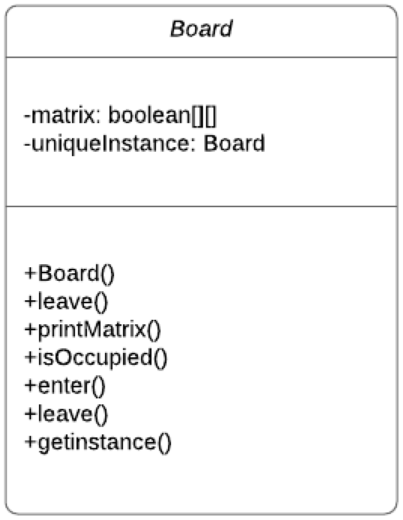
Assignment 3

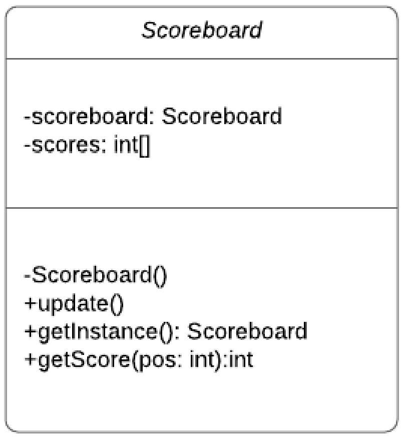
Part 1

Singleton

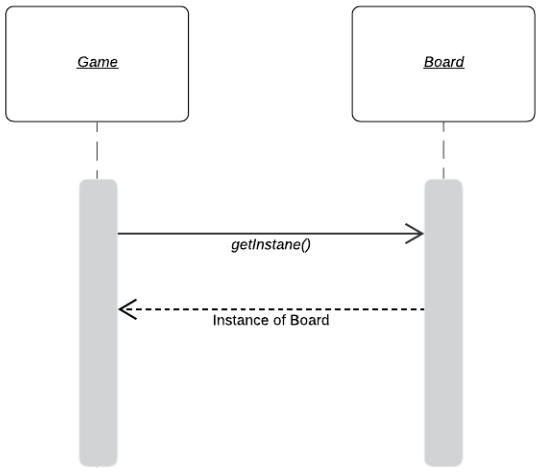
1)
We implemented the Singleton-Pattern in the "Board" class and the "Scoreboard" class (which we implemented in Part 3), because it should have only one instance of each of them during the running program. Furthermore, the implementation of this pattern provides a global point of access the instance of both classes. Therefore, we don't have to give the instance of the board to the pieces. Therefore, we don't have to give the instance of the board to the pieces. The implementation can be seen in the class diagram. In general, it is how we learnt in the lecture, the constructor of the board is private and we created a public function(getinstance()) to create an instance if no exists yet or if one exists, it gets returned.

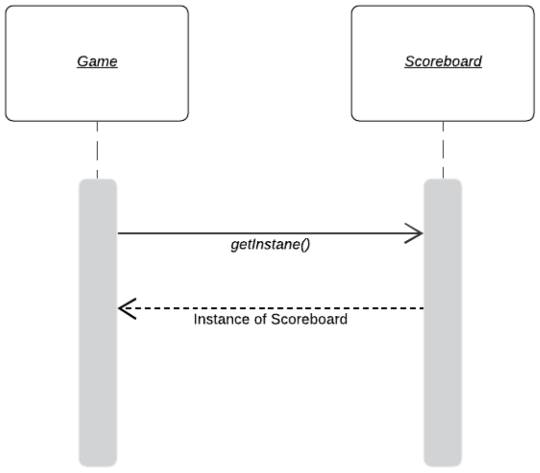
2) Class diagrams:





3) Sequence diagrams:



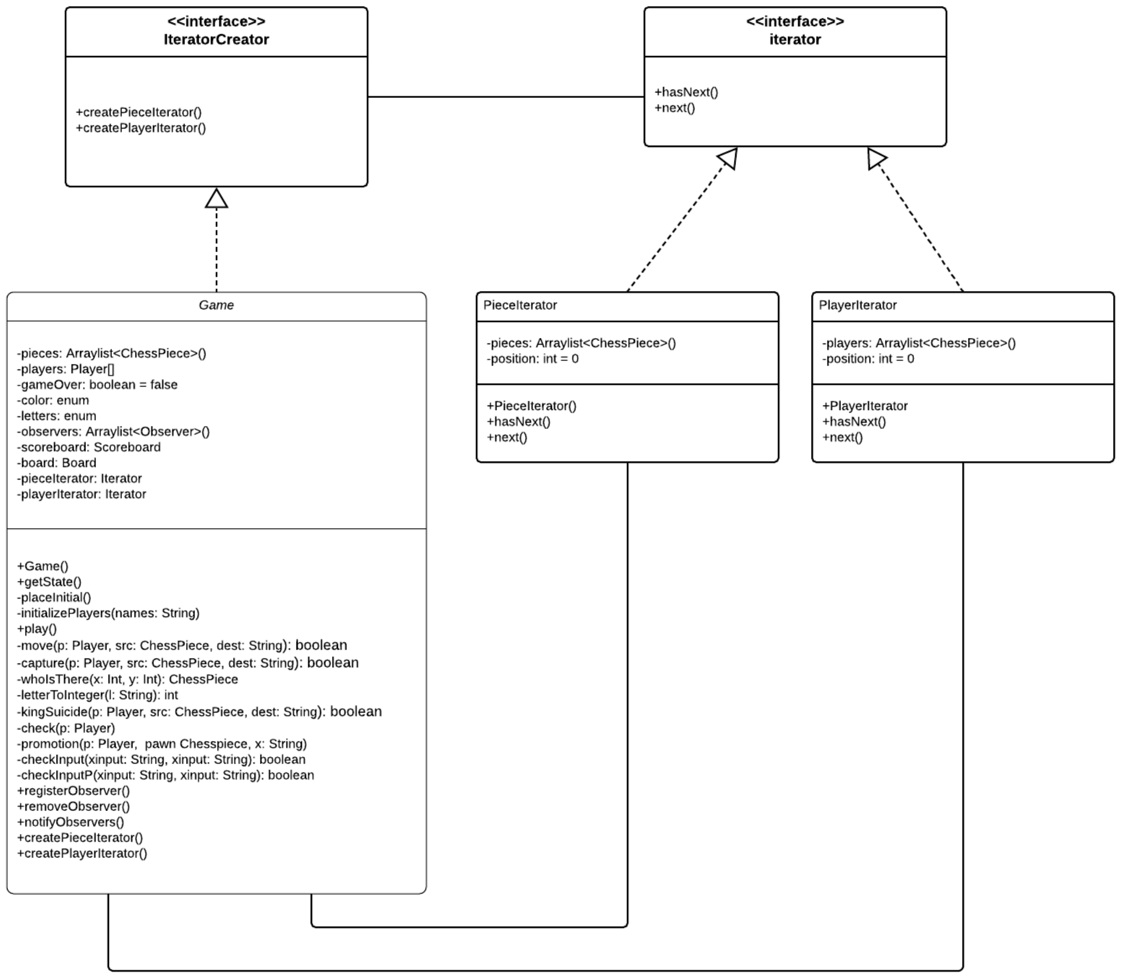


Iterator

1)

