Checkers

ISTE.442.01. - Web Application Development

Roman Pusec

Abstract

This manual describes the entire internal structure and the purpose of my final project from Web Application Development. We were assigned to develop a turn-based multiplayer game where at least two players can interact with one another. My version of the game was built with PHP working on the backend (handling user validations, and sending game states to the client) and JavaScript (plus external libraries) on the frontend.

In terms of technology, meekrodb was used for handling all sorts of database calls, and the rest architecture was built with pure PHP. As for the frontend, I've used the CreateJS, which offers abstractions for the visual aspect for the canvas, already preprogrammed animation algorithms, and other things that were not required for this project (e.g. sound, preloading assets).

The software architecture (on the backend) is based upon a slightly amended version of the MVC paradigm (includes Views, Controllers, Business classes, and database utility classes). The architecture of the frontend is diverged into Game Components, AJAX calls, and Business classes.

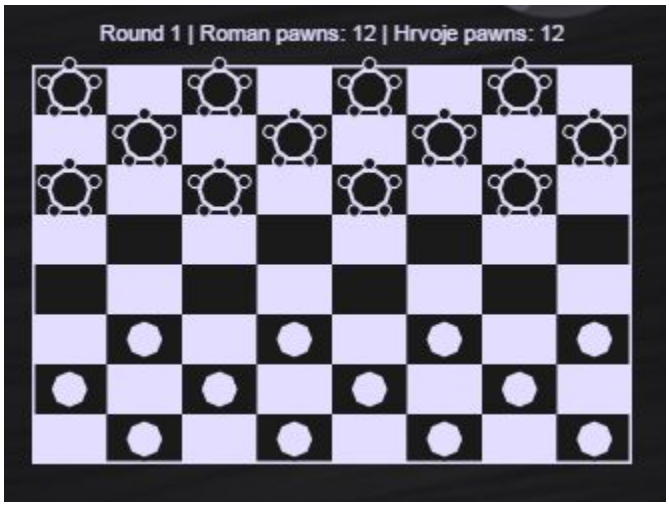
All of these features will be discussed in detail in this document.

Game Info

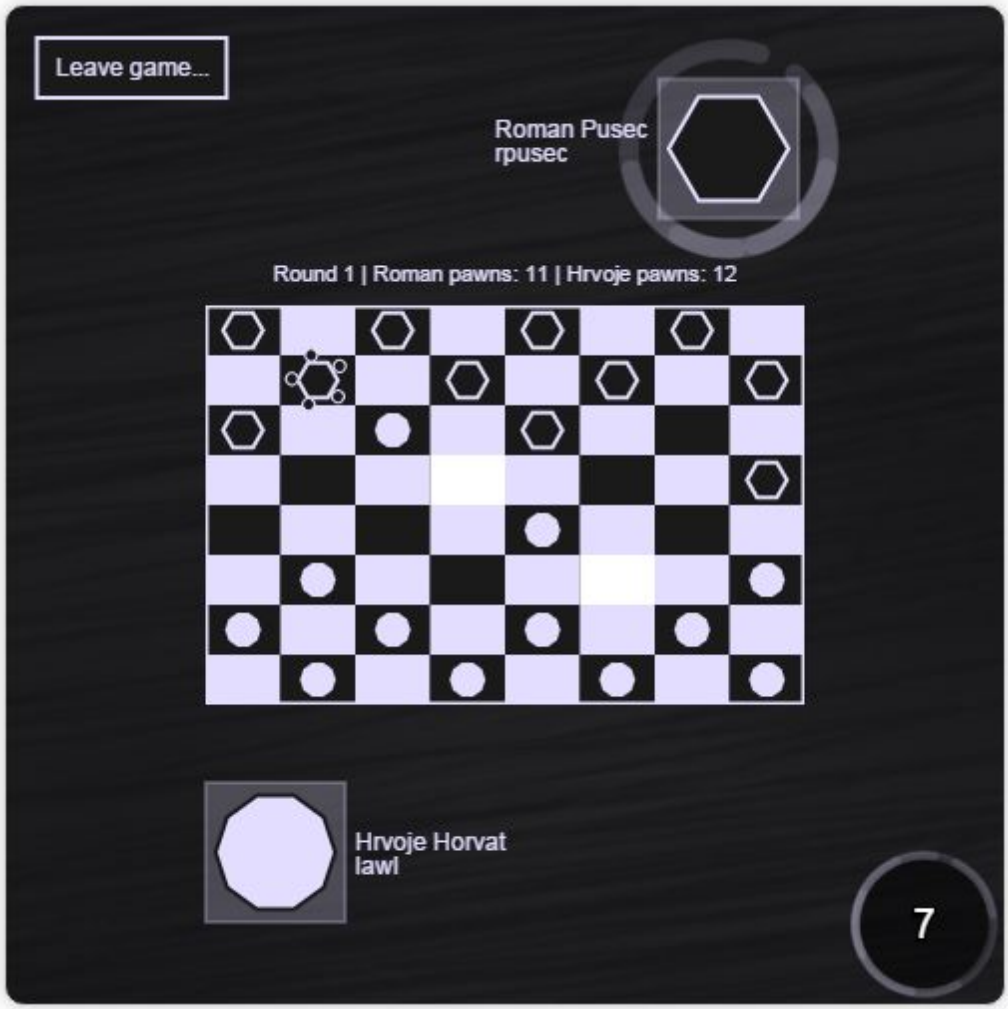
The game is called Checkers. It is a two player turn-based game and its goal is to be first to eliminate all of the pawns of your opponent.

