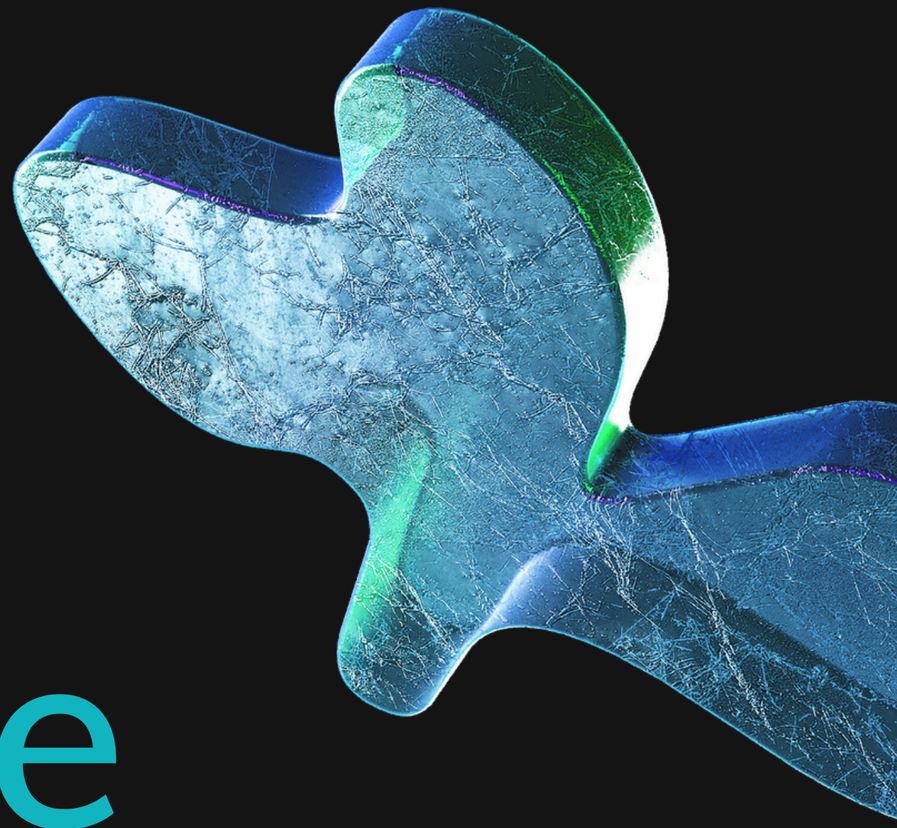
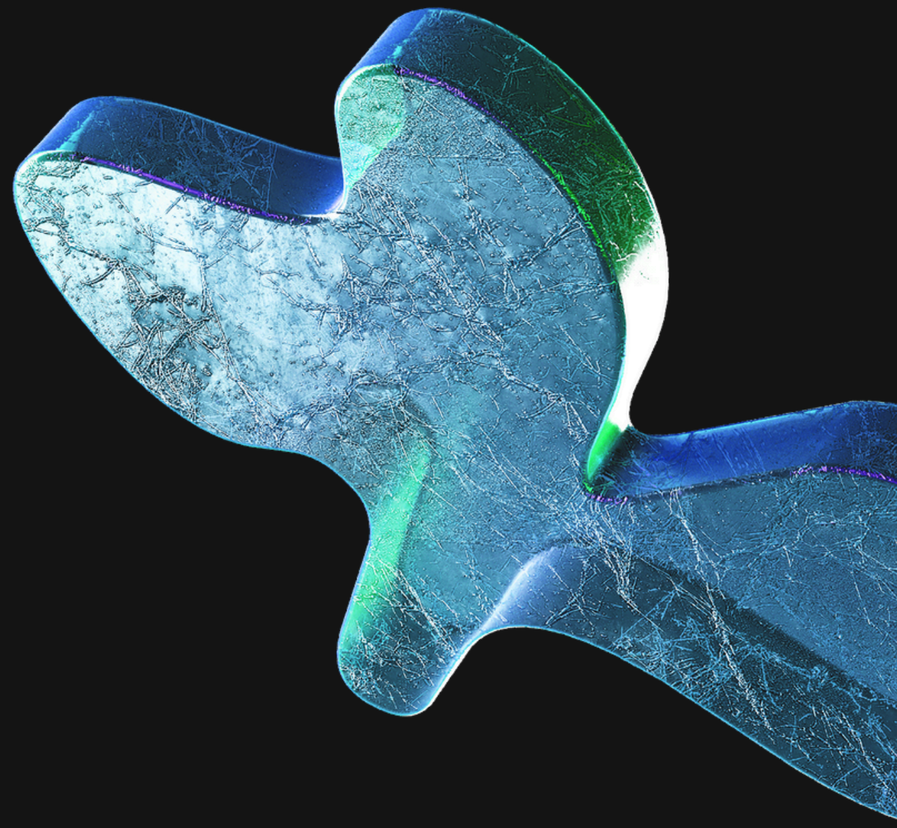
Developing Mandarin Square Game



