Question 1. A Checkers Game – Getters and Setters: OUT!

Solution:

The goal for this question is to remove all getters, setters, and public variables since using these is not a good practice in object oriented programming.

In our case, there were no getters and setters. However, there were 6 public variables mentioned as following:

Class name	Public variable	Description
Game	player	This variable is updated in Game class depending on whose turn it is to play. Since this variable was public and therefore was used by Board class, Moves class, and Validation class to know which player is playing.
Game	game_state	This variable changes from “Started” to “Ongoing” to “GameOver” depending on the state of the game. Even if this variable was public, it was not used by any other class than the Game class itself.
Moves	possible_diagonal_positions	This variable is an arraylist of a Point object. It stores all the positions of possible diagonal positions of a piece depending upon if it’s a king or pawn. Even if this variable was public, it was not used by any other class than the Moves class itself.
Moves	knock_out_positions	This variable is an arraylist of a Point object. In case there is any knock out in a particular turn, it stores all the positions of knocked out pieces so that Board class can use this information and remove it from the *board* while updating the *board*.
Board	board	This variable is a 2d integer array. It stores encoded values of pieces and checkers. This is the most overused and overshared variable among the classes (Game, Validation, Moves).
utils	scanner	This variable is used to take input from the terminal. Even if this variable was public, it was not

		used by any other class than the utils class itself.

On careful observation of the table above, one can realise that there are three out of six variables which are public but are not used outside the class. These variables are *scanner*, *possible_diagonal_positions,* and *game_state.* So these can be straightaway changed to private without any code changes.

This leaves us with three variables namely *board*, *knock_out_positions*, and *player* which we will look at one by one.

1. *board*

   Changing this variable to private required more refactoring as compared to other public variables conversion to private. The reason is that all the important classes need *board* in order to perform their task. This has been fixed by doing two things:

   1. If there is anything which can be done in Board class itself, then that part is moved to Board class. For eg: *check_input_validity* method is shifted in Board class. A new method *count_pieces* is also added in Board class addressing the same issue.

   2. If for some reason, the first fix is not possible then the *board* variable is given as input of method.

2. *knock_out_positions*

   Changing this variable to private was a little bit tricky as compared to others. This variable is an arraylist of Point object from java. As Professor suggested in lecture (also confirmed with TA) that if a variable is a list of objects then getters can be used for that variable. So we changed this variable from public to private. But we introduced a getter method named *getKnockOutPosition* in Moves class. This getter method is used in Board class in *update_board* method.

3. *player*

   Whenever there is information of the player (“Red” or “White”) required, it has been passed through methods along with other inputs. For eg.

   Before:

   ```
   Boolean validity = validate.is_valid_input(new_move);
   ```

   After:

   ```
   Boolean validity = Board.check_input_validity(new_move, player);
   ```

   While implementing this, we realised that the *player* variable can actually be just a variable in *play* method and doesn’t have to be a class variable since it is not used in any other method in Game class. So we shifted this variable inside the method *play.*

In addition to changing public variables to private, we also changed some methods from public to private since they are just used in the same classes as they are and making them public doesn't make any sense.

Conclusion:

For this question, we were supposed to remove 9 cases of getters, setters, and public variables. As mentioned above, we didn't have any getters and setters. We had 6 public variables and all of them are now converted to private variables. However, in order to do that we need to introduce a getter method named *getKnockOutPosition.* But permission to do so has already been taken from our TA.

Question 2. A Checkers Game – 20%-Time

Solution:

Improvements:

- Player turn is now decided by a state rather than odd or even integer
- Made the code object oriented. While doing so we directly implemented the observer pattern (see question 3)
- Added to the Gameover algorithm the scenario where current player can't make any more moves
- Use regular expressions to check the input format instead of if statements
- New feature "hint": The user can type "Hint@*position*" to get all possible moves from desired position. For eg.: type "Hint@a3". In order for this to work we refactored the moves class and implemented the rules for multiple jump moves.

The UML Diagram can be found in question 3 since we did those questions simultaneously.