Gameplay

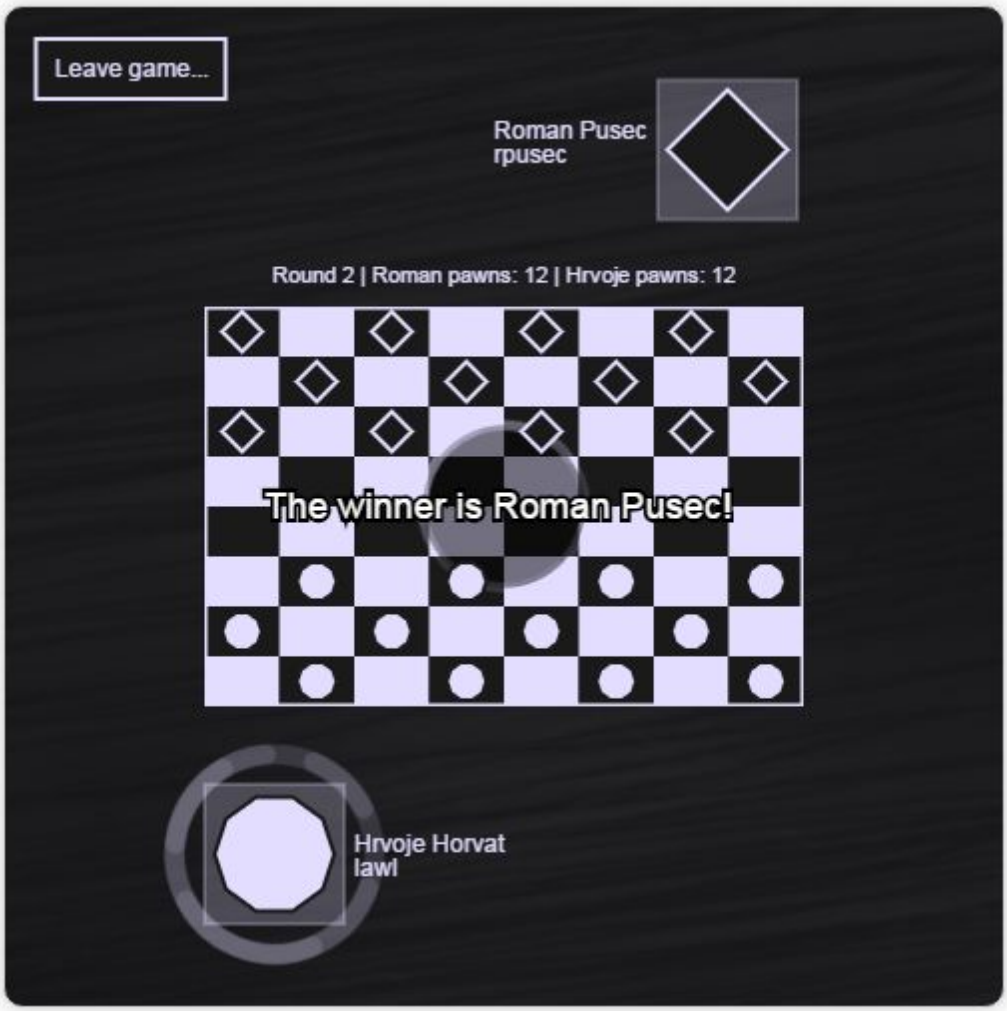
The pawns can be moved diagonally only, and the pawn cannot be placed on a black square. To eliminate an opponent pawn (or multiple opponent pawns), a player pawn has to be positioned right beside their opponent's pawn, then the player can move their pawn over their opponent's pawn, which would then eliminate the opponent's pawn. The pawns can be killed off recursively if multiple opponent pawns are positioned diagonally, with one square of space between them, the player can then eliminate multiple opponent pawns with a single turn - by placing their pawn after the last opponent pawn.



Both player's and opponent's pawns are different in shape and color. And the pawns that the player can choose are highlighted (tiny circles appear around them).

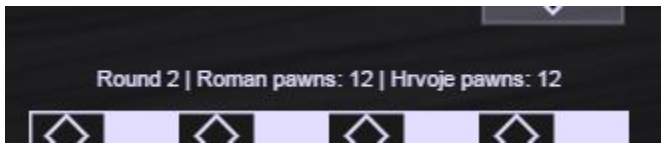


The screenshot shows a portion of the gameplay. The white squares indicate where the player can move their pawns. The diamond shaped pawns (the first twelve from above) belong to the user **rpusec**, and that player chose the pawn that's located on the second column and second row. Since there's a gap between two of the **lawl**'s pawns, **rpusec** can eliminate two of them by clicking on the second white square (the one that's located at the sixth column and sixth row). Lastly, since it is **rpusec**'s turn, their avatar is highlighted with tansparent lines that circle around.



The winner of a round is displayed on the screen and the pawns are reassigned to both players.

Each player turn lasts for 10 seconds, in case the player was imobile for more than 10 seconds, the turn is then passed to the opponent. There's a small timer appearing on the bottom right corner which displays the time.



Right above the board, you can see the amount of your pawns, your opponent's pawns, and which round it currently is.

On the top-right corner is a button which is quite self-explanatory, the player can leave the current game by clicking it.

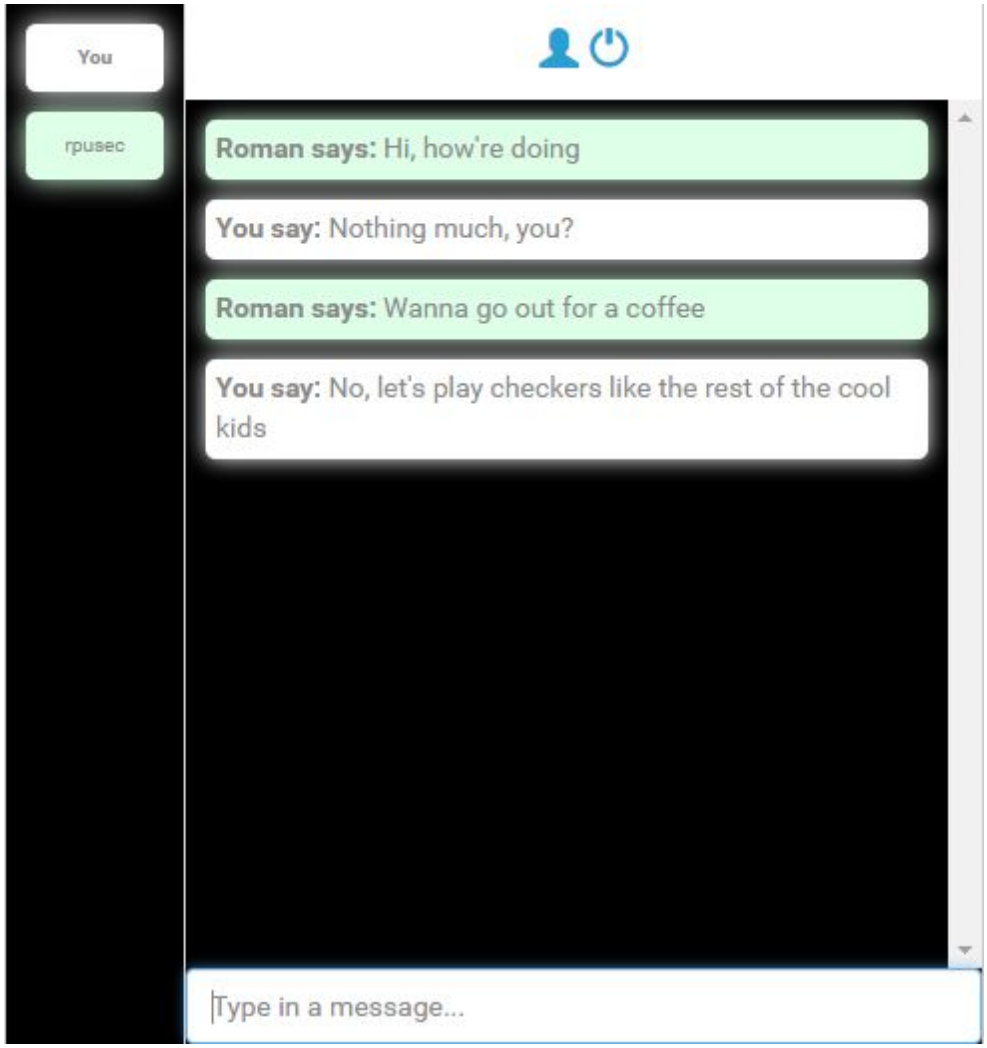
Once you enter a game room (discussed below), you'll be seeing a screen which displays "Waiting for second player...".