Group members:

- Nguyễn Hữu Nam Hòa - 202240
- Nguyễn Huy Hoàng - 202241
- Lê Hoàng Huy - 20225976
- Trần Việt Anh - 20226012



Introduction

Mandarin square capturing is a fascinating traditional board game

The game offers a surprising depth of strategy, requiring players to carefully plan their moves to capture opponent's pieces.

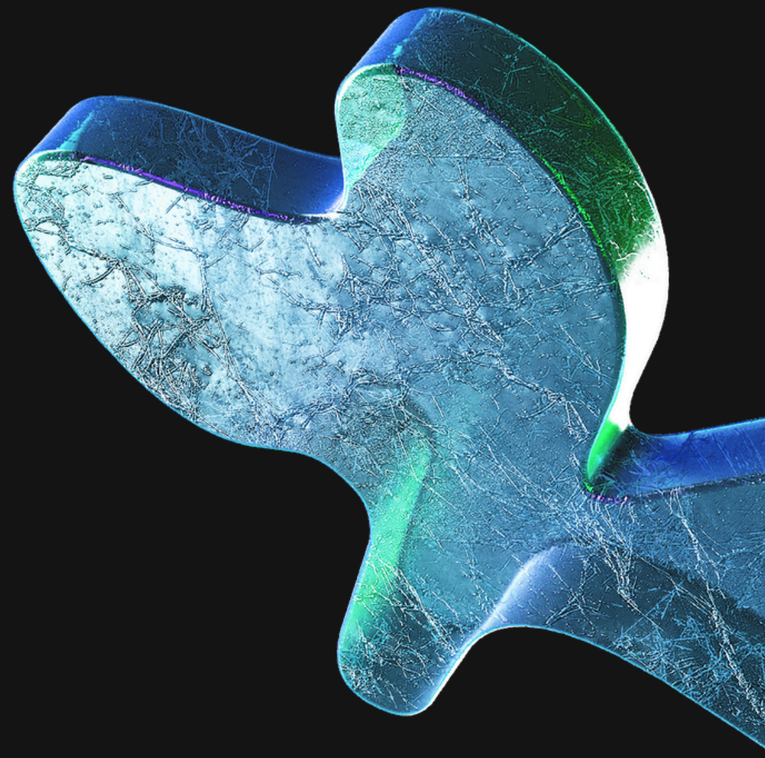
Designing a digital implementation of the game using principles of OOP. Show the power and versatility of OOP in building interactive applications.

