Assignment 2

Remark: The UML files will be provided separately as well to increase their readability.

Ex. 1.1

In the following list we describe the changes we made to our code regarding getters, setters, and public variables. We had to split points 1. and 2., otherwise we would have only had 8 getters/setters or public variables that we can refactor. You can count them as one if you need to when stated by the guidelines, just note that then we won't have more than 8 fields to approach in this exercise. Each specified line refers to the project state of assignment 1.

1. In the class "Game.java" on line 78 we had "setActivePlayer(Player playerstatus)". Since the method is private and is only used in the class itself, it is redundant. We delete the method and will directly access the boolean variable.
2. In the same class on line 82 we had "getActivePlayer()". The method may be public, but it is never used anywhere. For that case we can also simply delete it. We can also directly access the Boolean variable if needed.
3. In the class "Board.java" on line 75 we have "getPiece(int posX, int posY)". This method is essential for us since we store each peace on the board itself. We need to access the board at a specific coordinate to retrieve the piece at that location. We pass arguments and need to access an array in the method. Also, important to note is that the method is not in the "Piece" class. Getting that piece takes place in the "Board" class which means this is not a getter in the meaning of those we need to refactor.
4. In the class "Piece.java" on line 5 we have a public variable "color" of the type "Player" (enum). This means we need to make it private which in return means we need ways to access it. During the game there is no need to change the color so dealing with it in the constructor is more than enough. However, getting the color is definitely needed as we need to compare color in specific turns. This should not be a problem though since a returned enum cannot be mutated.
5. In the same class on line 16 we have the method "setKing()" which changes the boolean. Since the boolean only needs changes to true we don't need to add arguments to the method. To hide implementation details we will rename the method to "promote()".
6. In the same class on line 26 we had the method "setMultiJumping(boolean multiJumping)". We split this method into two new methods that don't take arguments and name them "startMultiJump()" and "endMultiJump()". This hides implementation details and helps us with small code improvements, especially regarding the coming observer implementation.
7. In the class "Turn.java" we have a public variable at line 12. The active player gets already set in the constructor when a turn is instantiated. We only need to retrieve the active player in the same way we need the color to make sure only the right pieces get moved. Thus, we cannot get rid of this getter, we must access that variable.
8. In the same class at line 13 we have another public variable "status". It is no longer needed with the prior mentioned improvements so we can simply delete it.
9. In the same class again at line 10 and 11 we have protected variables of the type "TilePositions". We made them private and created two methods "from()" and "to()" to access those variables.

Ex. 2.1

We decided to extend our project with a BoardLoader which is able to save the current board state and load a prior saved board state. We also implemented a UI that allows to move pieces directly on the board such that console input and output gets obsolete. To start the UI, we also had to add a “main.java” class that initializes the game. For our extensions that we coded, we provide a set of CRC cards and a UML. For both, new methods and new classes are represented in green. Deleted classes and methods are represented in red. Mentioning lines in new classes is useless since the whole class is new.

The addition of the UI brings the needs to initiate it and add more methods. Thus, we have a new method “start(Stage stage)” in the game class that initializes the UI and a “reset()” method that takes care of the clearing and reinitiating of everything. The method “getActivePlayer()” in the “Game.java” class that prior got deleted in Ex.1 had to be readded since new classes, especially the “BoardLoader”, need to access the active player. It makes no sense to pass the active player in arguments through a chain of methods, thus we cannot avoid the “getActivePlayer()” method. We could rename the method to something like “currentActivePlayer()”, but looking forward to Ex. 3 this method will be obsolete nonetheless so we decided to keep it as a placeholder.

The “BoardLoader” itself provides two important methods. The first loads a given board state from a .csv file with certain constraints, throwing exceptions when the file does not meet certain requirements. The second method just saves the current board state to a .csv file such that a user is able to read or even understand the board based on the file.

The purpose of the UI class is to allow the user to interact with the JavaFX window and to visualize the current state of the board after each turn while also showing the status message, which displays all kinds of information such as invalid turns, whose turn it is, or if a player won.

