CS246 A5 CHESS DESIGN DOCUMENT

By: Nicholas Rebello, Shahrukh Qureshi, and Shwetang Desai

Overview:

As our UML suggests, we've got a Board class that pertains to all the functions of the chess board and its interaction with our observers and subjects. There are 64 spaces on the board along with the text and the graphics display. In constructing the board, the program uses the command setupGame() to initialize the Board with pieces, set each player's respective colours, and set the starting turn. We have a game.setup() function that places sets different chess pieces in any stated place on the Board, as per the user's request. Both kings (black and white) must be declared first before creating other pieces. The board class has a vector of Piece pointers which stores all the pieces created during the setup process safely. It allows us to safely destroy certain pieces when they have been captured. Based on the nature of each piece, we have separated conditional Boolean values throughout each piece to specify the current state of conditions such as: check, checkmate, stalemate, castle, and en-passant.

We decided to break up pieces of chess functionality into their own classes and encapsulate all the logic in order to make the code more maintainable, testable, and more reusable.

In the input, note that running the command "game", the first argument is white, the second argument is black.

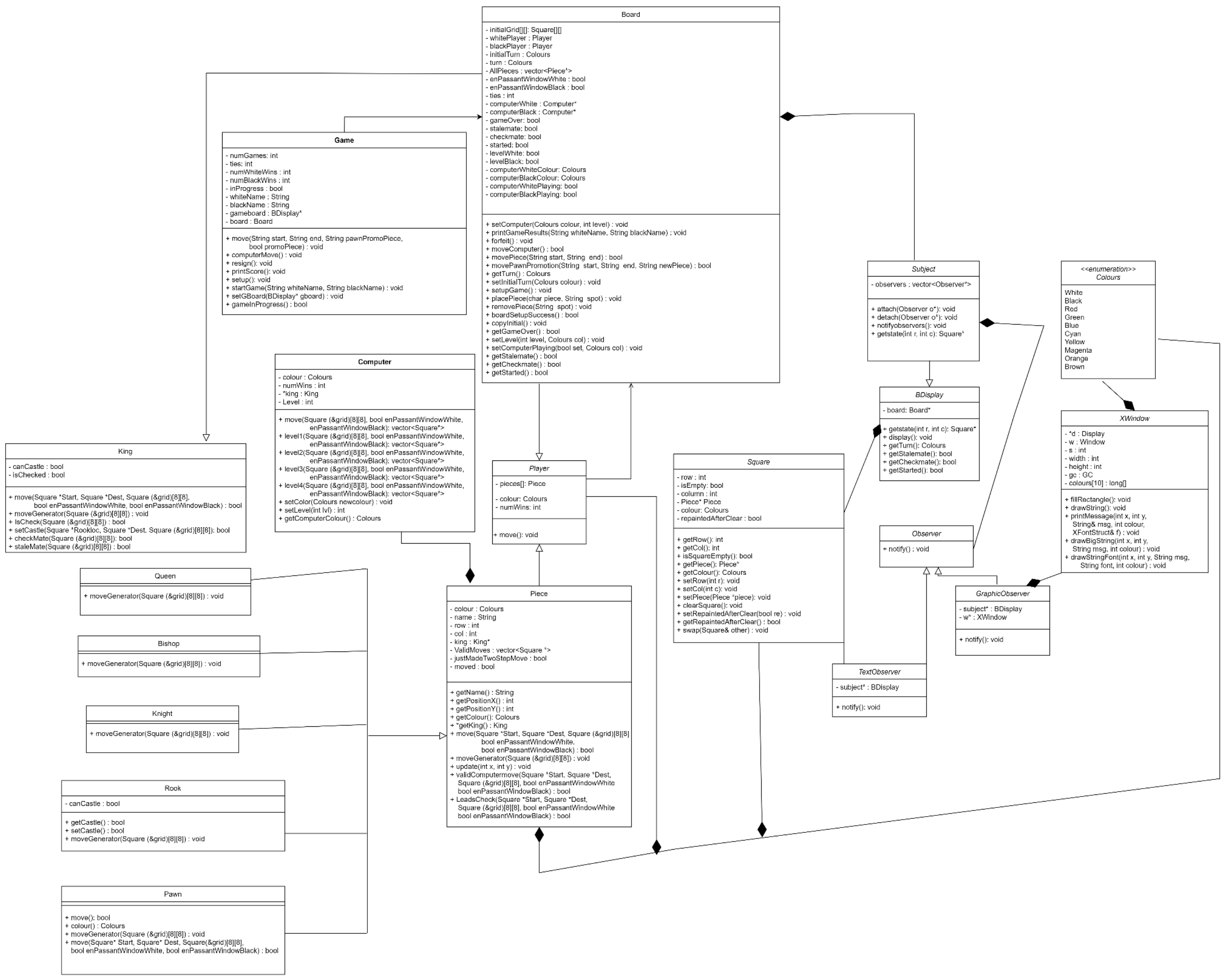
Some notable classes are: Piece, Square, Board.

