## Introduction

Chess is a tabletop board game consisting of 16 pieces, played on an 8x8 checkerboard. By the instructions provided within the CS246 A5 Chess Project guidelines, this checkerboard is laid out with the white square at position “H8” in the bottom right. The game follows the rules outlined by traditional chess, and the end goal is to win the game by forcing your opponent to make a series of moves that results in a checkmate. A draw is determined by a stalemate, a resignation by one player results in a win for their opponent, and score is kept between each color (“White” or “Black”). A computer player has been designed to play this game as well, either against another computer, or against a human. 2 humans may also play against one another.

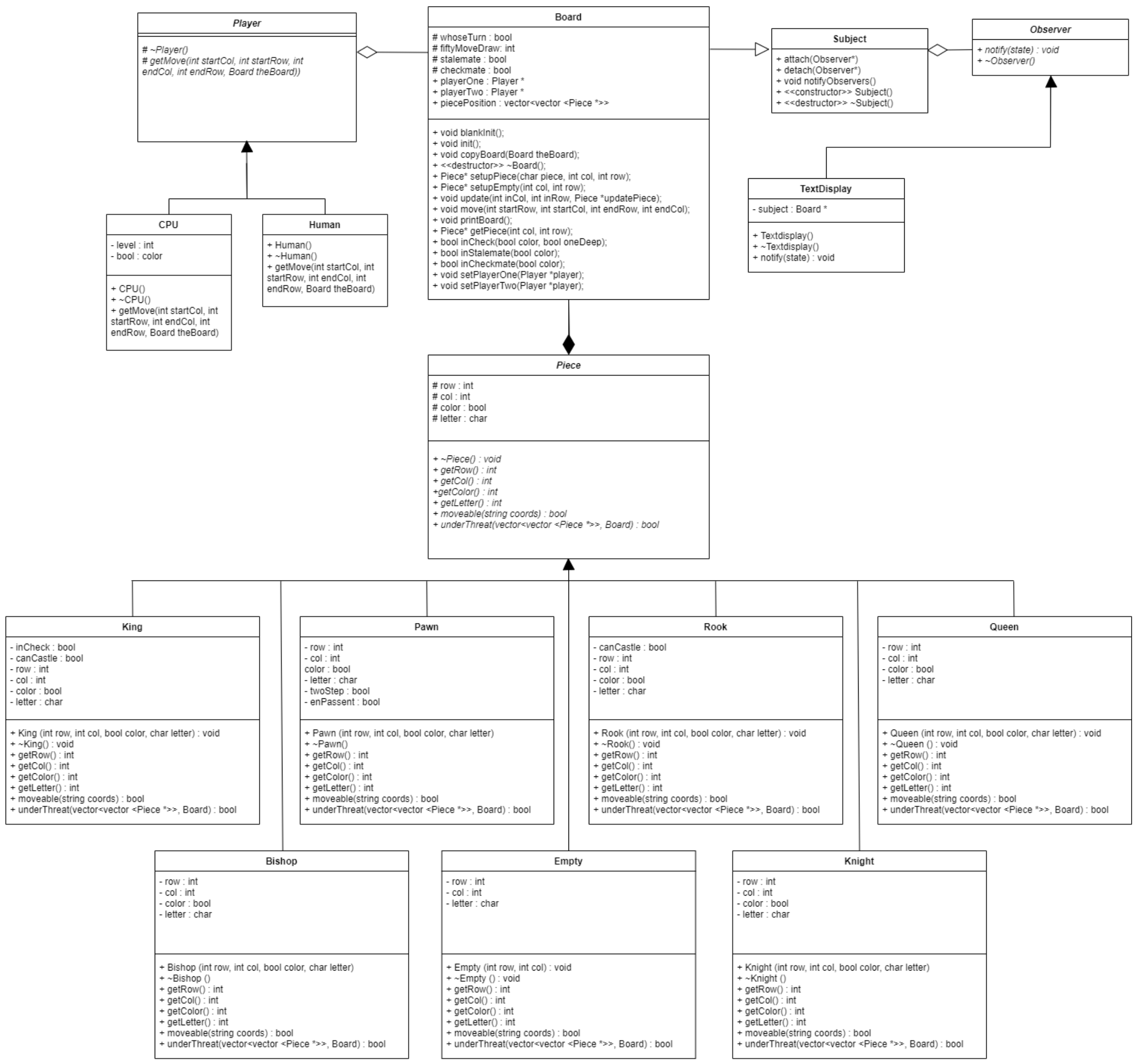
## Chess Overview

We have implemented a substantial number of features to the game of chess in our project written in C++. Our board may be set up according to conventional chess setup rules, with a king, queen, 2 bishops, 2 knights and 2 rooks in the back rows of the board for each player, as well as 8 pawns in front of the aforementioned pieces. The game is played on a text-based board, where a hypothetical graphical interface could have easily been attached through the existence of an observer design pattern. Pieces can be manipulated by both a human player and computer, to make only legal moves as deemed by the rules of chess. Missing regulations and interactions will be discussed in later sections. Game termination conditions can be determined and resolved by the program, new games can be started afterward, and score can be kept between the number of wins for “White” and number of wins for “Black” before reaching end of file (EOF).

The board class hosts the locations of every piece, as well as most functions for manipulating the positions of each of those pieces, such as Board::move(), which manipulates a piece to move from its current square on the board to another. The Board also acts as a concrete subject in the observer design pattern, coupled with an abstract observer class, subject superclass, and text display concrete observer. Additionally, this was done to accommodate for a planned graphical observer, which was not implemented due to time constraint and a desire to achieve other features first.

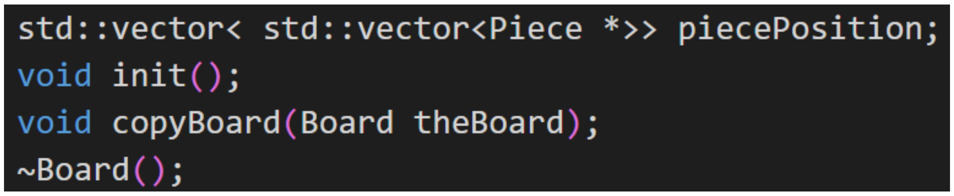
Each piece has its own class, a child class of the Piece abstract superclass. Each Pawn, Rook, Knight, etc. is handled as a piece by the board, but has its own unique movements defined within each of their respective classes. In Main, these functions are called to determine if each Piece can be moved on the Board.

## Updated UML


Player
# ~Player()
# getMove(int startCol, int startRow, int endCol, int endRow, Board theBoard))
Board