The initialize method initializes the scene including several UI components which are required by other methods that draw these components and calls several methods which draw the current state of the board including the board itself as well as the pieces whose position depends on the position of the pieces on the board at time of initialization. It also calls a method which draws and implements the functionality of the "save", "load", and "new game" buttons.

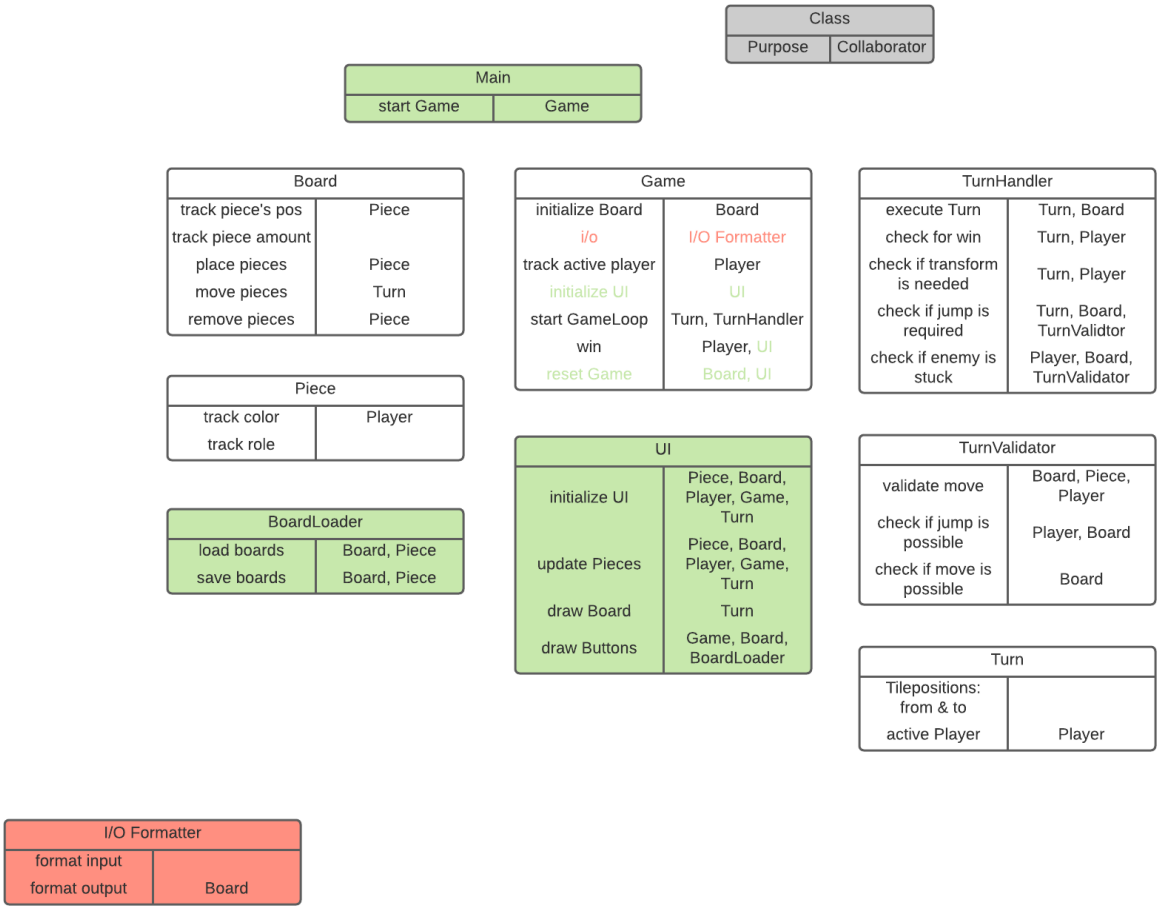
Throughout the game, a player may select one of its pieces by clicking on it, by doing this, the appearance of the piece is slightly modified to give feedback to the user that the piece was indeed selected and internally the coordinates of the selected piece are saved. After selection of the piece, the player may move to any desired tile on the board which unselects the piece, creates a Turn object based on the coordinates of the destination tile, and passes it to the Game class which executes the turn, if valid.

