Rastin Rassoli Cellina Patricia

CS 246 - Project
“Plan of Attack”

1) Breakdown

First things first, following the UML we will start with the view part. By doing so, we are able to test our program incrementally, which makes debugging easier. Starting from the ChessView class, we will then continue implementing the Observer class along with the TextObserver. The GraphicsObserver will be implemented at the very end after completing each function and when the TextObserver is working perfectly. By doing so, it is expected to have the GraphicsObserver implemented similarly to TextObserver; thus, if the TextObserver is working perfectly, then so does the GraphicsObserver.

After finishing the view module, we will move on to the ChessController. In order to run the game, the Chess Controller must be set up as the implementation of initializing the board is in this class. The controller has a composition of the board class; therefore, it is important to implement this class before making the model module.

Moving forward, we will start with the subject class from the Model module. The subject class is done first because it is connected with the observer classes, which is useful for testing and displaying the chess. Afterwards, we will divide ourselves into 2 teams; one will implement the main functions of board and some of the pieces, while the other implements move, position, and the rest of the piece classes. This must be done instantaneously since the Board class has a composition of move, position, and piece classes, so it is impossible to complete the board class without implementing the others. After successfully implementing the move and setup functions of the Board and Piece classes, we will test our program with both players as human. From here, we make sure that the basic functions and the displays are working fine before implementing the other main Board functions and the computer player classes.

Lastly, after everything, we will complete the model module with the ComputerPlayer class along with its subclasses. We want to implement the computer player after making sure that everything works fine as level 2 and level 3 requires the program to work perfectly. Without making sure that the rest are fine, it would be difficult to debug. The summary of the things to do, completion dates, and what each person is responsible for is shown below:

View
December 1st - December 2nd:

- ChessView (Rustin)
- Observer and TextObserver (Cellina)

Controller
December 2nd - December 3rd:

- ChessController: start, resign, move (Cellina)
- ChessController: setup_add, setup_remove, setup_color, setup_done (Rustin)

Model
December 4th - December 8th:

- Board, Knight, Queen, and Bishop (Rustin)
- Subject, Move, Position, Piece, King, Rook, Pawn (Cellina)

December 9th - December 12th:

- ComputerPlayer, Level1, Level2 (Cellina)
- Level3, GraphicsObserver (Rustin)

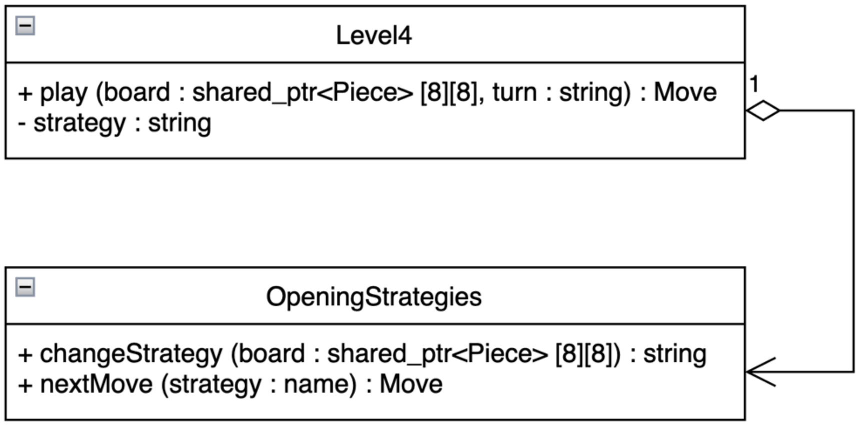
December 13th - December 15th:

- Testing, and implementing the bonus marks

2) Specified Questions

- Chess programs usually come with a book of standard opening move sequences. How would you implement a book of standard openings if required?
  In our current implementation, we have an object called *Move*. This object includes two *Position*s, piece name, removed piece (in case of an attack), and color. Position is a simple class which contains x and y (two Integers). By using Move object, opening strategies are simply an array of *Move*s. With the help of another class, which we call *OpeningStrategies*, we can implement this

requirement. Moreover, we should add a field called strategy to our Level4 *ComputerPlayer*. *OpeningStrategies* has two main functions *changeStrategy* and *nextMove*. *changeStrategy* inputs the current state of the board. Different opening strategies should be either stored in a file or in a map from name of the opening to the list of *Move*s. By comparing the current state with different strategies, it outputs the name of a strategy. The strategy feature of Level4 *ComputerPlayer* should be set to this name. After each move either the move matches the expected move of the strategy or it does not. If it matches, we continue with the current strategy by calling the *nextMove* function of *OpeningStrategies* and passing the name of the current strategy. If it does not match, we should change the current strategy by again calling *changeStrategy* and setting the new strategy. Note that in the case none of the strategies match the move we should follow the older strategy and we will not change it.

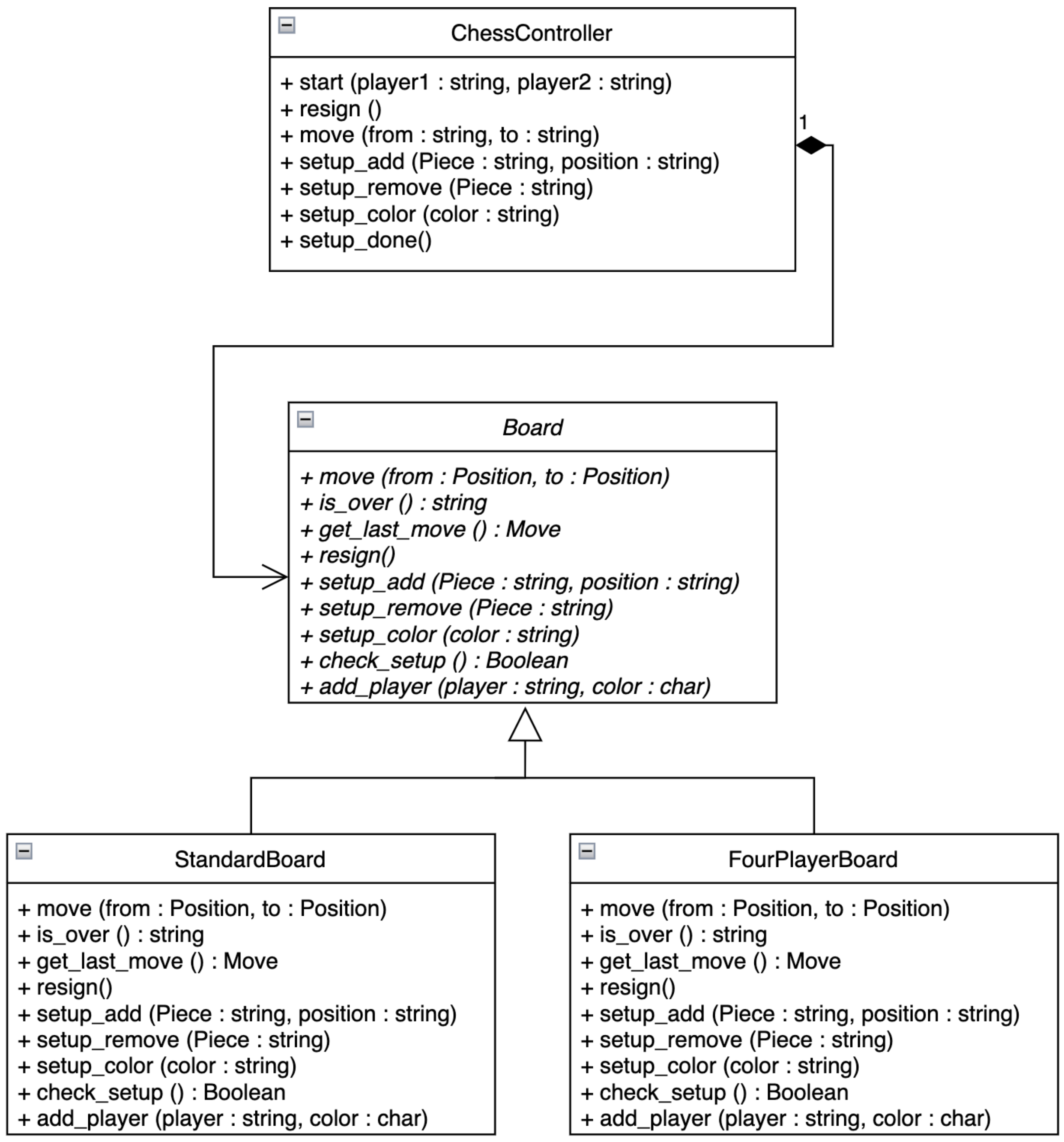

Level4
+ play (board : shared_ptr<Piece> [8][8], turn : string) : Move
- strategy : string
1
OpeningStrategies
+ changeStrategy (board : shared_ptr<Piece> [8][8]) : string
+ nextMove (strategy : name) : Move

- How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
  This is one of the bonuses we have chosen to implement; therefore, with our current UML we are able to add this functionality by adding just a few more functions. As described in the previous question, we have an object called *Move*. This object stores, from *Position*, to *Position*, and the removed *Piece*. Note that all of these elements are necessary for the implementation of undo. Now, by adding a simple feature called history (Vector<*Move*>) to the *Board*, which for

any call of move in our game adds the *Move* to the history, we can implement an unlimited number of undos. When the undo function of *Board* is called the last element in the history vector should be popped and the board updates itself with the opposite direction (to to from) and in case of an attack it should bring back the removed *Piece*.
Note: When the game is over due to the last move, the player does not have a chance for an undo in this implementation.

