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

Introduction

What is chess? Apparently a miserable game to program.

Overview

Board class (Model)

The board class acts as the model in the MVC structure of the program. It has access to all the pieces and has methods to execute moves and check for stalemate and checkmate.

Piece classes

Each piece type has a class which inherits from an abstract class. These classes implement the virtual 'getPossibleMoves' and 'getPossibleCaptures' methods, based on the moving rules for the piece.

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.

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.

Design Patterns Used

- Observerer design patterns used for display (MVC)

Design / Compare and Contrast

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

Planned:
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.

Final Product:
En passant went mostly as planned. When user is entering a move, we first perform a check if it is an en passant if it is 1) is a pawn 2) the piece horizontal to it is a pawn 3) the pawn is

going (newX, oldY). Then as planned, getPossibleCaptures() generates an en passant move, and we check it with a loop to compare if the move is valid and is an en passant.

Castling

Planned:
If the king has not moved, `getPossibleMoves()` should have an extra move if the space is clear (and not being attacked by an enemy piece). 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).

Final Product:
Castling was tough to handle in the isValid move method. Due to the unique nature castling of and the conditions under which it can occur we decided to completely re approach how we implemented chess. When a move is passed into `playMove()` it does all its basic checks to decide if a move is valid, like if it is your turn or not. But before it actually enters the other `playMove()` method it first checks the castling methods. `checkCastling()` returns a boolean. It does an extensive series of checks to see if the conditions of castling are met. This includes: are you trying to move the correct pieces, are there pieces in the way of castling, are the squares you want to castle being attacked by the opposing colours, etc. If any one of these checks fail the method returns false and it enters the normal `playMove` check. This is important because then `playMove` can handle and tell the user that move is invalid rather than reimplementing the logic. If all the checks pass then castling is valid and the castling method returns true moves the pieces accordingly and prematurely exits the function

Promoting

We did not initially consider pawn promotion. We added a check that if the white pawn reaches BOARD_MAX_HEIGHT - 1 or if the black pawn reaches BOARD_MIN_HEIGHT, we prompt the user for a pawn promotion. The user can use Q, B, R, N to choose which one to promote and the piece gets initialied and properly replaced on the board. If it is a computer on the other hand, it automatically defaults to the Queen piece as it is commonly known as the strongest piece on the chessboard and we have gave it the most weight in the piece.getValue() function used to determine weight.

Handle Checkmate

Planned:
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.

Final Product:
It is exactly as planned as aforementioned.

Handle Stalemate

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

Final Product:
It is almost exactly as planned. We first check if the current king is in check, and then if there are any moves available. If both conditions are met, it is a stalemate.

Computer Level 1 - 4

Planned:
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

Final Product:
Every level of bot is an improvement then the previous version of the other bot.

There was a getRandomMove() implemented that is used in every bot and is crucial to its working. getRandomMove() generates a seed then uses the algorithm library and uses uniform probability distribution to choose a move from the vector of moves provided.

Level 1:
This was the most straightforward bot to program. It simply takes all possible capture and regular moves and filters them compiling a list of legal moves. This level of bot has zero preference and is as random as it can get.

Level 2:
The level two bot is designed to first prefer moves that put the opponent in check. Then capturing moves, and if both of those aren't options then it defaults to a random move. The level 2 bot will always beat the level 1 bot despite not being that sophisticated because its level of checks is enough to beat pure random choice made by bot 1. For getting check moves it is done by getting all the capturing moves possible. It then filters for capturing

moves that attack the opponent's king. If that list of check moves empty then it returns just capturing moves. If capturing moves is also empty then it returns a random legal move.

Level 3:
This bot is designed the exact same way as bot 2 with one difference that gives it an edge. This bot avoidsCapture. After looking for checking moves it checks for avoid capturing moves then capturing moves before finally defaulting to any random legal move. Avoiding capturing moves takes the opponent's pieces and checks every piece that the opponent attacks. These pieces are friendly pieces and are put into a list. This list is first used to generate capturing moves. This is important because that means pieces under attack will retaliate before running away. If that is not an option, it will try to move out of the way. If that is also not an option then it proceeds like bot 2 does.