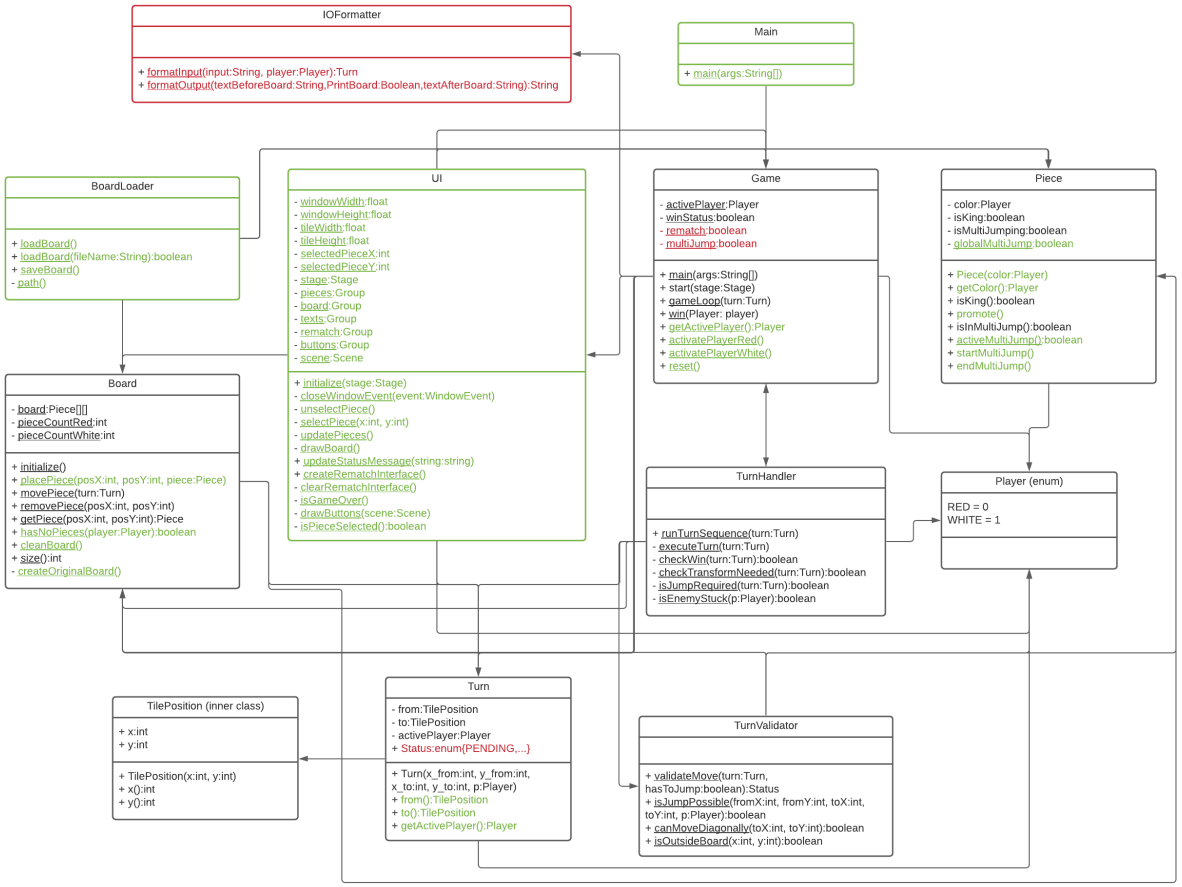
After each successful execution of a turn, the pieces on the board are updated and after each turn, even if unsuccessful, the status message provides feedback.

As soon as one player has won, a rematch interface is drawn giving the player the option to either start a new game or close the application.

If a player decides to close the JavaFX application, a small window will pop up to warn the player if they're sure they want to close the application, since there might be unsaved changes.