# whoseTurn : bool
# fiftyMoveDraw: int
# stalemate : bool
# checkmate : bool
+ playerOne : Player *
+ playerTwo : Player *
+ piecePosition : vector<vector <Piece *>>
+ void blankInit();
+ void init();
+ void copyBoard(Board theBoard);
+ <<destructor>> ~Board();
+ Piece* setupPiece(char piece, int col, int row);
+ Piece* setupEmpty(int col, int row);
+ void update(int inCol, int inRow, Piece *updatePiece);
+ void move(int startRow, int startCol, int endRow, int endCol);
+ void printBoard();
+ Piece* getPiece(int col, int row);
+ bool inCheck(bool color, bool oneDeep);
+ bool inStalemate(bool color);
+ bool inCheckmate(bool color);
+ void setPlayerOne(Player *player);
+ void setPlayerTwo(Player *player);
Subject
+ attach(Observer*)
+ detach(Observer*)
+ void notifyObservers()
+ <<constructor>> Subject()
+ <<destructor>> ~Subject()
Observer
+ notify(state) : void
+ ~Observer()
TextDisplay
- subject : Board *
+ Textdisplay()
+ ~Textdisplay()
+ notify(state) : void
CPU
- level : int
- bool : color
+ CPU()
+ ~CPU()
+ getMove(int startCol, int startRow, int endCol, int endRow, Board theBoard)
Human
+ Human()
+ ~Human()
+ getMove(int startCol, int startRow, int endCol, int endRow, Board theBoard)
Piece
# row : int
# col : int
# color : bool
# letter : char
+ ~Piece() : void
+ getRow() : int
+ getCol() : int
+getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
King
- inCheck : bool
- canCastle : bool
- row : int
- col : int
- color : bool
- letter : char
+ King (int row, int col, bool color, char letter) : void
+ ~King() : void
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Pawn
- row : int
- col : int
color : bool
- letter : char
- twoStep : bool
- enPassent : bool
+ Pawn (int row, int col, bool color, char letter)
+ ~Pawn()
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Rook
- canCastle : bool
- row : int
- col : int
- color : bool
- letter : char
+ Rook (int row, int col, bool color, char letter) : void
+ ~Rook() : void
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Queen
- row : int
- col : int
- color : bool
- letter : char
+ Queen (int row, int col, bool color, char letter) : void
+ ~Queen () : void
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Bishop
- row : int
- col : int
- color : bool
- letter : char
+ Bishop (int row, int col, bool color, char letter)
+ ~Bishop ()
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Empty
- row : int
- col : int
- letter : char
+ Empty (int row, int col) : void
+ ~Empty () : void
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool
Knight
- row : int
- col : int
- color : bool
- letter : char
+ Knight (int row, int col, bool color, char letter)
+ ~Knight ()
+ getRow() : int
+ getCol() : int
+ getColor() : int
+ getLetter() : int
+ moveable(string coords) : bool
+ underThreat(vector<vector <Piece *>>, Board) : bool