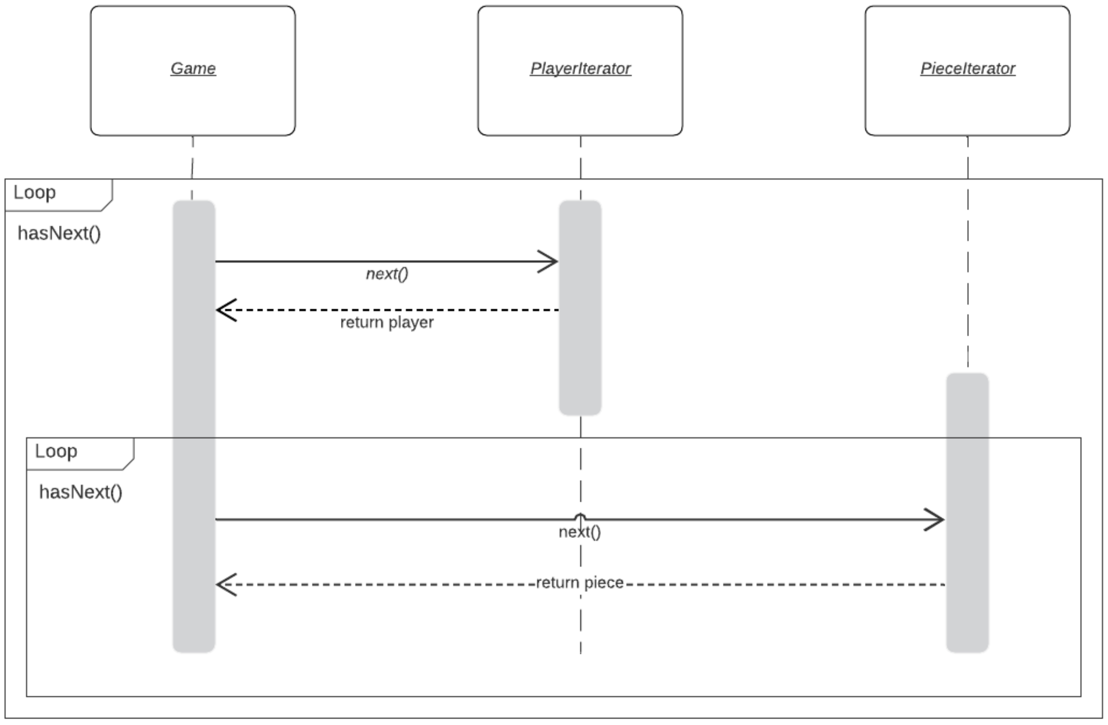
We implemented the Iterator-Pattern in the “Game” class to iterate over the chess pieces and the players. Both are store in a list with a different underlying presentation. The chess pieces are stored in an arraylist and the players are stored in an array. The implementation of this pattern allows us consistent access to the elements of both lists. Also this pattern is implemented how we learnt in the lecture. We created the iterator interface and two classes which are called PieceIterator and PlayerIterator, with the important corresponding functions hasnext() and next(). We then implemented two functions (createPieceIterator and CreatePlayerIterator) in our game class which create the respective iterator instance.

2) Class diagram:


<<interface>>
IteratorCreator
+createPieceIterator()
+createPlayerIterator()
<<interface>>
iterator
+hasNext()
+next()
Game
-pieces: Arraylist<ChessPiece>()
-players: Player[]
-gameOver: boolean = false
-color: enum
-letters: enum
-observers: Arraylist<Observer>()
-scoreboard: Scoreboard
-board: Board
-pieceIterator: Iterator
-playerIterator: Iterator
+Game()
+getState()
-placeInitial()
-initializePlayers(names: String)
+play()
-move(p: Player, src: ChessPiece, dest: String): boolean
-capture(p: Player, src: ChessPiece, dest: String): boolean
-whoIsThere(x: Int, y: Int): ChessPiece
-letterToInteger(l: String): int
-kingSuicide(p: Player, src: ChessPiece, dest: String): boolean
-check(p: Player)
-promotion(p: Player, pawn Chesspiece, x: String)
-checkInput(xinput: String, xinput: String): boolean
-checkInputP(xinput: String, xinput: String): boolean
+registerObserver()
+removeObserver()
+notifyObservers()
+createPieceIterator()
+createPlayerIterator()
PieceIterator
-pieces: Arraylist<ChessPiece>()
-position: int = 0
+PieceIterator()
+hasNext()
+next()
PlayerIterator
-players: Arraylist<ChessPiece>()
-position: int = 0
+PlayerIterator
+hasNext()
+next()

3) Sequence diagram:

The following sequence diagram shows the procedure where the program iterates through each player and iterates for each player through each piece to find the right piece to move.