Ex. 2.2


Class
Purpose
Collaborator
Main
start Game
Game
Board
track piece's pos
track piece amount
place pieces
move pieces
remove pieces
Piece
Piece
Turn
Piece
Piece
track color
track role
Player
BoardLoader
load boards
save boards
Board, Piece
Board, Piece
Game
initialize Board
i/o
track active player
initialize UI
start GameLoop
win
reset Game
Board
I/O Formatter
Player
UI
Turn, TurnHandler
Player, UI
Board, UI
UI
initialize UI
update Pieces
draw Board
draw Buttons
Piece, Board, Player, Game, Turn
Piece, Board, Player, Game, Turn
Turn
Game, Board, BoardLoader
TurnHandler
execute Turn
check for win
check if transform is needed
check if jump is required
check if enemy is stuck
Turn, Board
Turn, Player
Turn, Player
Turn, Board, TurnValidtor
Player, Board, TurnValidator
TurnValidator
validate move
check if jump is possible
check if move is possible
Board, Piece, Player
Player, Board
Board
Turn
Tilepositions: from & to
active Player
Player
I/O Formatter
format input
format output
Board

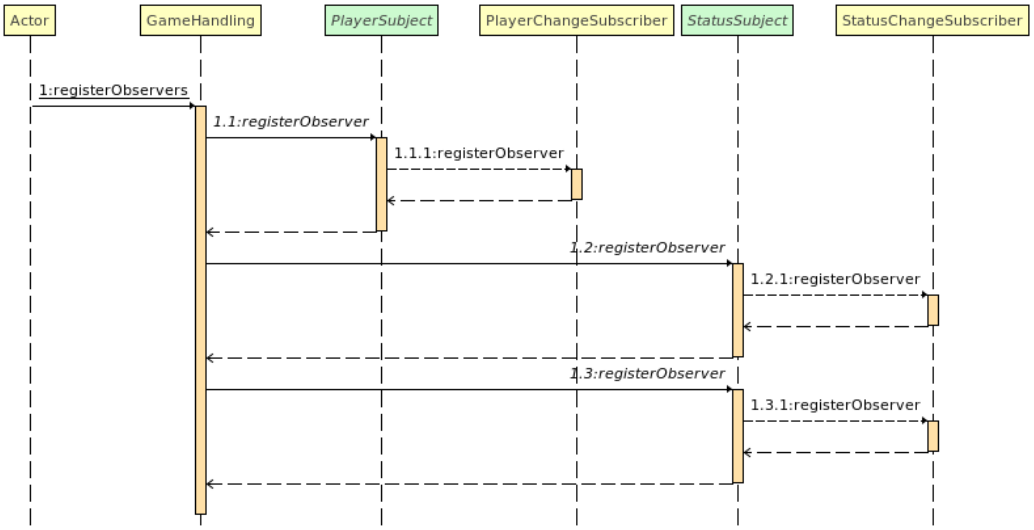


Ex. 3.1

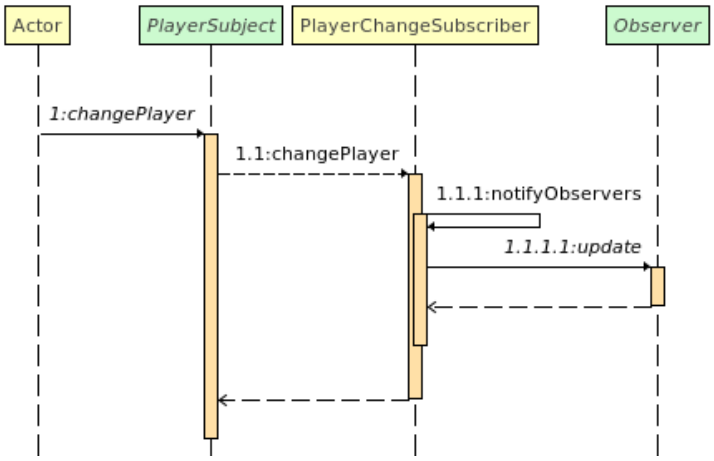
We chose the Observer Pattern for our game as we wanted it to provide cleaner control flow. This is because we felt the previous roundtrip approach between the “Game” and “TurnHandler” class was a bit hacky and difficult to debug. We also needed to pass around the “Turn” objects through the whole logic because the contained information had to be extracted and analyzed at many places. We now let the relevant observers subscribe to the relevant status and player changes in order to be informed automatically. This is a more direct, event-based approach that is easier to maintain and control. After attempting a turn, new status messages for example are directly fed into the “StatusSubject” which then calls the “StatusMessageChannel”’s “update” method that takes care of calling the UI for a text update. Player turns are now also updated using the Observable Pattern approach to be more consistent and take this out of the “Turn” class' responsibility.