Main Menu

The main menu is composed of six game rooms. A game room represents a separate place where two players can play the game. Since there are 6 game rooms, a maximum of 12 players can use the application.

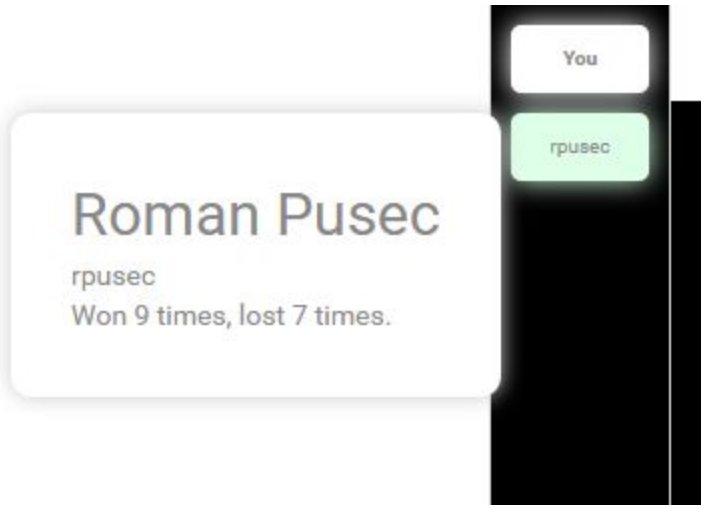
Users can also see if other users are waiting for their opponent from the main menu, as in the example above, the user **rpusec** is waiting for their opponent.

Chat Menu



The chat menu is pretty self-explanatoriy - it is composed of the list of users on the left side, the messages on the right side, and the message input field below.

The first icon (person) redirects you to the accout settings, and the second icon logs you off the game. More on these options later.

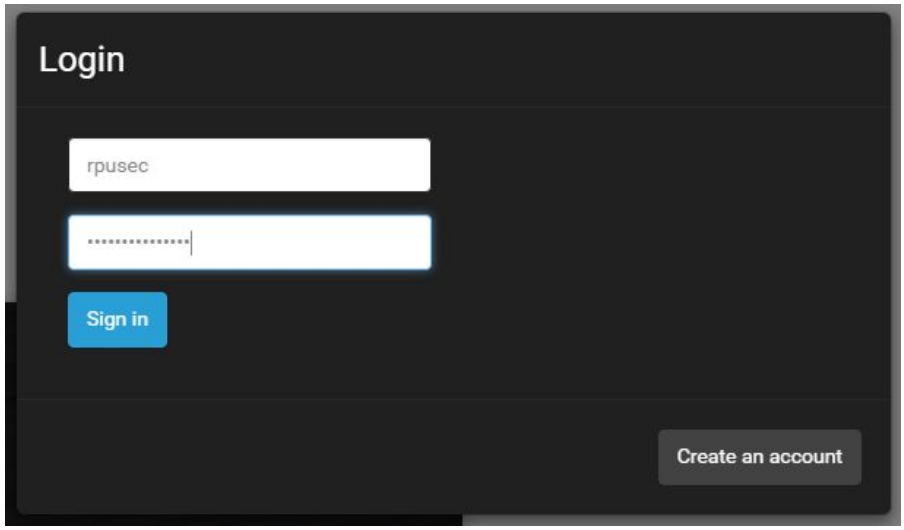


If you hover over a particular user on the chat list, you'd see more of their information, such as their first and last name, username, the amount of times they have won and lost.

The chat in the main menu is public, so everyone can chat with everybody. However during gameplay, the chat between the player and their opponent is private, so it's only between them.

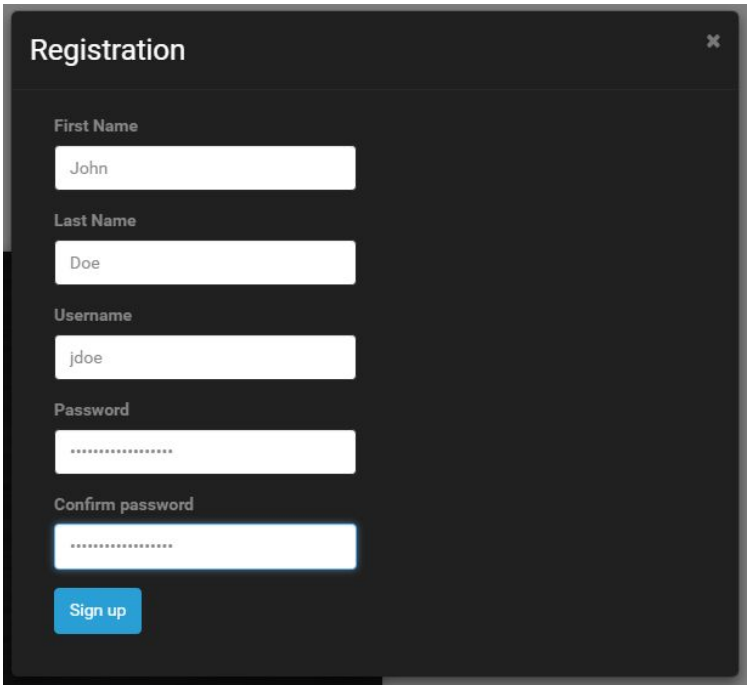
You should also keep in mind that the color of your game interface matches your chat bubbles. Basically, if your canvas interface is red, then your chat bubbles will also be red. That color is assigned upon login.

Login Screen



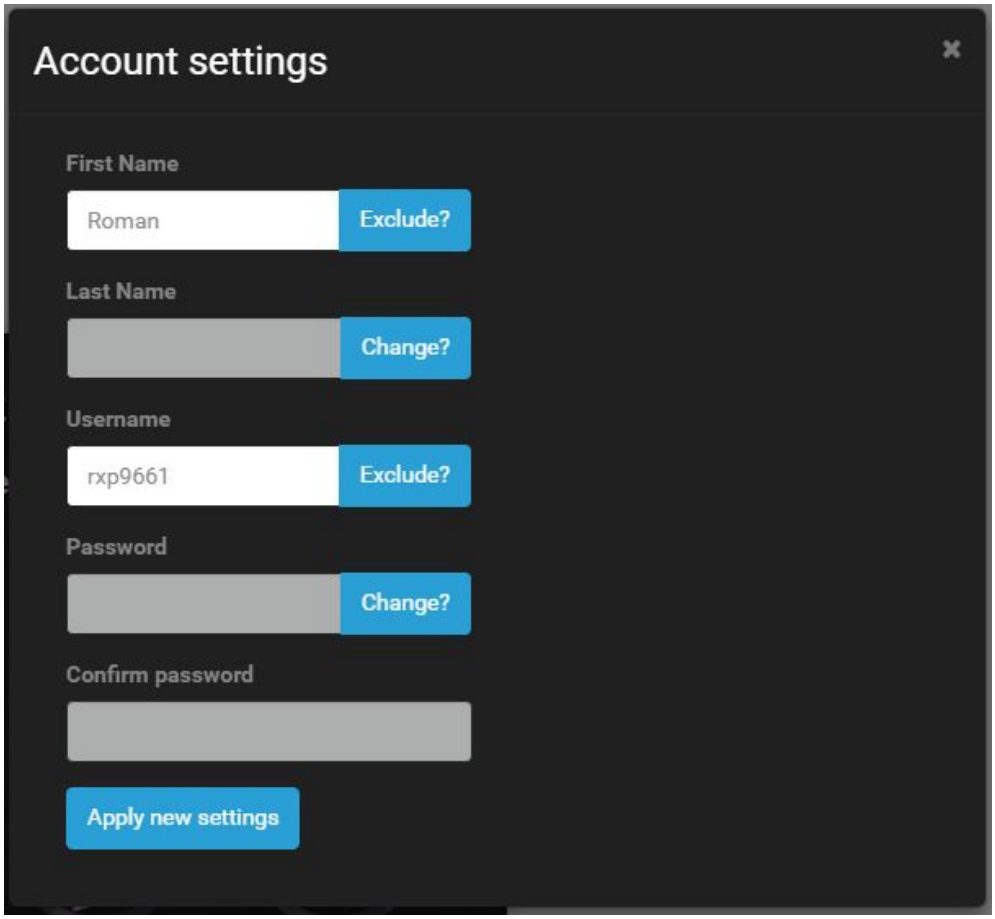
This is a screenshot of the login screen. The user can insert their credentials and sign in or signup by using the "*Create an account*" option.