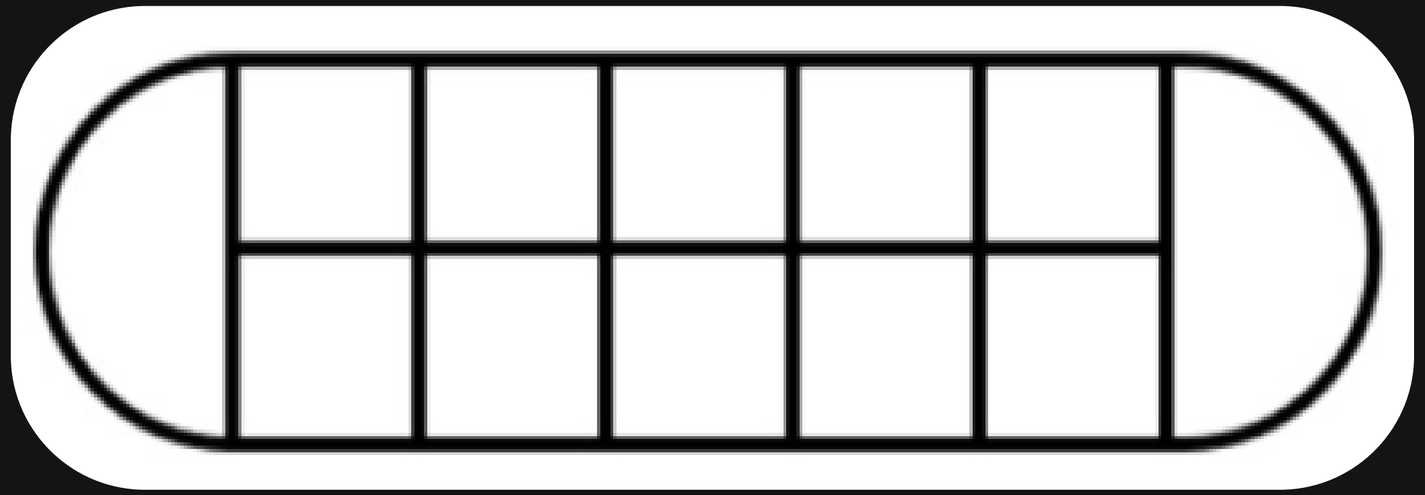
Description of Mandarin Square Capturing

Game Board	Pieces	Pieces Arrangement	Players

Game Board

- A rectangle with 10 squares - 5 on each side
- Mandarin squares: 2 semicircular squares are at the shorter ends of the rectangle
- The remaining squares are citizen squares

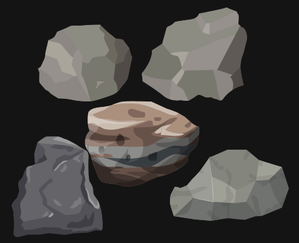




Pieces

MANDARIN PIECES

Larger pieces, 2 in total



CITIZEN PIECES

Smaller pieces, usually 50, though the number can vary

Pieces Arrangement

- Citizen pieces are placed in squares with 5 pieces per square
- Each Mandarin piece is placed in 2 semicircular squares

Player

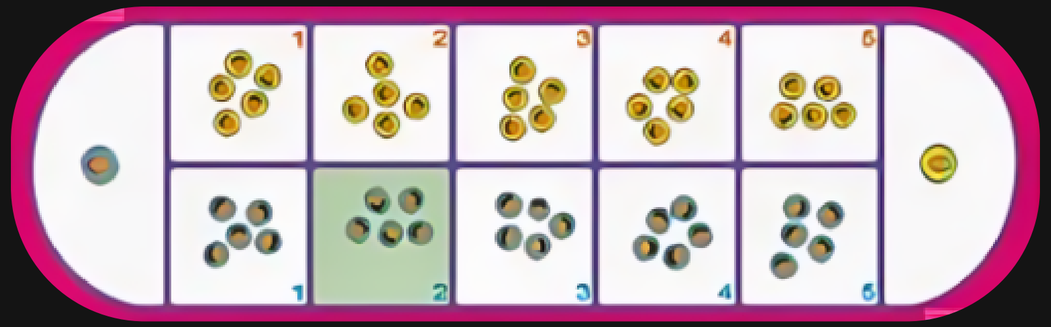
- 2 players
- Each controlling the 5 Citizen Square on each side

Game Mechanics of Mandarin Square Capturing

Set up - Scattering - Capturing - Passing - Dispatching - Winning

Set up

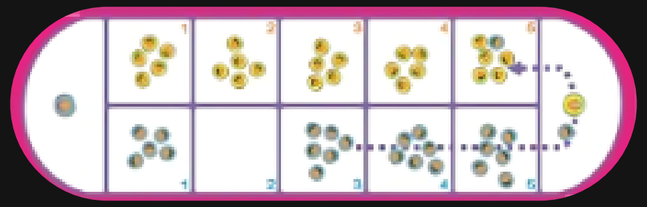
Player places 1 Mandarin piece in a Mandarin square and 5 citizen pieces in each of Citizen squares