Ex. 3.2

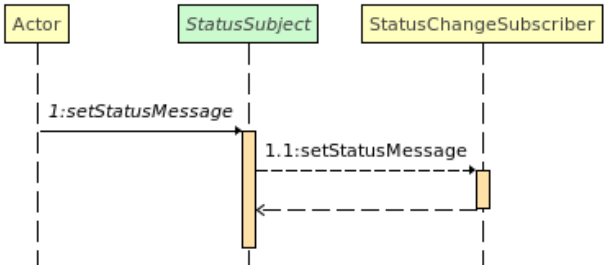
Register observers



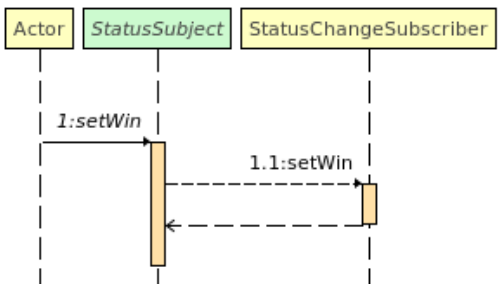
Change player



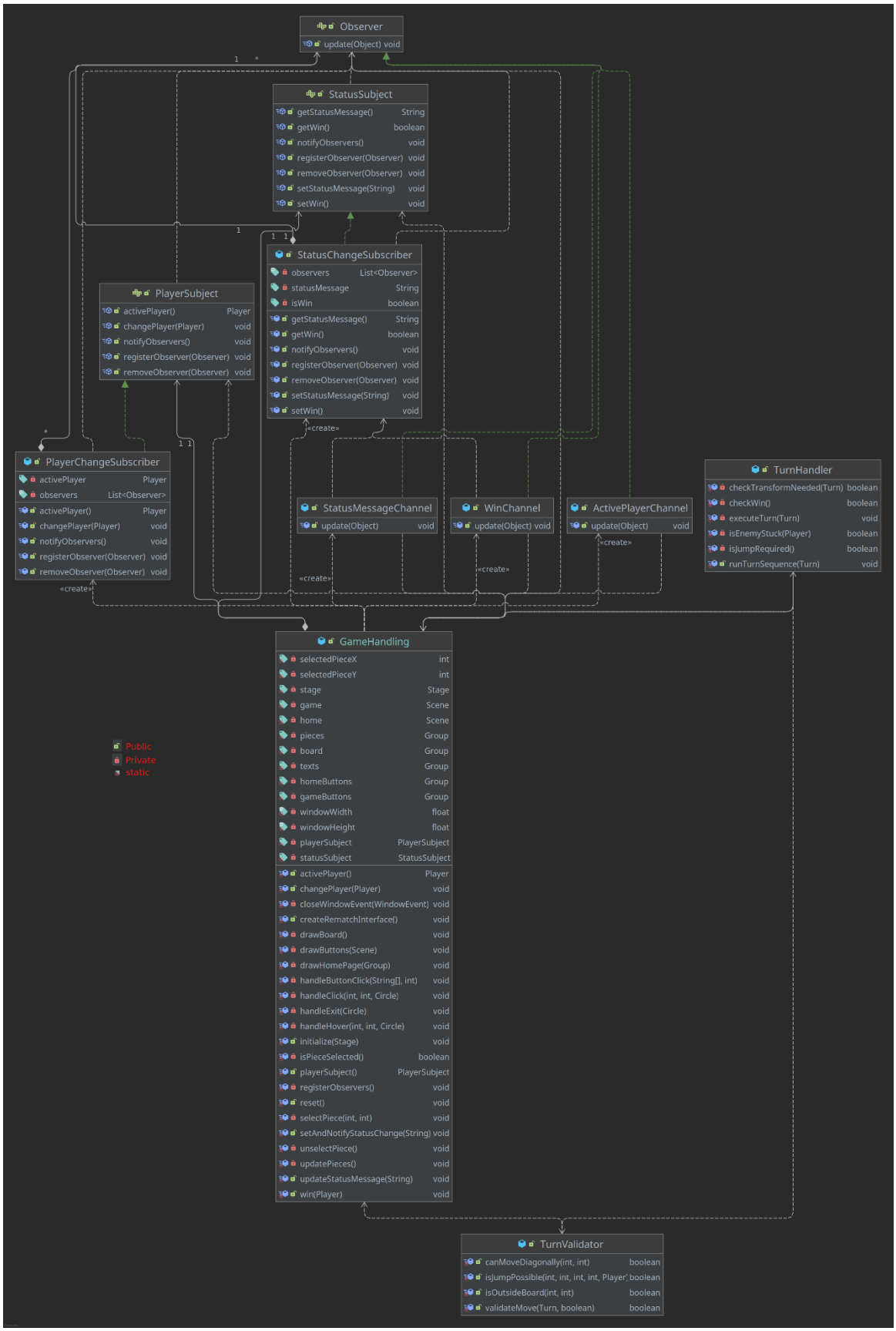
Change status



Win



Ex. 3.3


Observer
update(Object) void
StatusSubject
getStatusMessage() String
getWin() boolean
notifyObservers() void
registerObserver(Observer) void
removeObserver(Observer) void
setStatusMessage(String) void
setWin() void
StatusChangeSubscriber
observers List<Observer>
statusMessage String
isWin boolean
getStatusMessage() String
getWin() boolean
notifyObservers() void
registerObserver(Observer) void
removeObserver(Observer) void
setStatusMessage(String) void
setWin() void
PlayerSubject
activePlayer() Player
changePlayer(Player) void
notifyObservers() void
registerObserver(Observer) void
removeObserver(Observer) void
PlayerChangeSubscriber
activePlayer Player
observers List<Observer>
activePlayer() Player
changePlayer(Player) void
notifyObservers() void
registerObserver(Observer) void
removeObserver(Observer) void
StatusMessageChannel
update(Object) void
WinChannel
update(Object) void
ActivePlayerChannel
update(Object) void
TurnHandler
checkTransformNeeded(Turn) boolean