Registration Screen

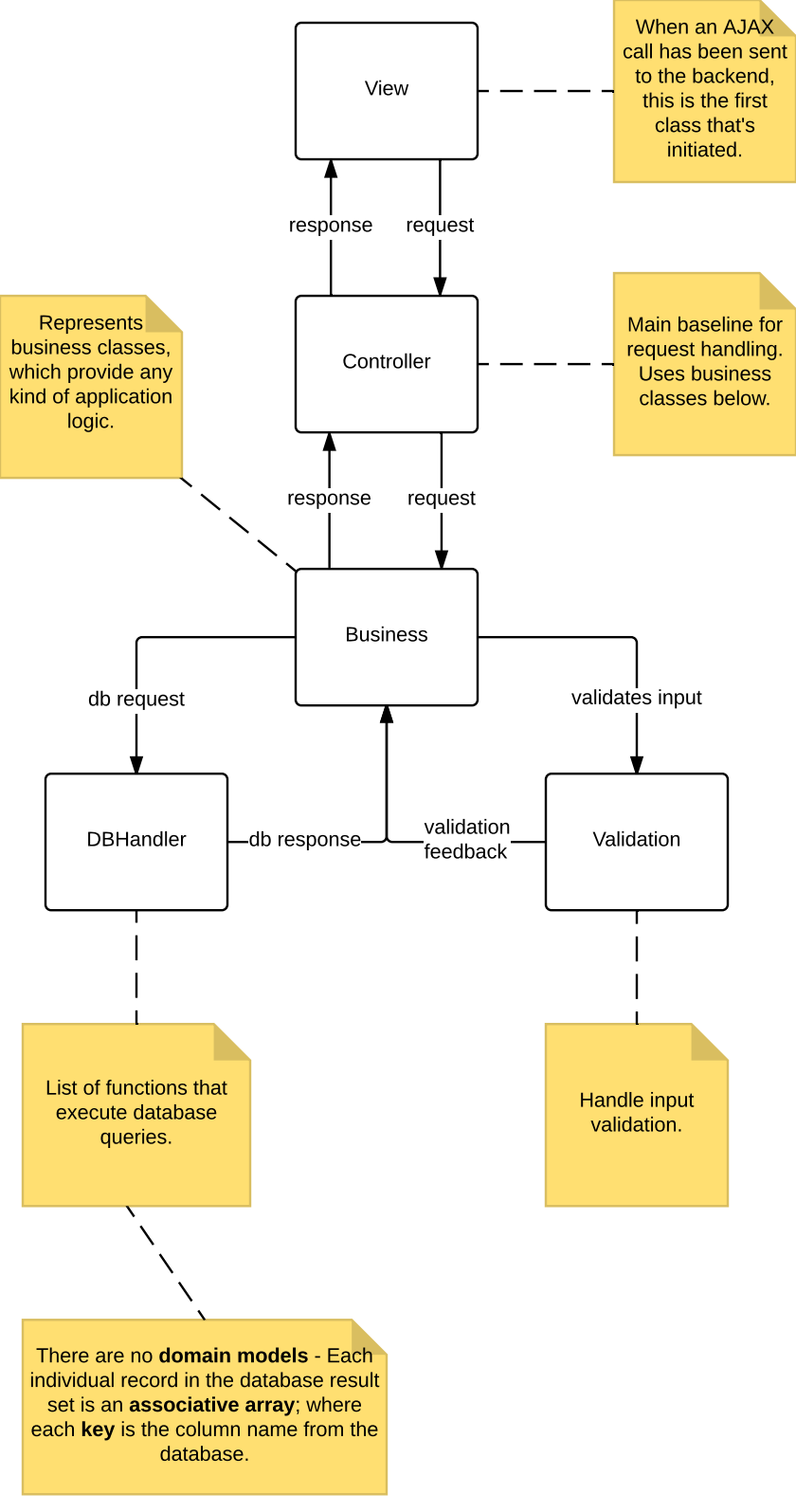


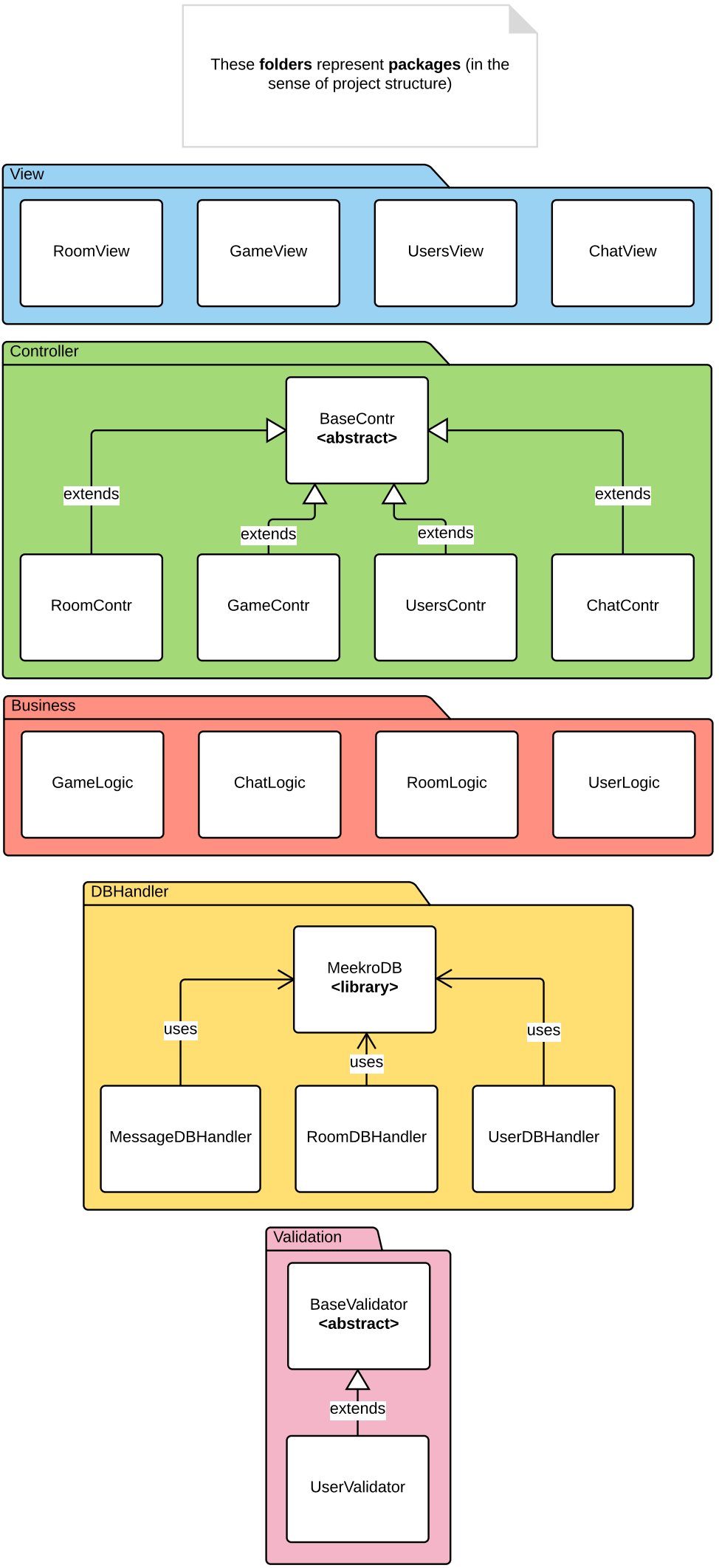
This is what the user registration screen looks like (it is displayed upon clicking on the "*Create an account*" option on the login frame).

User account settings



Changes the user settings. By pressing "*Change?*" on the right of an input field enables you to change the value of that field in the database, and only the fields that were changed will be updated. In case you change your mind and don't want to update a particular field that you were already amending, then simply press the "*Exclude?*" button on the right side of that particular input field. This window can be accessed by clicking on the blue **person icon** on the chat window.


View
When an AJAX call has been sent to the backend, this is the first class that's initiated.
response
request
Controller
Main baseline for request handling. Uses business classes below.
Represents business classes, which provide any kind of application logic.
response
request
Business
db request
validates input
DBHandler
db response
validation feedback
Validation
List of functions that execute database queries.
Handle input validation.
There are no **domain models** - Each individual record in the database result set is an **associative array**; where each **key** is the column name from the database.


These **folders** represent **packages** (in the sense of project structure)
View
RoomView
GameView
UsersView
ChatView
Controller
BaseContr
<abstract>
extends
extends
extends
extends
RoomContr
GameContr
UsersContr
ChatContr
Business
GameLogic
ChatLogic
RoomLogic
UserLogic
DBHandler
MeekroDB
<library>
uses
uses
uses
MessageDBHandler
RoomDBHandler
UserDBHandler
Validation
BaseValidator
<abstract>
extends
UserValidator

GameAJAXCall
<game.ajax.AJAXCallHandler>

GameView
<View>

GameController
<Controller>

GameLogic
<Business>

RoomDBHandler
<DBHandler>

This diagram explains the execution flow of whenever a **player makes their move**.

Each **View** contains a single **switch statement**, and each **case** calls a particular method from the **Controller** class that the **View** represents. In this example, the **case** is called **evaluate-player-move**, and there are others, such as check-if-opponent-is-done, check-if-a-player-left, etc.

This **switch case** checks whether the move that the player made is **valid**, so it calls validation methods that are stored in the **GameLogic business class**. Each one of these methods either returns **null** (indicating that the validation was successful), or **an array of errors**.

case: evaluate-player-move

evaluatePlayerMove(prevX, prevY, newX, newY)

isCoordinateInitializedAppropriately(prevX, prevY, 2, false)

returns **null** or **array with errors**