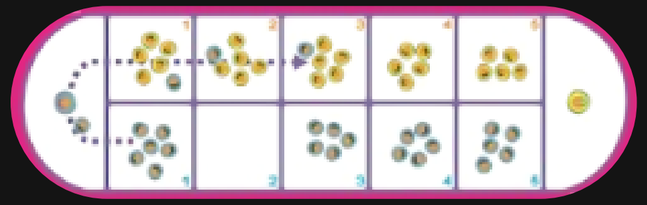
Scattering

- A player is chosen to start randomly
- Player collects all pieces from 1 of their citizen squares
- Pieces are distributed one by one to adjacent squares, in either direction
- When all pieces are distributed, player repeats by taking up pieces of the following square and distributing in the same direction

Distribute pieces to the left direction

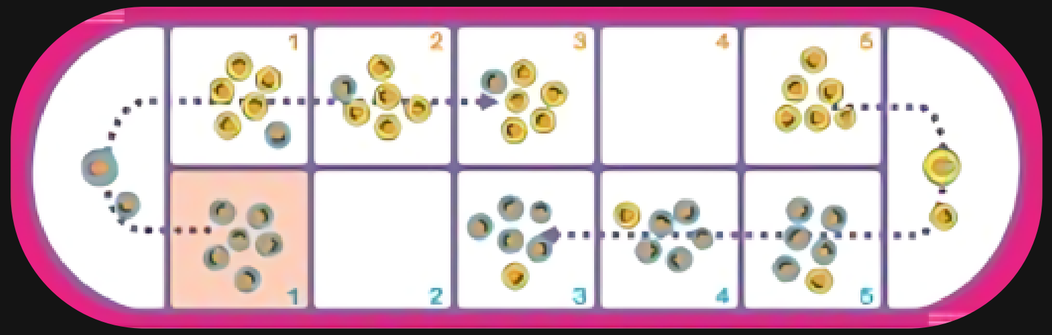


Distribute pieces to the right direction



Capturing

Players capture pieces by landing on squares following empty ones, continuing to capture if subsequent squares are also empty



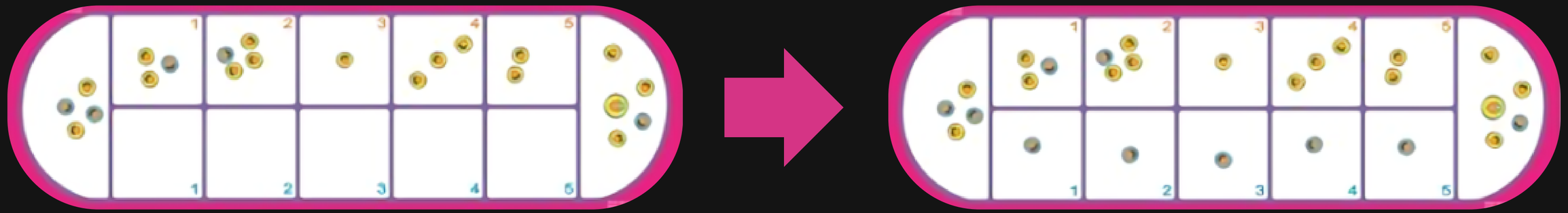
Passing

A player loses their turn if:

- Their next move lands in a Mandarin square
- Their next move encounters 2 empty squares
- They have captured all possible pieces