1. The Piece class has 6 children: King, Queen, Pawn, Rook, Bishop, and Knight. Each piece has a constructor and a moveGenerator, which generates a list of all possible valid moves for a certain piece. Rook has a boolean, canCastle, with its corresponding getter and setter: getCastle(), setCastle(). This is used to determine whether the Rook is able to castle with the King. The King has an exclusive move() overridden function, isCheck(), and an exclusive field called isChecked, which is used to track whether the king is in check, and a function called checkmate(), which identifies if the king is in check and cannot move anymore.
   Pawn has member variables and functions to consider the initial two step moves, potential Pawn promotion, and potential en-passant opportunity.
2. The Square class contains important fields such as its row (row), column (col), a boolean to states if the square is empty or contains a piece on it (isEmpty), one of the pieces (piece), a defined colour (colour), a boolean that states if a square has been repainted after cleared with white or black (repaintedAfterClear), and a swap function that swaps the contents of two

squares with each other (swap). They have their respective getters and setters.

3. In the Board class, we defined a player struct with the fields: color, numWins, and a king pointer (king). We initialize the 8x8 board with the following square constructors: (row=0, col=0, isEmpty=true, piece=nullptr, colour=White). Since chess has two players, we have both whitePlayer and blackPlayer to clearly identify both players, and a Computer player, which uses mapping to identify important moves to make. White always goes first, so we set the initialTurn and default turn to White. We stored all the pieces in a vector of Piece* to easily create, manage, and delete pieces. The vector will be popped off the stack so we dont need to worry about deletion and leakages. We have a counter to count the number of ties while the program runs. To output the end of game results, we use the command printGameResults(whitename, blackname) to print how many times white and black has won. There is a function called forfeit() should the player resign during the play. Piece movement is actually handled and placed on the board here using moveComputer(), movePiece(start, end), movePawnPromotion(start, end, newPiece). During setup, we have placePiece(piece, spot) to place the piece on the board, and removePiece(spot) to remove the piece from the board. In order to tell if the board is setup properly, we have the function boardSetupSuccess(), which returns true or false if the board contains two kings, and no pawns on either the first/eight rows (if white pawns usually start on row 2, then it cannot be placed on row 1. Similarly, if pawns are starting on row 7, it cannot be placed on row 8. They can however be placed on other rows.