isCoordinateInitializedAppropriately(newX, newY, 2, true)

returns **null** or **array with errors**

getAllRoomInfoByUserID(getLoggedUserID())

getAllRoomInfoByUserID(loggedUserID)

return array of room info

returns array of room info

checkIfPlayerTurn(targetRoom, getLoggedUserID())

returns **null** or **array of errors**

checkIfPrevPosIsNotAPawn(prevPosition)

returns **null** or **array of errors**

checkIfActualUserPawn(prevPosition, getPlayerNumber())

returns **null** or **array of errors**

checkIfPrevPosEqualToNewOne(newPosition, prevPosition)

returns **null** or **array of errors**

checkIfNewPosRepresentsAPawn(newPosition)

returns **null** or **array of errors**

diagonallyDismissOpponentsAllSides(...)

diagonallyDismissOpponents(..., LEFT_UP_DIR, ...)

diagonallyDismissOpponents(..., RIGHT_UP_DIR, ...)

diagonallyDismissOpponents(..., LEFT_DOWN_DIR, ...)

diagonallyDismissOpponents(..., RIGHT_DOWN_DIR, ...)

returns errors in String format (or an empty String)

createModifiedStringifiedBoard(boardRows, plOnePawns, plTwoPawns)

returns new stringified board

setOpponentTurn(targetRoom, getLoggedUserID())

returns the ID of the user who's turn it currently is

checkTurnDuration(getTimeInMills())

returns **null** or **an array of errors**

checkForWinner(playerOnePawns, playerTwoPawns)

returns **null** or **user ID of the winner**

updateRoom(arrayProps, roomID)

updateRoom(arrayProps, roomID)

returns properties that will be forwarded to the frontend

Forwarded via AJAX

Whenever **an error** is returned from one of the validation functions, the **evaluatePlayerMove()** method is then terminated and the **array of errors** are forwarded to the **frontend**.

The validation functions are indicated with **dark green** in the diagram, and are as follows...

isCoordinateInitializedAppropriately() - Checks whether the new/old coordinate of a chosen pawn is positioned in a black square on the board.
Note: The number **2** helps indicate the valid position of the pawn (since it checks for each **two** squares).

checkIfPlayerTurn() - Checks whether it's the player's (your) turn.

checkIfPrevPosIsNotAPawn() - Self-explanatory.

checkIfActualUserPawn() - Checks whether the chosen pawn actually belongs to you (hence why the first parameter is the previousPosition).

checkIfPrevPosEqualToNewOne() - Checks if the new and old positions are equivalent.

checkIfNewPosRepresentsAPawn() - Self-explanatory. New position should not be a pawn but an empty field.

checkTurnDuration() - Checks how long the player's turn lasted. If it lasted too long (longer than the **TURN_DURATION** constant allows), then the appropriate error is returned.

checkForWinner() - Checks whether there's a winner. Either returns the ID of the winner, or null if there are no winners.

