Design Document

Controller:

In general, the design of our chess program followed the MVC design pattern very closely.
The Controller class was the central part of the program and controlled game flow along with user interactions. More specifically, the Controller::play() method handled gameflow. It would start by waiting for user's input such as "game" or "setup" as examples. If "setup" was typed in, the method would go into a while loop that controlled how setup mode worked. A new Board would be instantiated and the while loop would run, accepting the the user's inputs and using a method Board::placePiece() that would either add or remove a piece from the Board, depending on the command. For each command that the user entered, the validity of the commands were always checked. When the "done" command is typed in, the program would check the validity of the board and act accordingly. If the board was not valid it would output an informative message and restrict the user from exiting setup mode. Otherwise the program would exit the setup mode while loop and wait for the next command to be entered in.

Importing saved game sessions was taken care of through a method in Controller called Controller::importGame(). If a command line argument for a file was read in, in Main(), it would call Controller::importGame() to import the saved game session.

When the "game" command is typed in, Controller will first decide what Board to use. If a game was imported or setup mode was previously used, then Controller would use the current board as that would be the custom board the user created. Otherwise Controller will create a new Board for gameplay. At this point the actual players of the game are read in ("human" or "computer") until valid players have been read in for both white and black sides. Now Controller will only accept two inputs, either "move" or "resign". If "resign" is entered, then variables dictating the score are updated accordingly, the current game session ends, and Controller goes back to it's neutral state, waiting for the next command. If Ctrl-D is pressed, then the main while loop will exit, the score will be printed and memory freed. However if the "move" command is used the Controller acts according to multiple factors (colour,human/computer etc..). So the way moves are executed are through a method Board::makeMove(). In the case of a human player, input would be directly read in (until valid input is observed) and the move information would be directly sent into the Board::makeMove() method. In the case of the computer, a method AIplayer::bestMove() would return the computer player's best move choice and that move information would be sent directly to the Board::makeMove() method.

Board:

The Board class was in charge of keeping the state of the board along with everything on the board (ChessPiece's) and all of the methods to modify board state, deduct information from a board state, and to grab information about the board state. The controller would use many methods in Board such as Board::isGameOver() to figure out what was currently happening in the game, and thus

Board is actively communicating with the Controller. Board also owns all the ChessPiece's and is able to obtain information from each individual chessmen for its own methods. Board will often update ChessPiece information and these changes are then observed by other methods also within Board.

In terms of how the board is drawn, Board owns a TextDisplay and has a method called Board::draw() (it sends the board state information to TextDisplay) that the Controller calls to update the textual display. For the graphical display, based on where the information that we needed Information to update the display) was located, we decided to give the Controller ownership of the display, as compared to the Board. This helped us avoid unnecessary complications in the code responsible for drawing the board along with avoiding having to change any code in Controller and Board.

When controller calls the Board::makeMove() method, Board will go into the ChessPiece at the corresponding spot and obtain a vector of all of the ChessPiece's possible moves. It will then search through the list and act accordingly based on the legality of the move.

ChessPiece:

The ChessPiece class was the super class to all the subclasses Queen,King,Bishop,etc. It has a pure virtual method updateMoves() that was implemented by each individual subclass (since each chessman moves differently). This also helped simply code for Board as it would only need to call updateMoves() and through polymorphism the correct method would be called. ChessPiece also "has a" Board, this is crucial as legal moves depend on the state of the board.

AIplayer:

Similarly, AIplayer was the super class to all the subclasses of the different level of computer players (level1, level2, etc). The controller owns AIplayers and would call the pure virtual method AIplayer::bestMove() to obtain the best move. By making bestMove() pure virtual, the correct method would get called dynamically and resulted in better designed code. The AIplayer also "has a" Board as this is needed for the computer player to create hypothetical board situations to analyze the best move.

Question: What lessons did this project teach you about developing software in teams? If you worked
alone, what lessons did you learn about writing large programs?

Answer: By working in a team you quickly realise the importance of communication between team members. Without communication the entire project starts to fall apart as code will get coded twice and time isn't managed efficiently. This is incredibly important due to the fact that we're working within a strict time limit. You also learn proper methods to share code between members (through git). However, the most important thing working on this project taught us was how to debate approaches and ideas to solving problems. Often both teammates would have different ideas how to approach problems and this would result in a pro/con argument and could often result in us realising an entirely new and improved approach to a problem.

Question: What would you have done differently if you had the chance to start over?

Answer: In general I believe that our approach to creating the game was very well thought out as me and my teammate have done collaborations before. However the main difference is that since this was school related we had a much more restrictive time limit. This resulted in us being rushed at times, and thus unable to produce fully tested code. For example we would finally have a newly implemented function and because of the restrictive time limit, we would move onto other functions that branched from that original one, instead of thoroughly testing it. This would result in bugs being carried in through and make debugging code much harder. So if we had a second chance at attacking this project, we would definitely make sure that all our newly written code is bug free before moving onto the next steps.