Updated UML


Board
- initialGrid[][]: Square[][]
- whitePlayer : Player
- blackPlayer : Player
- initialTurn : Colours
- turn : Colours
- AllPieces : vector<Piece*>
- enPassantWindowWhite : bool
- enPassantWindowBlack : bool
- ties : int
- computerWhite : Computer*
- computerBlack : Computer*
- gameOver: bool
- stalemate: bool
- checkmate: bool
- started: bool
- levelWhite: bool
- levelBlack: bool
- computerWhiteColour: Colours
- computerBlackColour: Colours
- computerWhitePlaying: bool
- computerBlackPlaying: bool
+ setComputer(Colours colour, int level) : void
+ printGameResults(String whiteName, String blackName) : void
+ forfeit() : void
+ moveComputer() : bool
+ movePiece(String start, String end) : bool
+ movePawnPromotion(String start, String end, String newPiece) : bool
+ getTurn() : Colours
+ setInitialTurn(Colours colour) : void
+ setupGame() : void
+ placePiece(char piece, String spot) : void
+ removePiece(String spot) : void
+ boardSetupSuccess() : bool
+ copyInitial() : void
+ getGameOver() : bool
+ setLevel(int level, Colours col) : void
+ setComputerPlaying(bool set, Colours col) : void
+ getStalemate() : bool
+ getCheckmate() : bool
+ getStarted() : bool
Game
- numGames: int
- ties: int
- numWhiteWins : int
- numBlackWins : int
- inProgress : bool
- whiteName : String
- blackName : String
- gameboard : BDisplay*
- board : Board
+ move(String start, String end, String pawnPromoPiece, bool promoPiece) : void
+ computerMove() : void
+ resign(): void
+ printScore(): void
+ setup(): void
+ startGame(String whiteName, String blackName) : void
+ setGBoard(BDisplay* gboard) : void
+ gameInProgress() : bool
Computer
- colour : Colours
- numWins : int
- *king : King
- Level : int
+ move(Square (&grid)[8][8], bool enPassantWindowWhite, enPassantWindowBlack): vector<Square*>
+ level1(Square (&grid)[8][8], bool enPassantWindowWhite, enPassantWindowBlack): vector<Square*>
+ level2(Square (&grid)[8][8], bool enPassantWindowWhite, enPassantWindowBlack): vector<Square*>
+ level3(Square (&grid)[8][8], bool enPassantWindowWhite, enPassantWindowBlack): vector<Square*>
+ level4(Square (&grid)[8][8], bool enPassantWindowWhite, enPassantWindowBlack): vector<Square*>
+ setColor(Colours newcolour) : void
+ setLevel(int lvl) : int
+ getComputerColour() : Colours
King
- canCastle : bool
- isChecked : bool
+ move(Square *Start, Square *Dest, Square (&grid)[8][8], bool enPassantWindowWhite, bool enPassantWindowBlack) : bool
+ moveGenerator(Square (&grid)[8][8]) : void
+ IsCheck(Square (&grid)[8][8]) : bool
+ setCastle(Square *Rookloc, Square *Dest, Square (&grid)[8][8]): bool
+ checkMate(Square (&grid)[8][8]): bool
+ staleMate(Square (&grid)[8][8]) : bool
Queen
+ moveGenerator(Square (&grid)[8][8]) : void
Bishop
+ moveGenerator(Square (&grid)[8][8]) : void
Knight
+ moveGenerator(Square (&grid)[8][8]) : void
Rook
- canCastle : bool
+ getCastle() : bool
+ setCastle() : bool
+ moveGenerator(Square (&grid)[8][8]) : void
Pawn
+ move(): bool
+ colour() : Colours
+ moveGenerator(Square (&grid)[8][8]) : void
+ move(Square* Start, Square* Dest, Square(&grid)[8][8], bool enPassantWindowWhite, bool enPassantWindowBlack) : bool
Player
- pieces[]: Piece
- colour: Colours
- numWins: int
+ move(): void
Piece
- colour : Colours
- name : String
- row : int
- col : int
- king : King*
- ValidMoves : vector<Square *>
- justMadeTwoStepMove : bool
- moved : bool
+ getName() : String
+ getPositionX() : int
+ getPositionY() : int
+ getColour(): Colours
+ *getKing() : King
+ move(Square *Start, Square *Dest, Square (&grid)[8][8] bool enPassantWindowWhite, bool enPassantWindowBlack) : bool
+ moveGenerator(Square (&grid)[8][8]) : void
+ update(int x, int y) : void
+ validComputermove(Square *Start, Square *Dest, Square (&grid)[8][8], bool enPassantWindowWhite bool enPassantWindowBlack) : bool
+ LeadsCheck(Square *Start, Square *Dest, Square (&grid)[8][8], bool enPassantWindowWhite bool enPassantWindowBlack) : bool
Square
- row : int
- isEmpty: bool
- column : int
- Piece* Piece
- colour: Colours
- repaintedAfterClear : bool
+ getRow(): int
+ getCol(): int
+ isSquareEmpty(): bool
+ getPiece(): Piece*
+ getColour(): Colours
+ setRow(int r): void
+ setCol(int c): void
+ setPiece(Piece *piece): void
+ clearSquare(): void
+ setRepaintedAfterClear(bool re) : void
+ getRepaintedAfterClear() : bool
+ swap(Square& other) : void
Subject
- observers : vector<Observer*>
+ attach(Observer o*): void
+ detach(Observer o*): void
+ notifyobservers(): void
+ getstate(int r, int c): Square*
BDisplay
- board: Board*
+ getstate(int r, int c): Square*
+ display(): void
+ getTurn(): Colours
+ getStalemate(): bool
+ getCheckmate(): bool
+ getStarted(): bool
Observer
+ notify() : void
TextObserver
- subject* : BDisplay
+ notify(): void
GraphicObserver
- subject* : BDisplay
- w* : XWindow
+ notify(): void
<<enumeration>>
Colours
White
Black
Red
Green
Blue
Cyan
Yellow
Magenta
Orange
Brown
XWindow
- *d : Display
- w : Window
- s : int
- width : int
- height : int
- gc : GC
- colours[10] : long[]
+ fillRectangle(): void
+ drawString(): void
+ printMessage(int x, int y, String& msg, int colour, XFontStruct& f) : void
+ drawBigString(int x, int y, String msg, int colour) : void
+ drawStringFont(int x, int y, String msg, String font, int colour) : void

