Design Changes from Initial Plan

- The overall design has changed significantly from what we have initially planned.
    - An InputValidation class was added to validate wrong inputs received from users since there were multiple commands in the game.
    - Board class has changed dramatically by adding methods to check special conditions such as en passant, castling, and pawn promotion.
    - The abstract Player class included more methods in order to access private fields because the Controller required the coordinates generated from random generation in Computer class.
- Our initial plan of the interaction between Board and View was that they should not have any interaction (according to MVC example from lectures). However, we modified the standard MVC (model-view-controller) pattern a bit by View class receiving a pointer to board object in Controller.
- A current player object was added in Controller in order to track down who the current player is. This was significant in moving the pieces of right player and adding score for the player.

Project Structure

This program depends on the Model-View-Controller design pattern, where the Board class represents Model and each of Text Display and Graphic Display inherits from View.

- Controller's roles include manipulating the overall flow of the program as well as interacting with the user to take inputs and print outputs.
- Controller "has" both Model (Board) and View classes, as well as the Player classes.
- We have decided that it would be more straightforward to allow View and Model to directly interact instead of making them communicate information via Controller.
- Therefore, the three parts of our program, Model, View, and Controller can communicate with one another.
- Every time a change is made, Controller notifies View by passing the pointer of Board. Then, View communicates with Board via function calls to print appropriate outputs to the interface.
- Board "has" a 2D vector of Piece classes, from which each of King, Queen, Rook, Bishop, Knight and Pawn classes inherit. We implemented this using Shared Pointers to minimize opportunities for leaks.

Controller

- As mentioned previously, Controller is responsible for overall flow of the program as well as interacting with the user to take inputs and print outputs. Also, initializing both Board and Player objects.
- The methods, *play, game, setup*, are used for receiving inputs from users and direct users to the part of the program they desired. For example, if a user wants to setup a

custom board, they would prompt setup and will be directed to *setup* method to initialize their own board.
- The method *init* initializes players by white or black colour from user input and the sets up board with default configuration. Also, before the method ends, it notifies the View with new board.
- *Notify* method notifies board to move the Piece using the coordinates input by users and checks if any special condition satisfies, such as castling. Again, before the method ends, it notifies View with new board.
- All other methods, *printScore, addScore, rebuild,* prints scores, accumulates players' scores, and resets the board with default setup if the user decides to play again.

Board

- Board is responsible for keeping track of the current state of the game as well as enforcing chess rules such as Castling, En Passant and Pawn Promotion
- At the beginning of the game, either one of the methods, *init* or *setup* is called by Controller to initialize the board accordingly.
- The current state of the board is managed using a field named *currStates* which is a 8x8 vector of shared pointers of Piece classes.
- The methods *getLetter, isEmpty* and *checkState* have been implemented to communicate the current state of the board to other classes.
- The method *canMove* determines whether a piece can move to a certain position according to chess rules.
- Other methods such as *isPromo*, *isbadPawnPromotion, castling, offEnPasant* were implemented to enforce advanced rules of chess.

Piece

- Piece classes are constructed by Board which communicates with Controller to ensure that they are set up appropriately.
- Each of **King, Queen, Bishop, Rook, Knight** and **Pawn** inherits from Piece.
- When a move is called, the *move* method is called followed by *notifyBoard* to change the coordinate of the piece and communicate the update to Board.
- Its information about coordinates, colour and letter are communicated to other classes using methods such as *getRow, getCol, getLetter* and *getColour.*
- **Pawn** has methods *canEnPassent* and *setEnPassent* which are called by Board to follow En Passant rules.
- A captured piece is permanently removed from the board. (Pieces are constructed using shared pointers, so they will eventually be deleted.)

Player

- Player classes are constructed by Controller from which **Human** and **Computer** classes are inherited.
- Its information about current score, colour and name(whether it's human or computer) are communicated to other classes using methods such as *getScore,*

getColour and *getName.* It also has *addScore* method which is called by Controller when the player wins a game.
- The methods *isStalemate, isCheck* and *isCheckmate* were implemented to communicate the state of the player in each chess game.
- The **Computer** class has private methods *randomMove, avoidCapture* and *capturingMove* which are called by the public method *nextMove* when the AI level is 1, 2 or 3, respectively.
    - *randomMove* determines the Computer's next move randomly; it scans through possible moves from the current state and randomly decides a move.
    - *avoidCapture* checks which pieces are under attack by the other player, and moves away the piece. When there are more than one pieces under attack, this AI level was given a preference to avoid captures of pieces in the order of Queen, Bishop, Rook, Knight and Pawn. When there are currently no pieces under attack, it randomly chooses a move.
    - *capturingMove* checks which pieces can be captured by the Computer. It was given a preference to capture pieces in the order of Queen, Bishop, Rook, Knight and Pawn. When there are no possible captures, it will avoid being captured by the other player.

View

- Controller takes inputs from the user to choose between Text and Graphic Display at the initiation of the game, and constructs a View class accordingly.
- The private field *theDisplay* is a 2-D vector of character which reflects the current state of the display and is updated every time a change is made to Board.
- It has *notify* method which is called by Controller to update *theDisplay.*
- In the display, capital letters denote white pieces and lowercase letters denote black pieces.
- In Text Display, unoccupied squares are denoted by a blank space for white squares, and an underscore character for dark squares.

Design Quality

- **Cohesion**: In order to maintain high cohesion, we keep elements in a module to perform one task by either going through helper function or directly. Elements in module are placed by their commonality, for example the Board class has vector of vector of Piece to perform as an Observer while other modules are prohibited to have this element. If an element requires data from other elements, it is passed on so that it is strictly manipulating with data the element has received.
- **Coupling**: We ensured that our program has low coupling by keeping the dependencies among program modules as low as possible. The modules communicate with each other via function calls with basic parameters and results. They were also implemented to affect each other's control flow and share global data. Thus, each of our modules is reusable in other programs, and a change to one module does not affect others.

Post-Project Questions

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?

- The software project has taught us the importance of having extra eyes for debugging. For example, during our development, one of our member was stuck on slight bug for an hour or so. But, when other two group members joined in to fix the bug, we were able to catch within few minutes.
- Another lesson we learned was that frequent communication between members are significant. We met everyday together and coded for few hours because we believed that despite each member was assigned to different parts of the development, when problem arises that are related to others' work, we were able to ask and notify the issue on spot.
- Version control was another important lesson we learned. When some of us were working on same files, a merge error occurred couple of times, and this required some work-together in order to avoid any deletion of code that other group members has pushed. Ultimately, this increased our skills of controlling by being able to reverse commits, fix merge errors, and much more.
- Finally, we learned about the benefits of correct distribution of work between the group members. If the work was not distributed accordingly to time and amount, we would not have been able to complete the project on time. However, as each member was assigned to parts which they were more comfortable with, we were able to have a good progress and almost able to follow our initial schedule.

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

- We would definitely create more subclasses for Board. Currently, the Board class is too overloaded and we believe that Board has too much "work load". Thus, we would create subclass such as "PotentialMove" class to divide the work of Board has. Or, even place the move methods of each Piece into the Piece subclasses: King, Queen, Bishop, Pawn, etc.
- We definitely would have allocated a bit more time on implementation of A.I level 4. We would have re-scheduled our planning so that we could have more time to brainstorm for level 4. Also, since we already have level 2 and 3 completed, a little more brainstorming for the level 4 algorithm could have resulted in completion of A.I level 4.