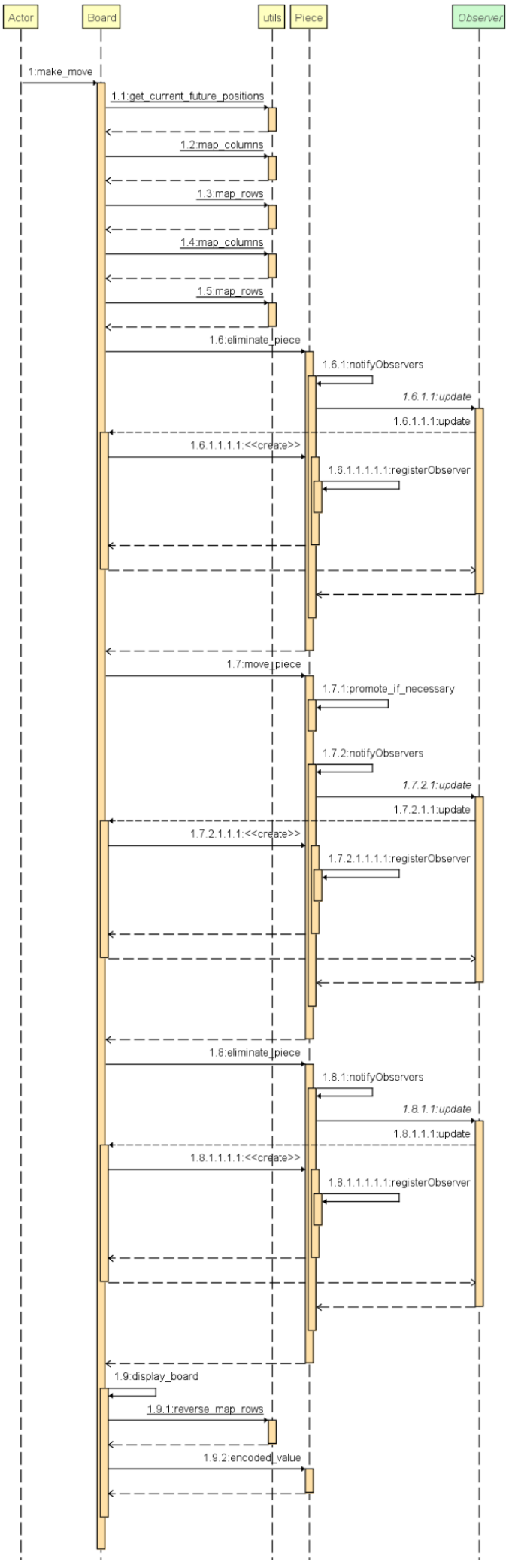
Question 3 - A Checkers Game – A Design Pattern IN!

1. We decided to implement the Observer pattern because it made the most sense. Singleton Pattern would have been our second choice, but we believe that Observer pattern is the most useful for our code and also the Pattern where we would learn the most while implementing it.
   As we were implementing the Piece class for Question 2, we noticed a problem: We had an array of Pieces in Board class, but each individual Piece had a pos-attribute too where it's coordinates were stored. This essentially led to the decision to implement the Observer pattern: Whenever a Piece's pos-attribute was updated/changed, the Piece could call notifyObservers() which would in turn make the Board update this pieces position in it's array, so that the Pieces pos-attribute and it's position in the Board-array would match.
   To implement this, we created 2 new Interface-classes called Observer and Subject. Board implements Observer and Piece-class implements Subject.

   Board has to implement the update()-function, where it sets the array-entry at new_pos equal to the entry at old_pos - which basically moves the piece in the board-array - and then sets the entry at old_pos to a new empty Piece.

   Piece has to implement 3 functions. The first is registerObserver, which is called every time a new Piece is created. This function adds Board to the List of Observers of the Piece so that Board can be called when the Pieces position is changed.
   The second function is removeObserver, which removes a given Observer from the observers-list. This is not really used in our code, but it still needed to be implemented.
   The last function is notifyObservers, which calls the update function on each Observer in the list. This function needs to be called every time a Piece's pos-attribute is changed. It makes sure that a change in a Piece's pos-attribute is also reflected in the Board-array. notifyObservers is called in Piece.move_piece to update it's position in Board-array, but it's also called in eliminate_piece to update the boards attribute CountWhitePieces and CountRedPieces, which keep track of the number of Pieces of each color on the board.

2. Sequence Diagram


Actor
Board
utils
Piece
Observer
1:make_move
1.1:get_current_future_positions
1.2:map_columns
1.3:map_rows
1.4:map_columns
1.5:map_rows
1.6:eliminate_piece
1.6.1:notifyObservers
1.6.1.1:update
1.6.1.1.1:update
1.6.1.1.1.1:<<create>>
1.6.1.1.1.1.1:registerObserver
1.7:move_piece
1.7.1:promote_if_necessary
1.7.2:notifyObservers
1.7.2.1:update
1.7.2.1.1:update
1.7.2.1.1.1:<<create>>
1.7.2.1.1.1.1:registerObserver
1.8:eliminate_piece
1.8.1:notifyObservers
1.8.1.1:update
1.8.1.1.1:update
1.8.1.1.1.1:<<create>>
1.8.1.1.1.1.1:registerObserver
1.9:display_board
1.9.1:reverse_map_rows
1.9.2:encoded_value

3. Class diagram