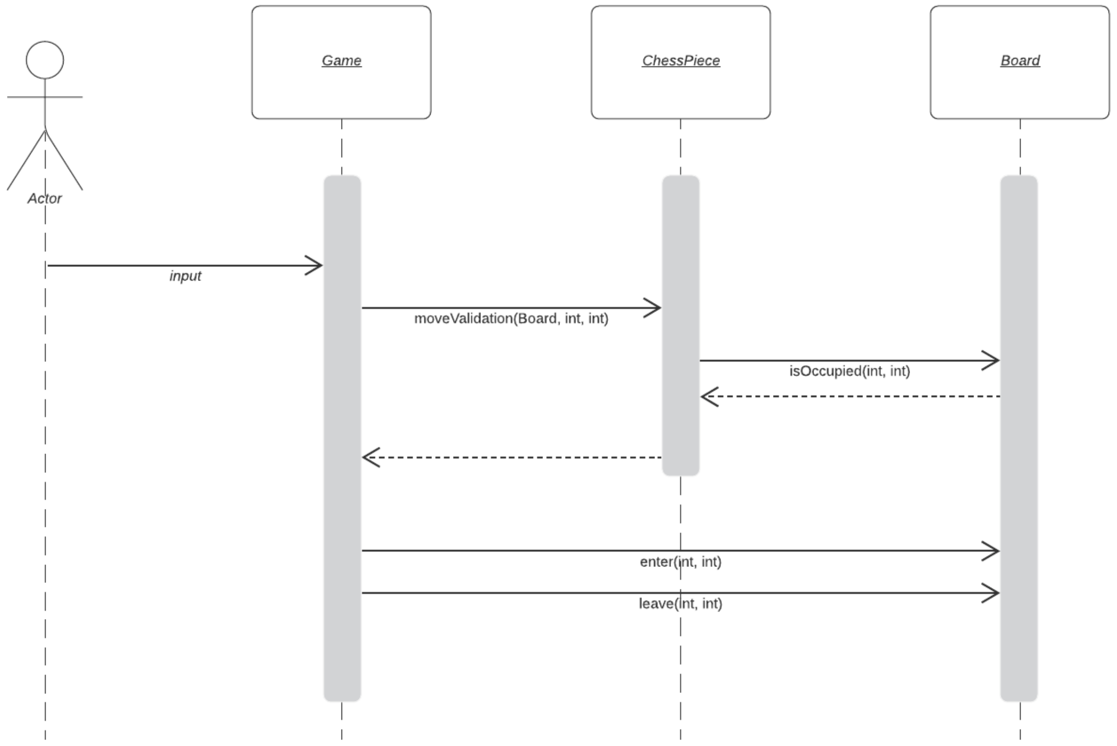


Part 2

Make a move

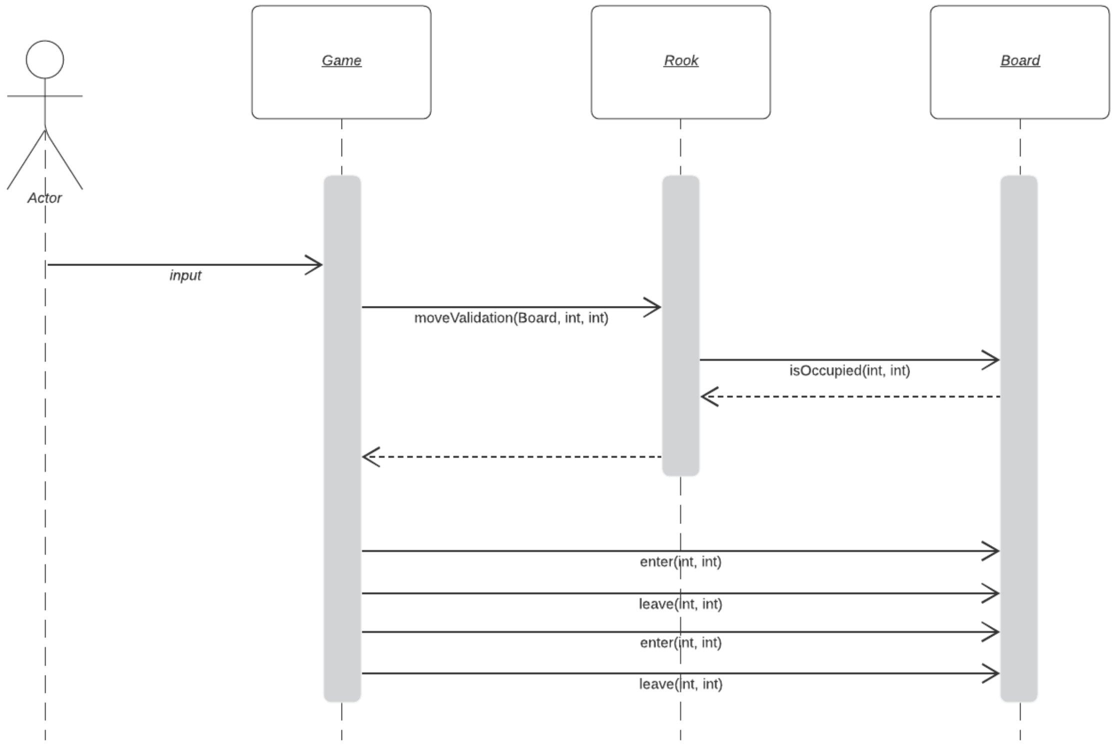
We consolidated all pieces in ChessPiece to get a clear overview. In our sequence diagrams we only considered the classes which interact with the board class.

After the actor has entered his move the Game class calls the validation of the specific piece which can reach the entered position. During the validation the piece checks the occupation of the board. Afterwards when the piece is being moved the occupation of the board is being updated.