Parameters are as follows:
- prevX
- prevY
- newX
- newY
- playerNumber
- boardRows
- removedPawnIDs
- errorMsg

The functions on the left share these same parameters, in addition to a constant (**e.g. LEFT_UP_DIR**) that points to which direction should the opponent pawns be killed off.

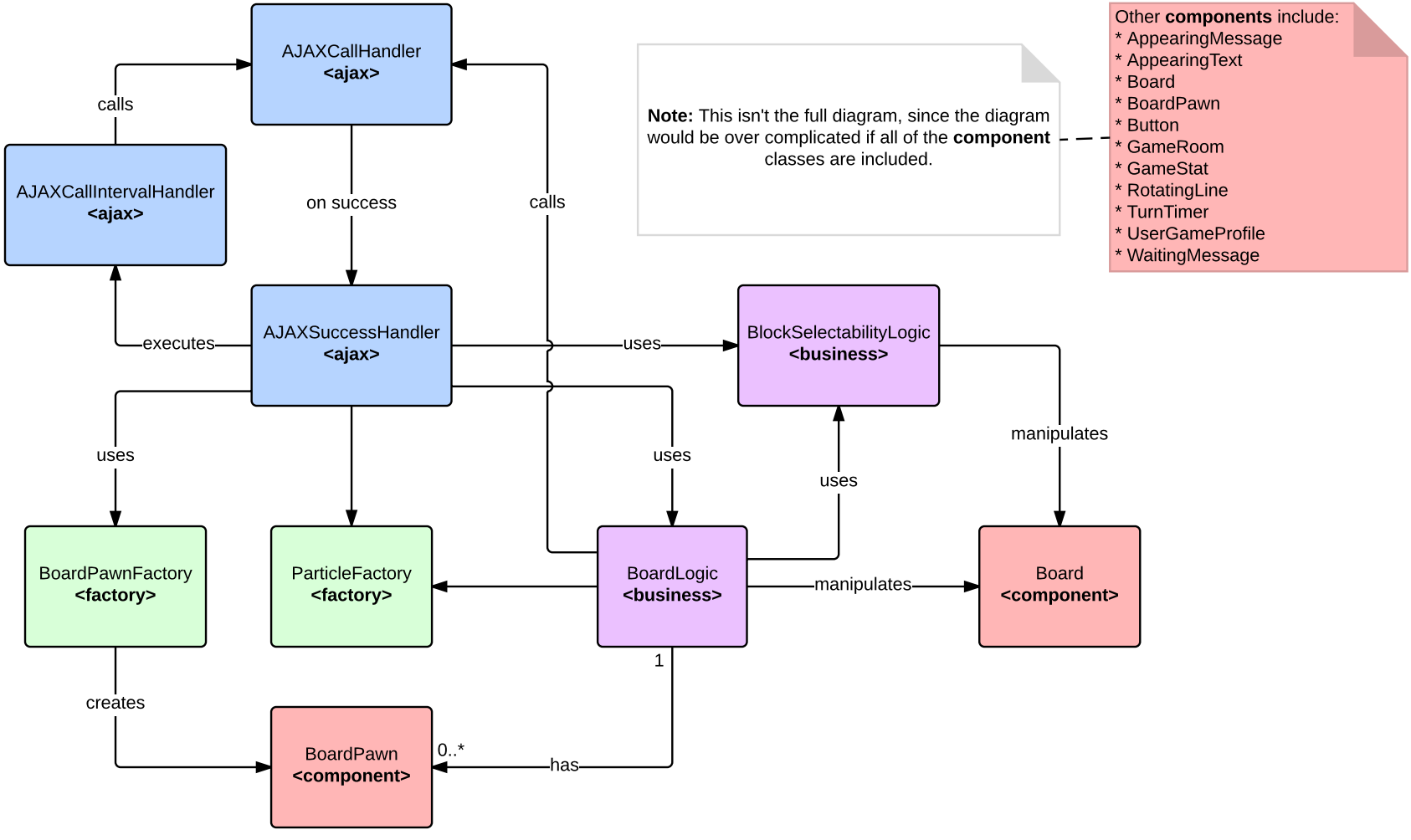
arrayProps is an associative array that consists of the following properties:
- whoseTurn
- stringifiedBoard
- lastMove
- removedPawns

All of the values of these properties are gathered from the returned values from the **GameLogic** methods.

Those properties are as follows:
- success (*which is true since vallidations have passed*)
- prevCoordinate
- newCoordinate
- playerNumber
- removedPawns
- winner

The values of these parameters will then be passed down to the frontend via AJAX, which will then update the page.

Class Diagram (Game) - High Level



The following high level diagram displays the **game portion** of the application (there are two other portions which manage **chat** and **user** operations, they won't be covered, **but they work in a similar principle (e.g. chat and user also have the AJAX classes as provided in this class diagram)**).

AJAXCallHandler - Class which is responsible for sending out an AJAX request to the backend.

AJAXSuccessHandler - Class which contains a stream of methods that are executed when the AJAX request that they represent was handled successfully.

AJAXCallIntervalHander - Makes certain AJAXCallHandler methods execute themselves after a certhain amount of milliseconds (by using setIntreval function). As an example, this is useful for repeatedly checking if your opponent made their move.

BoardPawnFactory - Creates BoardPawn objects.

ParticleFactory - Creates particles (when placing a pawn).

BoardLogic - Handles all of the logic regarding placing a pawn.

BlockSelectabilityLogic - Handles all of the logic regarding making board fields selectable or unselectable (in the sense that you can place a pawn there).

Board - Component which represents a board.

BoardPawn - Component which represents a single pawn.

Class Diagram (Game) - Low Level

AJAX

AJAXCallHandler <ajax>

- wmLoading:WaitingMessage

+ initialize(props):void
+ toGameRoomAJAXCall(targetRoomID:int):void
+ offGameRoomAJAXCall():void
+ checkGameRoomAvailabilityAJAXCall():void
+ displayAllRoomsAJAXCall():void
+ checkForOpponentAJAXCall():void
+ whoseTurnAJAXCall():void
+ evaluatePlayerMoveAJAXCall(prevX:int, prevY:int, newX:int, newY:int):void
+ checkIfOpponentIsDoneAJAXCall():void
+ checkIfAPlayerLeftAJAXCall():void
+ notifyTimeOutAJAXCall():void

- displayWMLoading():void

AJAXCallIntervalHandler <ajax>

- checkForOpponentInterval:int
- checkRoomAvailabilityInterval:int
- checkIfOpponentIsDoneInterval:int
- checkIfAPlayerLeftInterval:int

+ initialize():void

+ setCheckForOpponentInterval():void
+ setCheckRoomAvailabilityInterval():void
+ setCheckIfOpponentIsDoneInterval():void
+ setCheckIfAPlayerLeftInterval():void

+ clearCheckForOpponentInterval():void
+ clearCheckRoomAvailabilityInterval():void
+ clearCheckIfOpponentIsDoneInterval():void
+ clearCheckIfAPlayerLeftInterval():void

