively and aggressively. So, if a black bishop is diagonally opposite to a white pawn, instead of just moving forward to be in a safe zone, the site pawn would actually capture the black bishop (its predator).

- Also, Level3 has foresight, which means that it analyzes 2 moves at any given time; the move at hand and the prospective opponent move after the present move has been made. So, for this it uses various important methods of the `Grid` and the `Piece` classes like `attackRange()` or `legalMoves()` for a Piece and the `reAttach()` method of the Grid which makes itself useful in the case that an opponents pawn makes it to the end of the board and is replaced by another piece (Piece).
- For this, these classes use the methods `randomMove()` which generates random legal moves (to be used in the scenarios where the computer doesnt have anything to capture from the opponent or is not in any kind of danger in the present scenario) and `makeMove()` which does all the work of analyzing and seeing whether it can find anything worth capturing or defending or else it just calls `randomMove()`.

3 Accommodation of Change

We had a very easy time adding in new methods and functions without changing any of our old implementations, and this allowed for us easily expanding and making methods to accommodate our AI classes.

Our Cell, Piece, Grid, and Display classes all have high cohesion as all of the functions implemented within the respective classes only perform actions pertaining to that class and provide a general interface to accessing information about that class; which is why we were able to give the AI classes the information that it needed to be implemented properly.

For example, when we wanted to implement an undo function so we can revert the state of the board, we already had a stack of moves in our grid class because the grid needs to keep track of the moves completed to maintain it's state. The only changes we had to make was to pop the move off the `record` vector and update the grid accordingly.

We have also minimized coupling by using proper design styles such as encapsulation, we've minimized the number of friend classes to classes that absolutely require the members of a specific class. This also contributes to our accomodation of change because each individual class works entirely whenever we add or take away features from our program.

4 What we got to learn besides the Obvious

We definitely learned the importance of effectively communicating your work to others. We often found ourselves asking what eachother's code did, and we found that if we could explain it effectively, it aided us in our debugging process. We learned about leaving com-ments in our implementations so others can follow and understand that we implemented; this improves readability and understanding for every member of our group.
We also learned how to consistently use version control (git) and maintained a document that acted as our TO-DO list, and so even if we worked at different

times, we would continuously update the repository to keep track of what we needed to accomplish.

We also learned about the importance of planning when working on a large project like this, and allocating our time and resources effectively so we don't waste any time during the programming phase. We made quite a few changes to our UML document by the end of the project compared to the initial document.

In conclusion: collaboration, communication, planning, and time management are necessary skills when taking on a large software project such as this and we learned a lot about each of these aspects during the planning and development phase.

What we would have done differently if we had the chance to start over?

We definitely would have planned out the UML document better to account for the AI class from the very beginning unlike this time. If we had done that in the beginning, the AI development would have been completed much quicker than we had planned.

We would also look into smarter ways (more efficient ways) to implement each level of the AI. In our implementation, we have to analyze every situation and every piece on the board to make a proper decision on what to move; this becomes very taxing and inefficient: we would do some more research into the logic behind Chess AI before actually trying to implement something more sophisticated. Perhaps we would actually implement a chess engine to make sophisticated decisions for us such as `Stockfish` which we found on a Google search.

Also we would look into better graphics libraries or game development libraries that allow us to have more colours and have options for loading images and displaying them to the screen. We felt that X11/XWindow was very limited in terms of colours and not having any functionality to import images and to display/draw them onto the screen.