## Design

Our group found the UML diagram from Due Date 1 to be a big asset in deciding on our final design. We already thought through most of the relationships between classes and various objects, and tried to bring our ideas to life through code. Our design journey began by creating our Piece class, Board class, and Main.cc file, as those were the most basic foundation of our project design. We started off by designing our Board class, as the structure of our board would be crucial for any piece placement, movements, and functions arguments moving forward. As seen in our code snippet below, we settled on the idea of a 2D Vector of Piece pointers, so we could easily iterate through the 2D Vector, and it would be graphically similar to the way a real chess board is. This would also allow us to easily pass our public 'piecePosition' 2D Vector to any function calls.



```
std::vector< std::vector<Piece *>> piecePosition;
void init();
void copyBoard(Board theBoard);
~Board();
```

Following the limited implementation of our Board class, we began to diversify into the different Piece Subclasses, namely Bishop, King, Knight, Pawn, Queen, and Rook. Creating these subclasses in separate .cc and .h files gave us the ability to work on different files simultaneously and tackle each pieces' movement in a time efficient way. Within each of the .cc files for the pieces, we established the necessity for the instance variables row, col, color, and letter for each new piece subclass object. These variables would permit us to easily access each piece's coordinates on the board, move the piece, determine i's color, and get its representative letter for easy printing. As our board and pieces started to come together, we solidified the list of all our Piece class virtual functions. This included the destructor, functions to access instance variables (getLetter, getRow, getCol, getColor), and a boolean function to determine whether or not the piece object is moveable given the current Board conditions.

A specific design challenge we faced with the Moveable function in each Piece subclass, was figuring out if the piece movement would leave the player's king in check. Originally, our

Moveable function's only arguments were row, column, and our board object. But, given this king check issue, we needed to change the function. Our team members came together to brainstorm possible solutions to add this feature. For example, another 2D Vector that contains all the pieces that are currently watching the king. In the end, we settled on creating a copy of the current Board, making the move we are trying to check, and seeing if this move resulted in a check.

Many other smaller design issues arose throughout our project, such as whether to pass pointers to objects, printing the chess board according to the a-h, 1-8 system, the observer design pattern implementation, problems with memory allocation, and more. Our team tackled each of these issues through communication, peer code reviews, constructive feedback, and extensive testing in Putty.

## Resilience to Change

Our program is rather flexible to development. With the ManualSetup mode, we allow the board to be manipulated at all times, and different modes may be developed through adding other handlers for modes beyond the existing "game" and "manualsetup" modes in Main. In addition to a text display, our observer design pattern accommodates a graphical display that can be created as a concrete observer, accessing and calling the same variables and functions, and utilizing libraries such as XWindow to produce the graphical display. Any number of CPU levels may also be added to the program, beyond the existing 2, and the suggested 4. Given an API, algorithm, or supercomputer, a hypothetical "5th level" could be achieved using a chess engine such as Stockfish, or IBM's Deep Blue.

ManualSetup allows any number of any piece to be placed on the board, given that there is no more than one king for either color, and the board may fit every piece. No pieces may share the same position either.