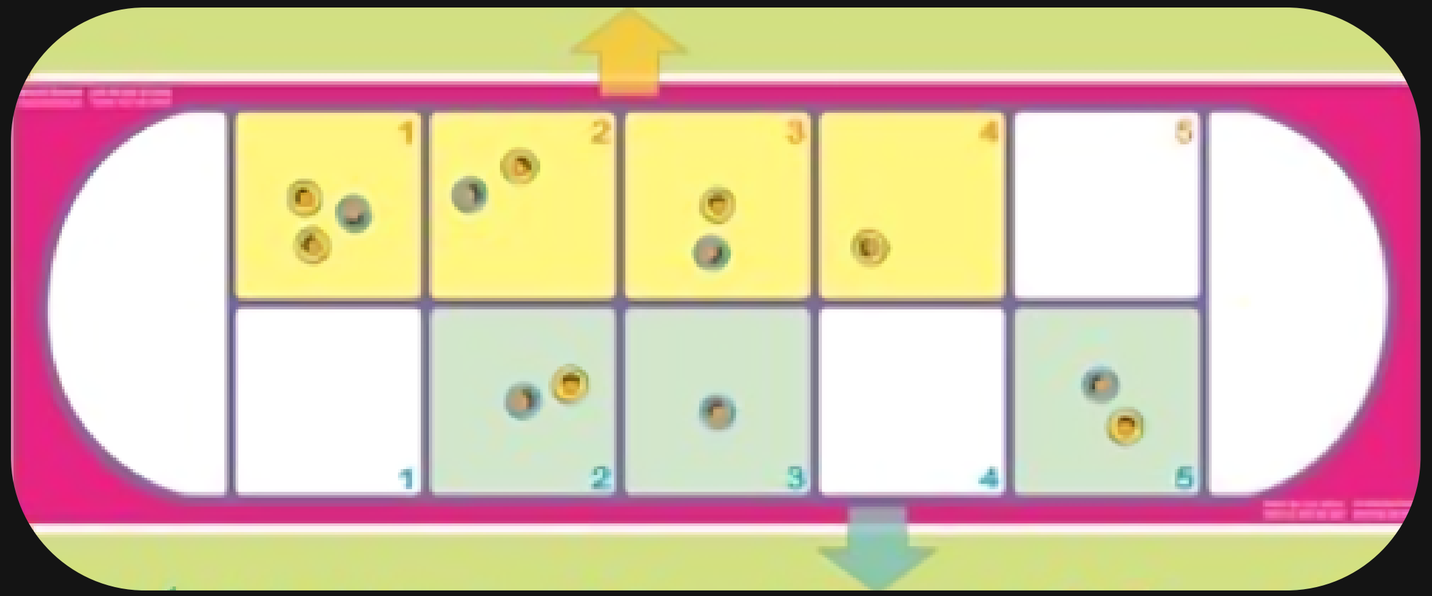
Dispatching

- Before beginning their turn, a player must use their 5 previously acquired pieces to place 1 piece in each square on their side if there isn’t a single remaining citizen piece on their side of the board
- If they are short on pieces, they have to take the remaining ones from the other player and give them back when the game is over



Winning

- The game ends when all pieces are captured
- The player with the most pieces wins
- If both Mandarin pieces are captured, the remaining citizen pieces belong to the player controlling that side



Then you say:

“Mandarin is gone, citizen dismiss, retreat”

Architecture Design

Model-View-Controller(MVC)

Model

- Manage game logic, data, and rules
- “Logic” package: Player, Board, CitizenSquare, MandarinSquare,...
- Process game logic according to user interaction given to it by GUI