+ clearAllIntervals():void

AJAXSuccessHandler <ajax>

- stage:createjs.Stage
- gameNameText:AppearingText
- selARoomText:AppearingText
- board:Board
- btnLeaveGame:Button
- wmSecondPlayer:WaitingMessage
- contGameRoom:createjs.Container
- turnTimer:TurnTimer
- wmLoading:WaitingMessage
- gameStat:GameStat
- playerOnePawns:Array<BoardPawn>
- playerTwoPawns:Array<BoardPawn>
- playerOneProfile:UserGameProfile
- playerTwoProfile:UserGameProfile

+ initializeAllProperties(props):void
+ toGameRoomSuccessHandler(data):void
+ offGameRoomSuccessHandler(data):void
+ checkGameRoomAvailabilitySuccessHandler(data):void
+ displayAllRoomsSuccessHandler(data):void
+ checkForOpponentSuccessHandler(data):void
+ whoseTurnSuccessHandler(data):void
+ evaluatePlayerMoveSuccessHandler(data):void
+ checkIfOpponentIsDoneSuccessHandler(data):void
+ checkIfAPlayerLeftSuccessHandler(data):void
+ notifyTimeOutSuccessHandler(data):void

- hideSecondPlayerWaitingMessage():void
- movePlayer(targetPawn:BoardPawn, newCoordinate:Point, onCompleteFunct:Function):void
- removePawnsFromBoard(targetPawns:Array, playerNum:int):void
- createAndSetupPawns(playerNumber):void
- announceWinner(winnerPlayerNumber):void
- setupAndDisplayGameStat():void
- hideWMLoading():void

Factory

BoardPawnFactory <factory>

- sides:int
- playerOneSides:int
- playerTwoSides:int

- getPlayerOnePawnOptions():Object
- getPlayerTwoPawnOptions():Object
+ createPlayerOnePawns(board:Board, amount:int):Array<BoardPawn>
+ createPlayerTwoPawns(board:Board, amount:int):Array<BoardPawn>
+ resetSides():void
- createPawns(amount:int, pawnOptions:Object, board:Board, isPlayerOne:boolean, playerSides:int):Array<BoardPawn>
- positionPawns(arrPawn:Array<BoardPawn>, board:Board, isPlayerOne:boolean):void

ParticleFactory <factory>

- stage:createjs.Stage
- particles:Array<createjs.Shape>

+ initialize(options:Object):void
+ spawnParticles(point:createjs.Point):void

Business

BlockSelectabilityLogic <business>

- board:Board
- playerOnePawns:Array<BoardPawn>
- playerTwoPawns:Array<BoardPawn>
- targetBoardBlocks:Array<createjs.Shape>

+ makeBoardBlocksSelectable(currSelectedPawn:BoardPawn):void
+ makeBoardBlocksUnselectable():void
+ setBoard(board:Board):void
+ setPlayerOnePawns(plOnePawns:Array<BoardPawn>):void
+ setPlayerTwoPawns(plTwoPawns:Array<BoardPawn>):void
+ findBoardBlockByCoordinates(point:createjs.Point, ...<createjs.Point>):Array<createjs.Shape>

- makeBoardBlocksSelectableFrom(targetCoordinate:createjs.Point, playerNumber:int, whichSide:int)
- makeBoardBlocksSelectableFromCase(targetCoordinate:createjs.Point, playerNumber:int, whichSide:int, createPointForFunction:Function)
- findBoardPawnsByIds(...<Integer>)
- findBoardPawnsByCoordinates(playerNumber:int)
- emitSingleElementOrFullList(targetArray:Array<Object>)
- createPointForLeftUp(pawnX:int, pawnY:int, offset:int)
- createPointForRightUp(pawnX:int, pawnY:int, offset:int)
- createPointForLeftDown(pawnX:int, pawnY:int, offset:int)
- createPointForRightDown(pawnX:int, pawnY:int, offset:int)

