Estimated Completion Dates

Task	Description	Date
UML Diagram	Finish UML Diagrams, think about the approach to meet project specifications	16 Jul
Plan of Attack	Complete planning document, setup github repo	17 Jul
Creating Files	Create all interface and implementation files necessary for the program	19 Jul
Build the pieces	Build the inheritance hierarchy and the methods required for each piece	21 Jul
Build MVC	Build the Observer design pattern and the MVC to display the board	21 Jul
Build Controller / Board	Link the MVC and the pieces together by building the board and controller, and the methods associated with it.	21 Jul
Human vs Human	Using the tools previously built, complete a human vs human test run using the text interface	22 Jul
Build AI Level 1	See below	22 Jul
Build AI Level 2, 3, 4		23 Jul
Extensive Testing	Use the setup command to create scenarios to ensure that chess rules are followed in the implementation.	24 Jul
Complete Documentation	Write documentation as changes are made	25 Jul
Submit Project		25 Jul

Partner Responsibilities

Zaid

1. Main file
2. Controller
3. Observer

Talha

1. Pieces
2. AI

Jason

1. Pieces

2. Controller
3. Main file

Style Guidelines

- 4 space tabs
- Camel case
- else if / else on new line
- No spaces in at the start and end of parameters

Project Specifications

Handling Display (Text / X11 Window)

This program uses the MVC (Model, View, Controller) structure implemented using the Observer design pattern. As shown in the UML diagram, the two forms of displays inherit from an abstract observer class. The chess board has a class which inherits from an abstract subject class. Whenever the board changes, it will notify its observers. The text view implements the notify method by writing the board to standard output. The window view will draw to an X11 Window instance, similar to the one used in a4q3.

Handling User Input

The controller will be responsible for handling user input. The controller will prompt the user with a set of possible instructions to input and the user can input their instructions through the command line. We will have checks in place to ensure the program does not crash given invalid inputs.

The setup command is a method of the controller class. When the user places a piece, the board's placePiece function is called with a pointer to a newly initialised Piece object of the given type with the correct colour (starting from white) and the position given by the user.

Checking for if move is valid

Each piece has a `getPossibleMoves()` function to generate a list of possible locations it can go excluding capturing other pieces. We then have `getPossibleCaptures()` to generate a list of potential moves that will lead to capturing an opposing piece. In the board, we will then have an `isValidMove(Move m)` function that consumes a Move returns a boolean that checks for the following,

1. Checking if the move is a valid move given the constraints of the type of the piece by invoking `getPossibleMoves()`
2. If you are currently in check, run `getPossibleCaptures()` to ensure your move will get you out of check

3. If you are currently not in check and is not a King, run `getPossibleCaptures()` on every opposing piece to check that the King is not in any of the lists (Ensure the move will not lead to a check)
4. If you are currently not in check and is not King, run `getPossibleCaptures()` on all pieces to check that the King is not in any of the lists (Ensure the move will not lead to a check)

En passant

When a pawn makes its first move, if it moves two spaces, it will notify pawns to the left and right of it that an en passant move is possible (by changing a boolean attribute). Then when `getPossibleMoves()` is called, the output will include the move if the en passant boolean is true.

Castling

If the king has not moved, `getPossibleMoves()` should have an extra move if the space is clear. If the castling move is made, the king moves two spaces, and the correct rook moves next to the king (left castling moves the left rook one space right of the king, right castling moves the right rook one space left of the king).

Handle Checkmate

We will use the `isCheckmate()` that returns a boolean. If you are currently not in check, return false. If you are currently in check, generate all possible moves the player can do with all pieces with `getPossibleMoves()` then run each move against the opposing pieces' `getPossibleCaptures()`to check if the King is contained in it. If so, return false. Otherwise, return true and terminate the game.

Handle Stalemate

Using the above algorithm for valid moves, if both sides have no possible legal moves. It is considered a stalemate

Computer Level 1 - 4

Level 1: Randomly find a piece and invoke its functions to generate a legal move, then randomly choose a move.

Level 2: Perform a random capturing move (from `getPossibleCaptures()`)**.** If there are none available, find a move (from `getPossibleMoves()`) that results in a check and perform it. Otherwise pick a random move.

Level 3: If there is a potential capture, find a move that escapes the capture. If given the opportunity to capture an opposing piece, it will capitalise on it.

Level 4+: Using an algorithm that considers depth e.g. stockfish

Keeping track of score

The scoreboard object will be responsible for keeping track of score. Everytime a checkmate happens, the winner's score will be incremented by one.

Other

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.
The book of standard opening moves can be stored as a json file. The information can then be stored in a tree and the computer can read from it to determine the best move. The book of standard opening should contain the first 5 - 10 moves. Otherwise, traversing the tree will be very computation heavy and slow.

Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
The program has a move class which describes how the board changes (piece moved, piece captured, previous and current position). Whenever a move is made, it is appended to the board's 'moveHistory' vector. If the user wants to undo, the board can access the last element of this vector and move the piece back to its original position. If a piece was captured, it can be readded to the board.

Question: 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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?

In four player chess, the behaviour of pieces does not have to change much but the board class would be quite different. The display would also have to include a way to differentiate between 4 different colours instead of just 2. The controller class would need to support cycling between 4 different players (including setup mode). Instead of counting wins, there can be a scoring system (checkmating players, taking pieces).