- Variations on chess abound. For example, four-handed chess is a variant that is played by four players. Outline the changes that would be necessary to make your program into a four-handed chess game.
  Generally, when we have different game modes with very similar characteristics, the strategy design pattern is very useful. The board in the chess game has very clear characteristics and in four player chess or other variations certain rules always hold. We follow the four-player chess as an example to demonstrate how we could implement different variations. The first small change occurs in the *Piece* objects. Now, the Move object has two more colors and it is not just black and white anymore, so Piece objects should accept these two colors as well. Then, we create a new abstract class called *Board* and our current *Board* becomes a subclass of *Board*, and we change its name to *StandardBoard*. We create another subclass for *Board*, called *FourPlayerBoard*. All of the public functions of our previous board become a pure virtual function in our current abstract class *Board*. Both the *StandardBoard* and the *FourPlayerBoard* should implement these functions based on their own rules. Moreover, we put a protected field called players, which is a map from the color of the player to the type of the player (human vs computer), in the abstract class. We still need a minor change in the *ChessControler* as well. The start function now should also accept the type of the game. Based on the players and the type of the game, the appropriate board should be created and the players should be added using the add_player function. With this strategy design pattern, we can implement other variations as well unless there is a fundamental change in one of the main rules. The following UML illustrates the changes:

Rastin Rassoli   Cellina Patricia


ChessController
+ start (player1 : string, player2 : string)
+ resign ()
+ move (from : string, to : string)
+ setup_add (Piece : string, position : string)
+ setup_remove (Piece : string)
+ setup_color (color : string)
+ setup_done()
1
Board
+ move (from : Position, to : Position)
+ is_over () : string
+ get_last_move () : Move
+ resign()
+ setup_add (Piece : string, position : string)
+ setup_remove (Piece : string)
+ setup_color (color : string)
+ check_setup () : Boolean
+ add_player (player : string, color : char)
StandardBoard
+ move (from : Position, to : Position)
+ is_over () : string
+ get_last_move () : Move
+ resign()
+ setup_add (Piece : string, position : string)
+ setup_remove (Piece : string)
+ setup_color (color : string)
+ check_setup () : Boolean
+ add_player (player : string, color : char)
FourPlayerBoard
+ move (from : Position, to : Position)
+ is_over () : string
+ get_last_move () : Move
+ resign()
+ setup_add (Piece : string, position : string)
+ setup_remove (Piece : string)
+ setup_color (color : string)
+ check_setup () : Boolean
+ add_player (player : string, color : char)