Design

Our final UML design differs from the older UML (from Due Date 1). The basic ideas and concepts remain the same, but we have added extra functions to allow for easier interactions and management of piece and the board. The Observer pattern is used with the Board to take care of Square and piece movement interaction during a play.

We have added more member variables and methods compared to our original program design to take care of specific conditions such as en-passant, stalemate, checkmate, check, castling, etc.

After experiencing issues with locating some pieces and deleting them appropriately, we decided to contain them in a vector. We defined it as a vector of piece pointers (Piece*) to ensure that when a piece was captured, we could safely iterate through the vector until we found it, delete it, and move on without having to worry about segmentation faults. Eventually, the vector would be popped from the stack, letting us continue without having to deal with deleting the vector.

Instead of manually checking every move player makes to ensure its validity, we decided to create a vector<Square *> called ValiedMoves. This vector stores every valid square a given piece can move to (based on their nature). So, when a player made a move, if the square that they are moving to exists within that piece's vector of valid moves, then the player can make that move. Otherwise, they must try again as it is invalid.

Generally speaking, we decided to have most of our objects on the stack as opposed to storing on the heap to avoid having memory leaks. Objects on the stack would be popped off by itself, so it would lead to less leakages than otherwise.

We assigned the text and graphics display classes as observers. The text and graphics display classes observe each square of the board. When a square is modified, the text and graphics display update their data based on the new notification. During the development stage, we initially had a very slow text and graphics display. For example, we created a small "Four-move checkmate" which took about 5 mins to completely render. After carefully modifying and optimizing our displays, it now takes less than 30 seconds to completely render and run the test fully.

We noticed during the implementation stage the usefulness of the Observer Pattern:

1. It defines a "one-to-many" dependency between objects so that when one object changes its state, all subjects dependent on it are immediately notified and updated automatically - Low Coupling
2. Based on how we structured our implementation, it is very easy to simple attach and detach observers to each subject.

Resilience to Change

Our goal for this implementation was to make the classes and functions in such a way that we would not need to completely rethink our steps and game logic to accommodate for changing of features, additions of design, simplicity, and/or extra rules.

An example of such would be when the user inputs board coordinates. We only need to set up a mathematical conversion from their input to our team's way of representing row and columns. For example, if there is a pawn on A2 and the user inputs the following command:

move a2 a4

All we need to do is convert a2 into our representation of rows and columns: a: column 0, 2 = row 1. 4 = row 3. So this function would translate to: move the contents of the square at [1][0] to the square at [3][0].

We can easily add new features and functions other than checkmate, en-passant, etc. All we would need is to add/remove corresponding variables (mostly booleans) and methods for the involved pieces and squares, write their respective getters and setters, and perhaps have an overridden move just to deal with that feature's condition.

Even regarding the computer's AI engine, most of the upper levels depend on the easier levels. For example, level 3 may use a level 1 computer for certain cases, level 4 may use both level 2 and level 3 for cases, etc.

Questions from DD1

The questions below are from Due Date 1. They are unchanged:

1. 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.

   To implement a book of standard openings, my group would need to create an association list, a dictionary, or a vector of tuples, which would contain a particular state of the board and moves mapped together. Each element of this vector would be a tuple of two elements.

   The first element would be a particular opening state of the board. The second element would be the next move that the computer must take (it would look like a condition – given this state of the board (element 1), the computer must execute the following move (element 2)).

The computer would “read” from this book of standard openings and be able to determine how to proceed.