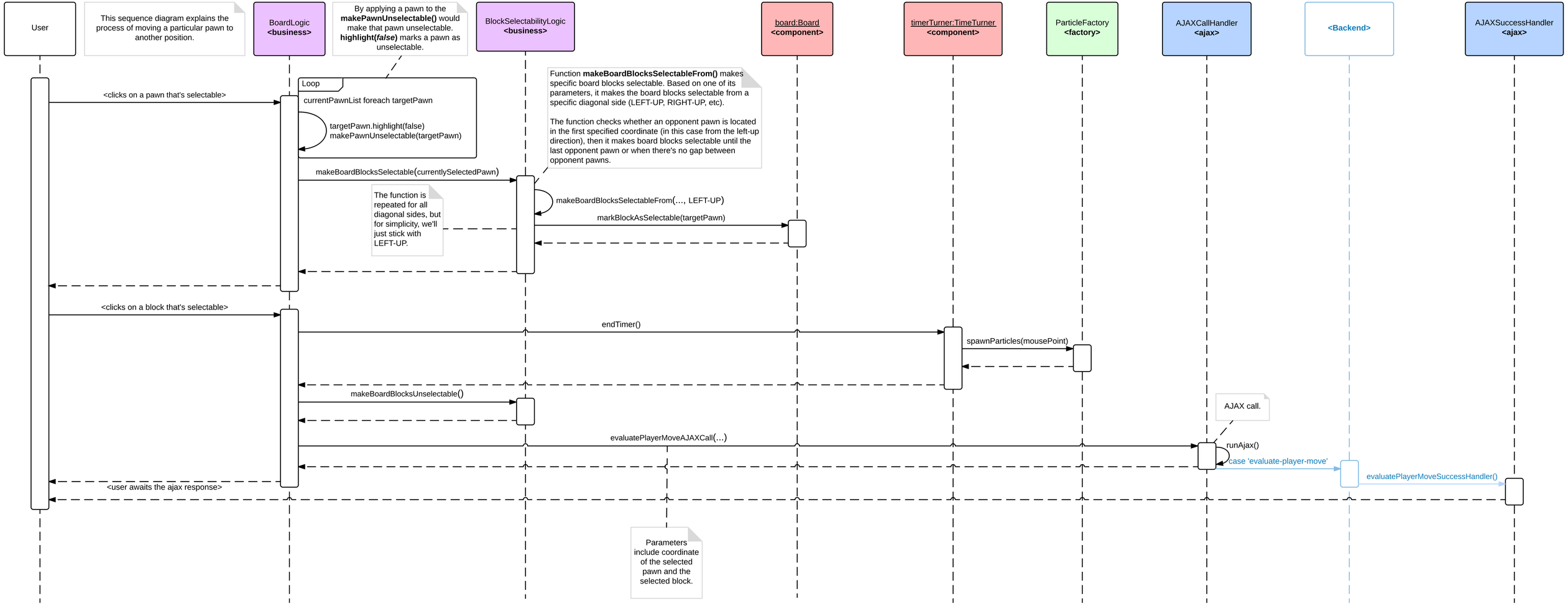
BoardLogic <business>

- board:Board
- currentPawnList:Array<BoardPawn>
- turnTimer:TurnTimer

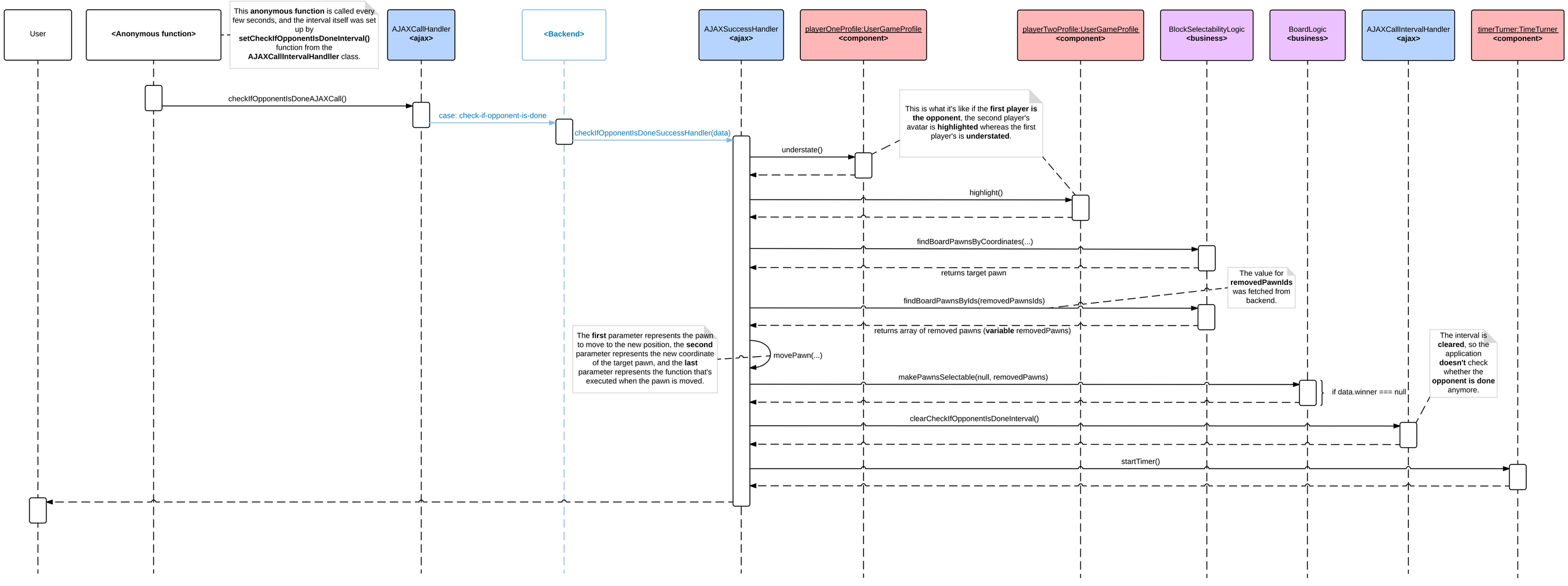
+ initialize():void
+ setBoard(boardBoard):void
+ setCurrentPawnList(currentPawnList:Array<BoardPawn>):void
+ setTurnTimer(turnTimer:TurnTimer):void
+ makePawnsSelectable(arrPawns:Array<BoardPawn>, pawnsToIgnore:Array<BoardPawn>):void
+ makePawnsUnselectable(arrPawns:Array<BoardPawn>):void

- makePawnSelectable(targetPawn:BoardPawn):void
- makePawnUnselectable(targetPawn:BoardPawn):void
- activateTargetPawnClickHandler(event:Event):void
- deactivateTargetPawnClickHandler(event:Event):void
- boardBlockClickHandler(event:Event):void

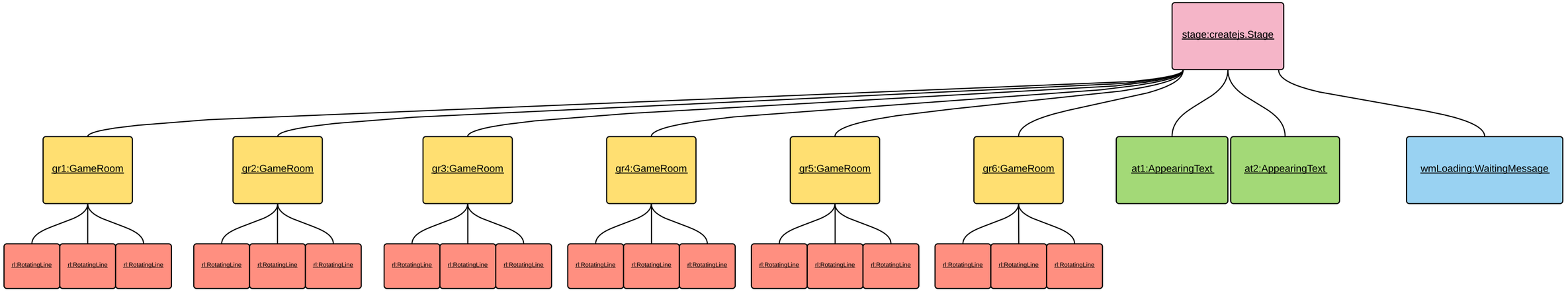
Sequence Diagram - Moving a pawn



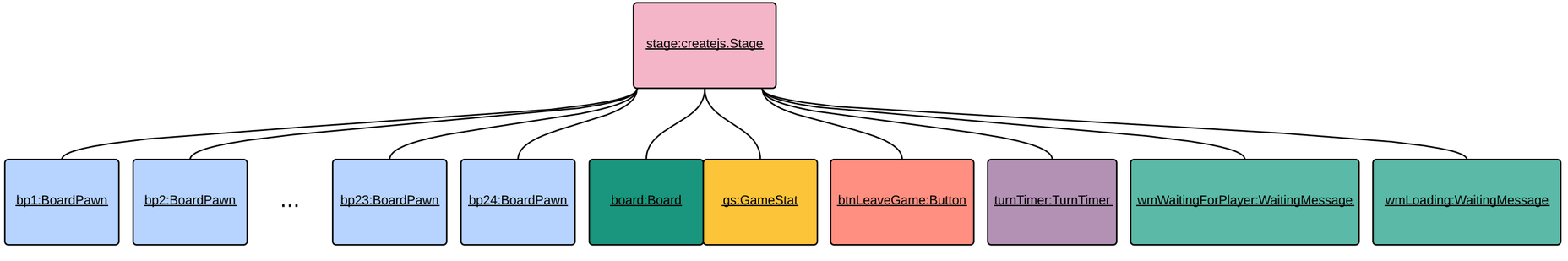
Sequence Diagram - Check if opponent is done



MENU


stage:createjs.Stage
gr1:GameRoom
gr2:GameRoom
gr3:GameRoom
gr4:GameRoom
gr5:GameRoom
gr6:GameRoom
at1:AppearingText
at2:AppearingText
wmLoading:WaitingMessage
rl:RotatingLine

GAME


stage:createjs.Stage
bp1:BoardPawn
bp2:BoardPawn
...
bp23:BoardPawn
bp24:BoardPawn
board:Board
gs:GameStat
btnLeaveGame:Button
turnTimer:TurnTimer
wmWaitingForPlayer:WaitingMessage
wmLoading:WaitingMessage