Castling

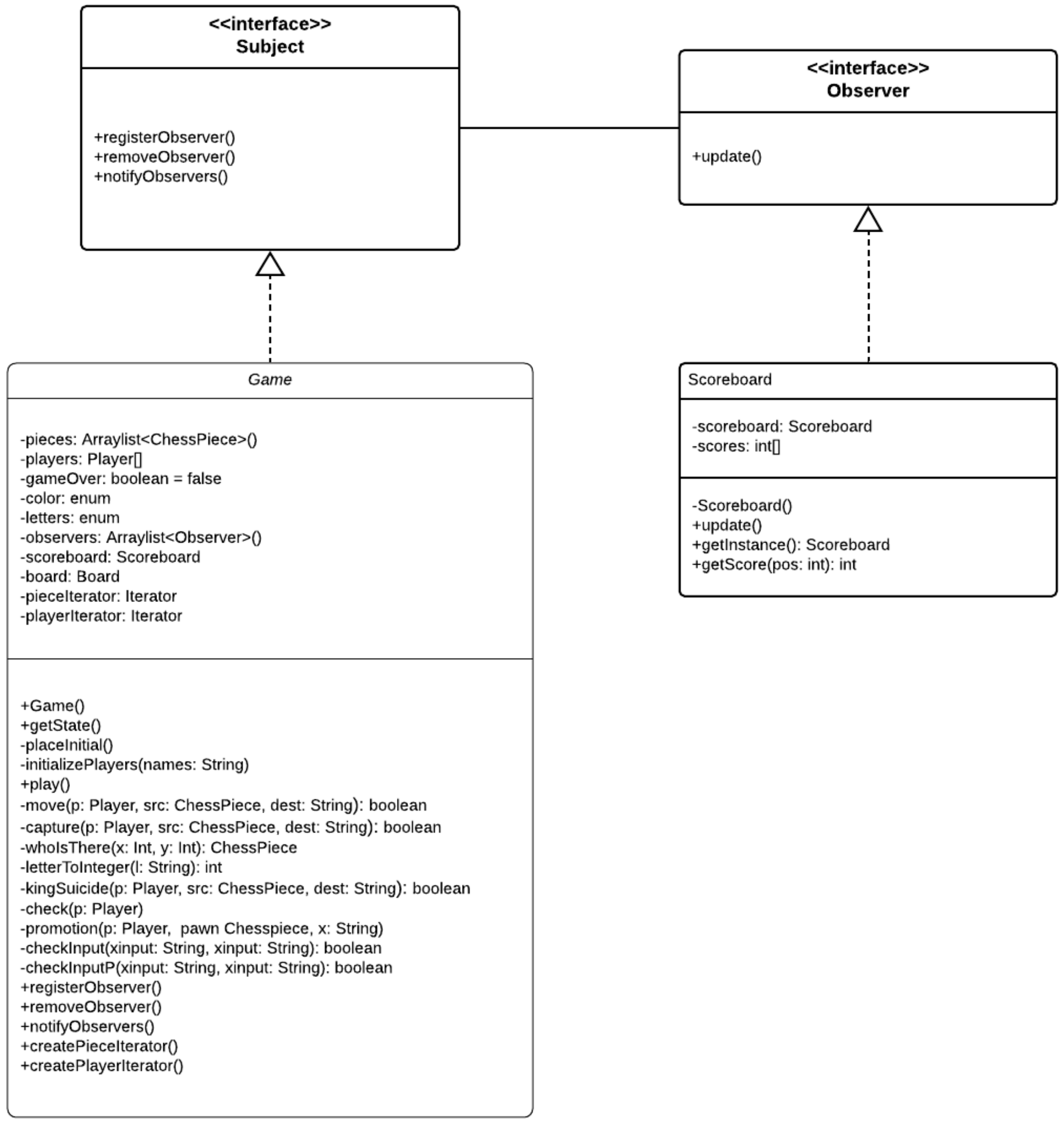
After the actor has entered the castling command the Game class calls the validation of the specific rook which can reach the position of the king. During the validation the rook checks the occupation of the board. When the castling is executed the occupation of the board is being updated. We noticed that we implemented the castling incorrectly. The sequence diagram below shows the new castling procedure which should work but isn't implement in our code.



Part 3

Observer: Scorecard

The following class diagram shows how we implemented the Scorecard using the Obeserver-Pattern. Every time a player captures an enemy piece, he gets five points in case the enemy piece was the queen and one point for all other eaten pieces. Whenever it is the turn of a player, his current points are displayed.



<<interface>>
Subject
+registerObserver()
+removeObserver()
+notifyObservers()
<<interface>>
Observer
+update()
Game
-pieces: Arraylist<ChessPiece>()
-players: Player[]
-gameOver: boolean = false
-color: enum
-letters: enum
-observers: Arraylist<Observer>()
-scoreboard: Scoreboard
-board: Board
-pieceIterator: Iterator
-playerIterator: Iterator
+Game()
+getState()
-placeInitial()
-initializePlayers(names: String)
+play()
-move(p: Player, src: ChessPiece, dest: String): boolean
-capture(p: Player, src: ChessPiece, dest: String): boolean
-whoIsThere(x: Int, y: Int): ChessPiece
-letterToInteger(l: String): int
-kingSuicide(p: Player, src: ChessPiece, dest: String): boolean
-check(p: Player)
-promotion(p: Player, pawn Chesspiece, x: String)
-checkInput(xinput: String, xinput: String): boolean
-checkInputP(xinput: String, xinput: String): boolean
+registerObserver()
+removeObserver()
+notifyObservers()
+createPieceIterator()
+createPlayerIterator()
Scoreboard
-scoreboard: Scoreboard
-scores: int[]
-Scoreboard()
+update()
+getInstance(): Scoreboard
+getScore(pos: int): int