checkWin() boolean
executeTurn(Turn) void
isEnemyStuck(Player) boolean
isJumpRequired() boolean
runTurnSequence(Turn) void
«create»
GameHandling
selectedPieceX int
selectedPieceY int
stage Stage
game Scene
home Scene
pieces Group
board Group
texts Group
homeButtons Group
gameButtons Group
windowWidth float
windowHeight float
playerSubject PlayerSubject
statusSubject StatusSubject
activePlayer() Player
changePlayer(Player) void
closeWindowEvent(WindowEvent) void
createRematchInterface() void
drawBoard() void
drawButtons(Scene) void
drawHomePage(Group) void
handleButtonClick(String[], int) void
handleClick(int, int, Circle) void
handleExit(Circle) void
handleHover(int, int, Circle) void
initialize(Stage) void
isPieceSelected() boolean
playerSubject() PlayerSubject
registerObservers() void
reset() void
selectPiece(int, int) void
setAndNotifyStatusChange(String) void
unselectPiece() void
updatePieces() void
updateStatusMessage(String) void
win(Player) void
Public
Private
static
TurnValidator
canMoveDiagonally(int, int) boolean
isJumpPossible(int, int, int, int, Player) boolean
isOutsideBoard(int, int) boolean
validateMove(Turn, boolean) boolean