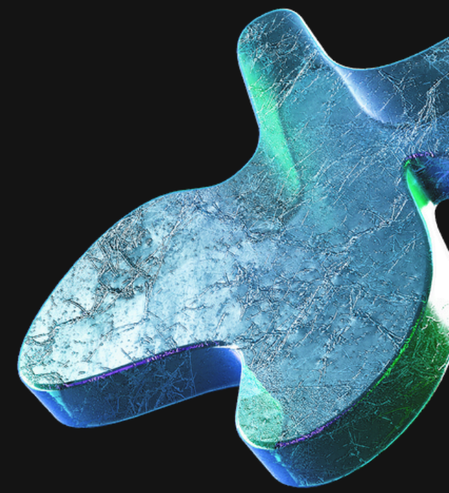
View

- Handles GUI and displays game state
- “View” model is named as GUI

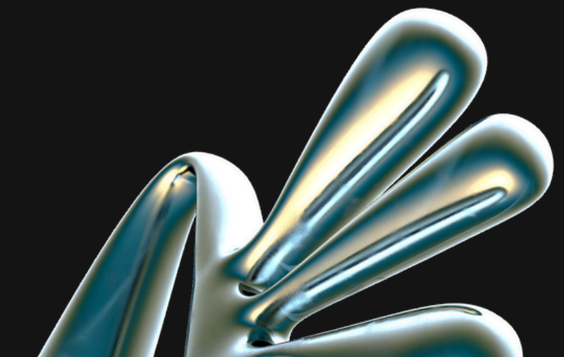
Controller

- Mediates between Model and View, processing user inputs and updating Model
- “Controller” is the Control package and is mostly implemented with Menu class
- View components take more OOP spaces than needed