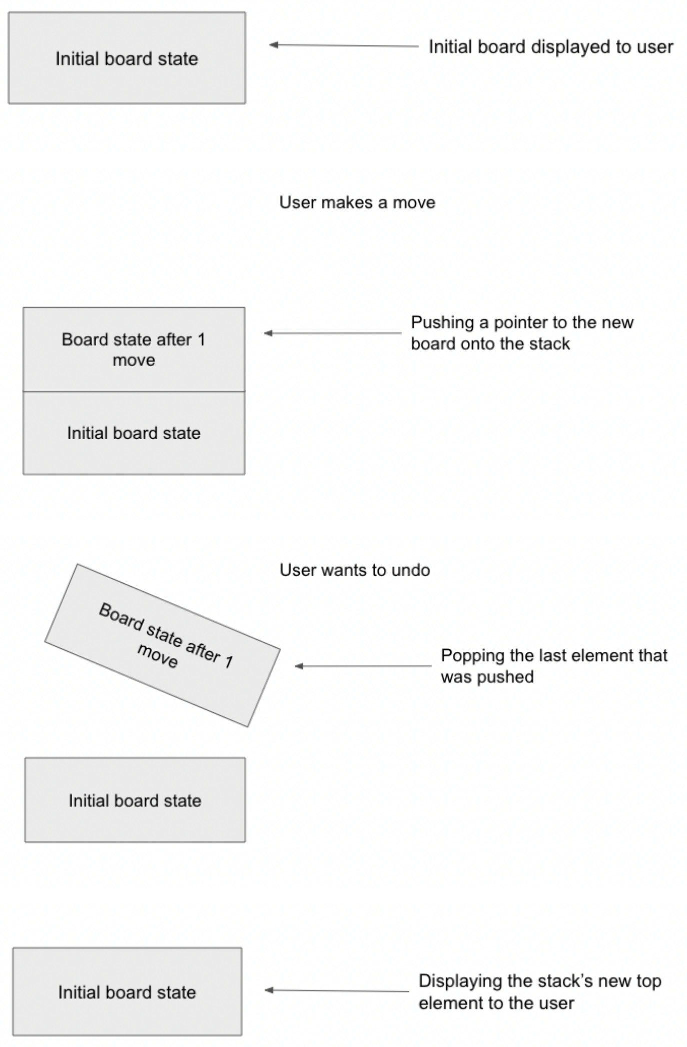
This may only be effective for the first 10-15 moves, as it would be difficult to curate millions of different board states and the next move for that state. A larger vector would only slow down the efficiency of the computer, making the program run slower as the computer searched through a large array of tuples for a suitable solution. Even then, it may not find a solution for longer games.

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

   An undo feature can be implemented by keeping track of the previous iterations of the game. Upon the undo feature being invoked, it would simply render the previous iteration of the current game. An undo feature would go back a single move so a new iteration of the game would be created after each move. Each move will alter the state of the chess board so we would have to store a pointer to each chess board each time a move is made. We can use a stack to keep track of each iteration. We would instantiate a stack of board pointers and after each move, we push a pointer to the board onto the stack. We would continue this until a winner is determined or until the game is stalemated. If the user invokes the undo feature, we simply return the board pointer at the top of the stack then pop it off the stack. This would essentially undo the move by returning to the previous iteration of the game. This implementation would also allow the user to undo every move until they return to the initial iteration of the chess board which represents the beginning of the current game.

   Additionally, if the user wanted to undo past their current game, assuming the user played a previous game, we could keep track of each game using a stack of game pointers. Once the stack of moves for the current game is empty, we would pop the current game off the stack and display the previous game and now each undo will invoke the previous game’s stack of moves.

Diagrams below to illustrate how the undo feature would work:


Initial board state
Initial board displayed to user
User makes a move
Board state after 1 move
Initial board state
Pushing a pointer to the new board onto the stack
User wants to undo
Board state after 1 move
Popping the last element that was pushed
Initial board state
Initial board state
Displaying the stack's new top element to the user

3. 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.