Provided that bounds are known, the existing program could operate on a board of any size. E.g. a 7x7 board, 4x8, or any number of dimensions. The rules for a standard game of chess would have to be altered, but the code remains flexible enough to withstand such a change.

The Player class is also resilient to change. Existing are a human and CPU player. Should a third type of Player be required for a different game mode (such as an “alternating” mode, where 2 players each take alternating turns playing for the same side) this could also be easily achieved through creating a new child subclass to the Player parent class or manipulating the Board class and Main to take multiple players.

Additionally, given a vector of players in Main, multiple humans and CPUs could participate in essentially a “tournament” or conglomeration of games on the same run of the program.

## 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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.**

The best way to implement such a feature on our program would be to create a new child class of Board to store a book of sequences of opening moves (which will either be stored as strings, have their own subclass created, or as a map) and provide each move through a getter function to the Board. This getter would have to be given the previous sequence of moves in the form of a vector and determine the following move by inputting each sequence as a key to determine the next suggested move stored in the map. This emulates the “book” or “dictionary” nature of opening chess sequences, where given an existing sequence as an inputted key, a value (the next move) can be returned. This value would either come in the form of a new sequence, which is the key but extended by 1 move, or as a singular move, and the next key would need to append this move to again determine the next suggested move. Should no key exist, then no value would be returned, much like a chess “book.”

**Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**

Creating a vector of Boards or Board pointers would allow us to track each Board state in main after every turn. The undo function would be passed the vector as an argument, and traverse backward through the Board vector, modifying it and returning a previous Board state as the new current Board state. The existing current Board state would be discarded and destroyed once the program moves out of scope.

**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.**

The board would have to be modified to accommodate the additional squares, as well as to ensure that 2x2 squares of cells are removed in each corner. The rules would remain largely the same, with piece logic incorporating new "directions." The existing directions hinge on the game being played solely in "forward" and "backward" directions. The new program would have to completely rework that train of thought, or map it to a horizontal equivalent for "left" and "right." Four players would have to be supported by Board and Main. Victory conditions would also have to be modified to accommodate the "20-point win condition." Each piece would have to be given a point value. Points would have to be tallied according to each Board as a result of this win condition. Victory would also otherwise only be achieved when 3 of the 4 initial players are eliminated from the game.

## Extra Credit Features

We unfortunately did not finish the entire project in time to be able to account for extra credit features, but we hope to try to implement them as a group even past this deadline and our presentation.

# Final Questions

**1. 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?**

This project taught us a significant amount about developing software in teams, specifically collaboration in editors, maintaining code styles, and working towards a collective goal.

Our team worked almost exclusively in the Visual Studio Code editor, with testing occurring in Putty. As such, we needed to plan ahead in order to divide up the workload equally, and play to each of our strengths and weaknesses. This required a great deal of communication, several in-person programming sessions, and a base level of coding proficiency. Early on in the project, we set up a GitHub repository for version control and sharing code. This proved to be crucial in saving our progress, and returning to previous versions when grave errors did arise. We also found a LiveShare extension for VSCode which allowed us to simultaneously work on one teammates' editor from different devices. This was also critical in ensuring we didn't have many merge conflicts.

We also learned very quickly that maintaining a cohesive code style throughout was essential. This was specifically seen in the way we manage rows and columns with varying indexes, as well as general brackets, spacing, commenting, and naming conventions. These practices all served to keep our code organized and made it easier to help peer review when issues arose.

Finally, we learned that sharing a collective goal for the project helped us stay on track. We had frequent discussions and bounced ideas off each other to keep our code moving in the right direction.

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

If we had the chance to start over, we probably would have started earlier. Despite working on this project for tens of hours, we still felt as though more time could have benefitted our implementation, and given us the opportunity to implement all the levels and extra features we hoped to have. Some suggested features, such as castling, en passant, and pawn promotion would also have been quite easily implemented given more time spent on the project.

## Conclusion

In conclusion, a substantially functional chess game was a significant challenge to tackle, and was equal parts rewarding as we worked through the various design and implementation issues as a team. We wished to have more time to work on the project and complete every aspect in the Chess instructions, but are proud of what we accomplished in the given time frame, and how we applied the course concepts to our code.