Level 4 and 5:
This is the final level of bots. The magnus carlsen of our little chess world. It is a huge improvement over the previous bots. Previously, bot 2 and 3 were close although 3 was consistently performing better. Bot 4 introduces intrinsic value to pieces and this is used to affect the decision making of the bot. Bot 4 takes the code from bot 2 and bot 5 does the same with bot 3 but now applies the value of pieces to the decision making process. Pawns are assigned the value of 1 bishop and knights 3 rooks 5 queen 9 and king infinite value. Now instead of choosing a random capturing move it will choose a capturing move that captures the piece with the highest value. Or for both 5 instead of just avoidCapture of a random piece it will specifically avoid capture and prioritise pieces of higher value. This obviously outperforms the other iterations of the bots.

Resilience to Change

Answers to Questions

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?

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).

Final Product:
All of our code can be easily reused for a 4 player chess. We would only need to change the board.cc slightly and the controller. The scoreboard can also be changed slightly to facilitate for the change

Cohesion and Coupling

We ensured that there is high cohesion and low coupling. All of our objects are only passing around primitive types and arrays or structures. We also made sure all the objects has a purpose and they work together to achieve the same goal.

Additional Features

- Computer Level 5
    - More advanced level of level 3 with better capture decisions and avoid capture decisions
- command line argument (-nogui)
    - Does not display GUI and only prints in the CLI
- command "cvc"
    - Automatically runs the game given that it is "game computerX computerX" until game end state

Final Questions

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

This project was instrumental in teaching us software design. We first did initial planning in forms of writing design documents and UML diagram, giving us a basic idea of our implementation even before writing our first line of code. This brainstorming gave us a clear roadmap of our tasks distributions and the where to start for the project. After creating DD1, we immediately created all the header files necessary so we can each start working on our assigned sections. We worked on distinction sections to ensure there is not much overlap in our code.

During our design, we ensured that we have high cohesion and low coupling. We also ensured that we mostly only pass through primitive types or structures for low coupling. This will ensure code reuse and maintain will be easy, and we can easily pick up where we left off.

For version control, we used Git. This project has reinforced the Git skills we learnt in CS 136L. We each worked on different branches and we merged to the main branch we ensure our features are thoroughly tested. We also peer reviewed our code to ensure that our logic makes sense and our code is in good style.

We also learnt to document code properly. We initially wrote code with a lack of comments which makes it very hard for others to read and understand your logic, making it very difficult to add your existing code. We had a team discussion and we ensured that we all wrote proper documentation and comments throughout the code so that we it is easier to maintain and reuse.

Finally, we also explored various c++ libraries that we have not learnt in class. For instance, when we generate random outputs for the computer. Our understanding of OOP and C++ proliferated significantly, making us a much better programmer overall. We can now create design patterns and inheritance etc. with ease.

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

Time management would definitely be something we would have done differently. We started the project near DD1, leaving us with a little over a week to write the entirety of chess. We underestimated the scale of building the entire chess program. If we could have done the project again, we would have started on the very first day the project was announced. Given that we have more time, we would have added more additional features like showing move history, undoing moves etc.

As a team we felt that the bots were very rough around the edges and brutish in their approach to the game, although they did perform the tasks they had in mind.

Starting over with the bots I would use a minimax algorithm that uses depth. This would be a huge improvement to the bot for several reasons. The minimax algorithm would simply be superior to the current versions of the bots we have but the main thing would be that we can control the depth of the bot. This means that we can program a single bot function and pass its depth as a parameter which controls its complexity and level. For example bot 1 would be depth 1 each bot depth by 2. The only drawback of this is as depth gets higher it would get quickly computationally heavy but would be a very elegant and comprehensive solution to the ai portion of the chess game.