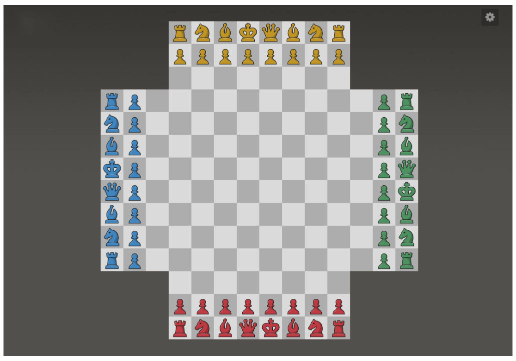
   Four handed chess is a variant where you have four teams instead of two, with each team a different colour, positioned on one one of the four sides on a board. The first change we would have to make is to change the board size and display to accommodate for the increased number of teams. We would use a standard 8x8 board like in our standard implementation, but extend 3 rows of 8 cells on 2 of the sides to place the starting set of each team, as per chess regulation.

   To do this, we would simply need to modify the square array in our board class to accommodate for the size changes, and create more square objects as required to ensure our display is working properly. Next, we would need to create a total of 4 player classes instead of two, with a unique colour for each player class, then like the standard implementation, each player would store their respective set of pieces within the object using an array.

The game engine would work similarly, however, the turned based logic would now have to be slightly modified to account for 4 turns instead of two. This can be done in our main quite easily. As for the gameplay itself, it will be free-for-all 4 player chess, with the opportunity to have as many computers or human players as possible. Simply put, we will create “computer objects” for players that are computers and just player objects for players that are humans. Each computer player can be adjusted to be a level from 1-4, giving lots of flexibility in how hard or how easy the gameplay will be.

The display would be handled using the subject/observer pattern, we will use an approach similar to A4 to display our board, since we have a class square, we can use that class and arrays of squares to generate the information required to create our board (colour, size, piece information). We will attach a text and graphical observer to our subject instance, allowing us to display either text or graphic, or both. Lastly, the game winning condition would also need to be changed, since we now have 4 players, the game will simply continue until the “last man” standing. That is, a checkmate to a player would not cause the game logic to end if the amount of players still playing other than the one checkmated is either 2 or 3.

Image of how 4 player chess would look:



(*4 player chess - chess terms*. Chess.com. (n.d.). Retrieved November 24, 2022, from https://www.chess.com/terms/4-player-chess)

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?

Working on a team coding project, such as implementing a chess game in c++, can teach several important lessons about developing software in teams.
First, it's important to establish clear communication and collaboration among team members. Our team set up regular meetings during the initial phases to discuss progress, assign specific tasks to each team member, and make sure everyone is on the same page when it comes to the project's goals and requirements. Later on, we scheduled calls to help debug code.
Second, it's crucial to establish a version control system to manage the codebase. We set up a GitHub group repository to enable us to work on the code simultaneously without running into conflicts or overwriting each other's work.
Third, it's important to take a structured, organized approach to the development process. This might involve using a project management tool to track progress, setting deadlines for

each task, and regularly reviewing and testing the code to ensure it's working as intended. During due date 1, we created a plan where we set deadlines for each task, and regularly reviewed and tested code to ensure that we had no memory leaks or bugs. For example, once we have set up all the pieces, we tested piece placement. Then, when we implemented the piece move methods, we tested each piece's movements around the board. We later developed tests for special conditions, capturing, checkmating, stalemating, and eventually, full-scale games.
Overall, working on a team coding project has been a valuable learning experience that helps us develop important skills such as communication, collaboration, and project management.

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

If we had the chance to start over, there are several things we might do differently.
First, we might consider setting up a more structured development process, with clear goals and deadlines for each task. This could help ensure that the project stays on track and that everyone knows what they need to be working on at any given time.
Second, we might consider using a more robust version control system to manage the codebase. This could help prevent conflicts and make it easier for us to work on the code simultaneously.
Third, we might take a more collaborative approach to the development process, by setting up regular meetings to discuss progress and allowing for more input and feedback from team members.
Overall, by taking a more structured and organized approach to the project, you can help ensure that the development process is smooth and efficient and that the final result meets the project's goals and requirements.

Conclusion:
In conclusion, this design document outlines the key considerations, decisions, and thought process that were made during the planning and development phase of this project. It provides a clear and detailed description of how we approached this project, how the project was implemented, and depicts how we solved problems that occurred during the implementation phase. By following the guidelines and recommendations outlined in the assignment description and guidelines, the team ensured that the project was developed in a consistent and efficient manner, and that the final result met the project's goals and requirements.