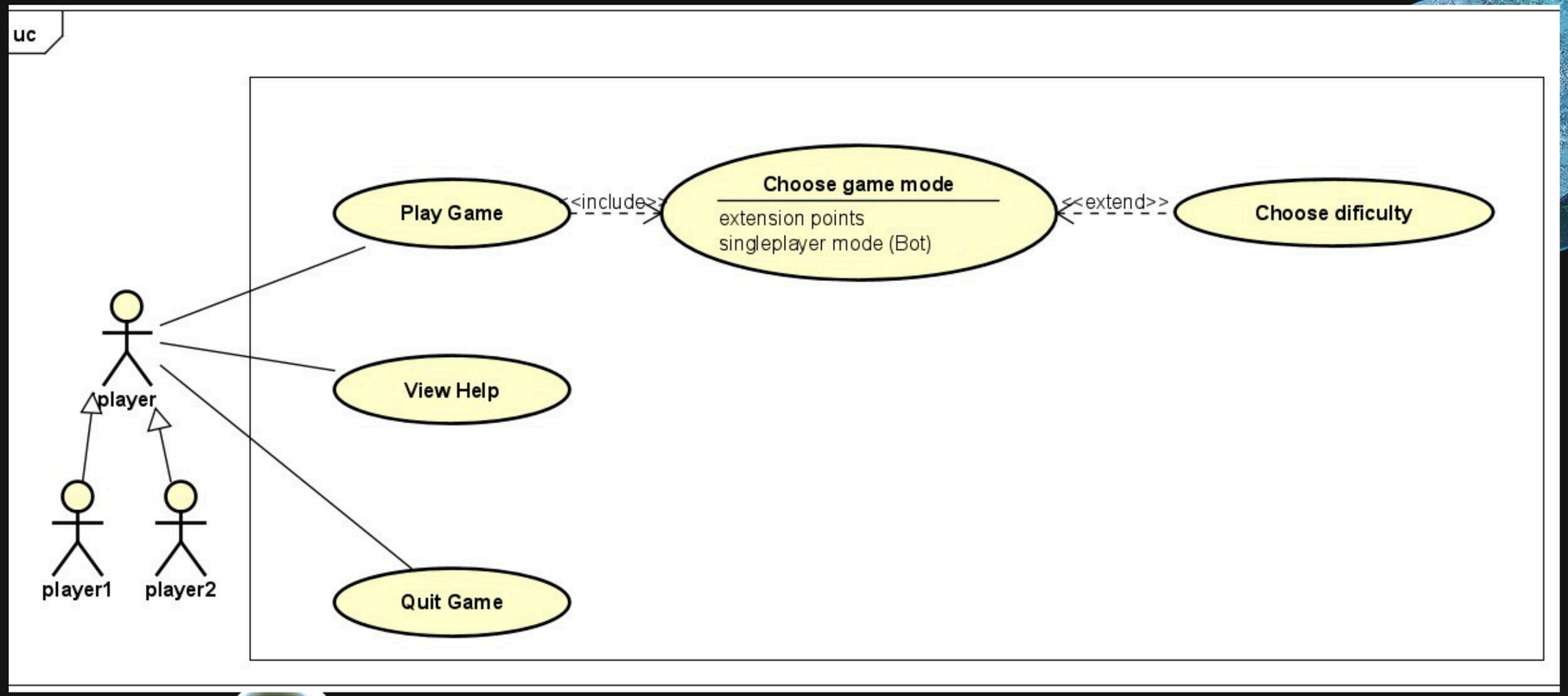
UML Design



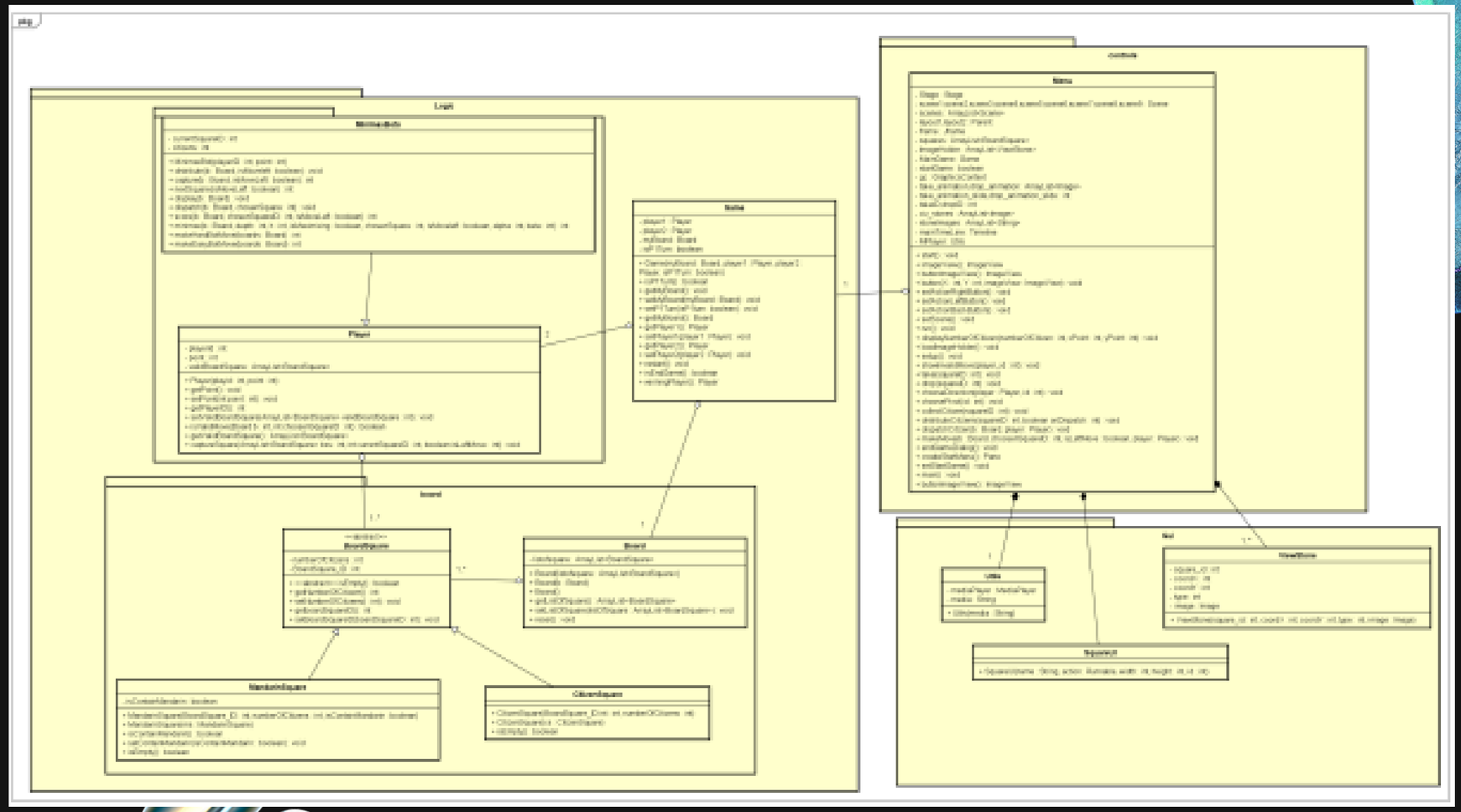
Usecase Diagram	Class Diagram	Logic Module	GUI Module	Control



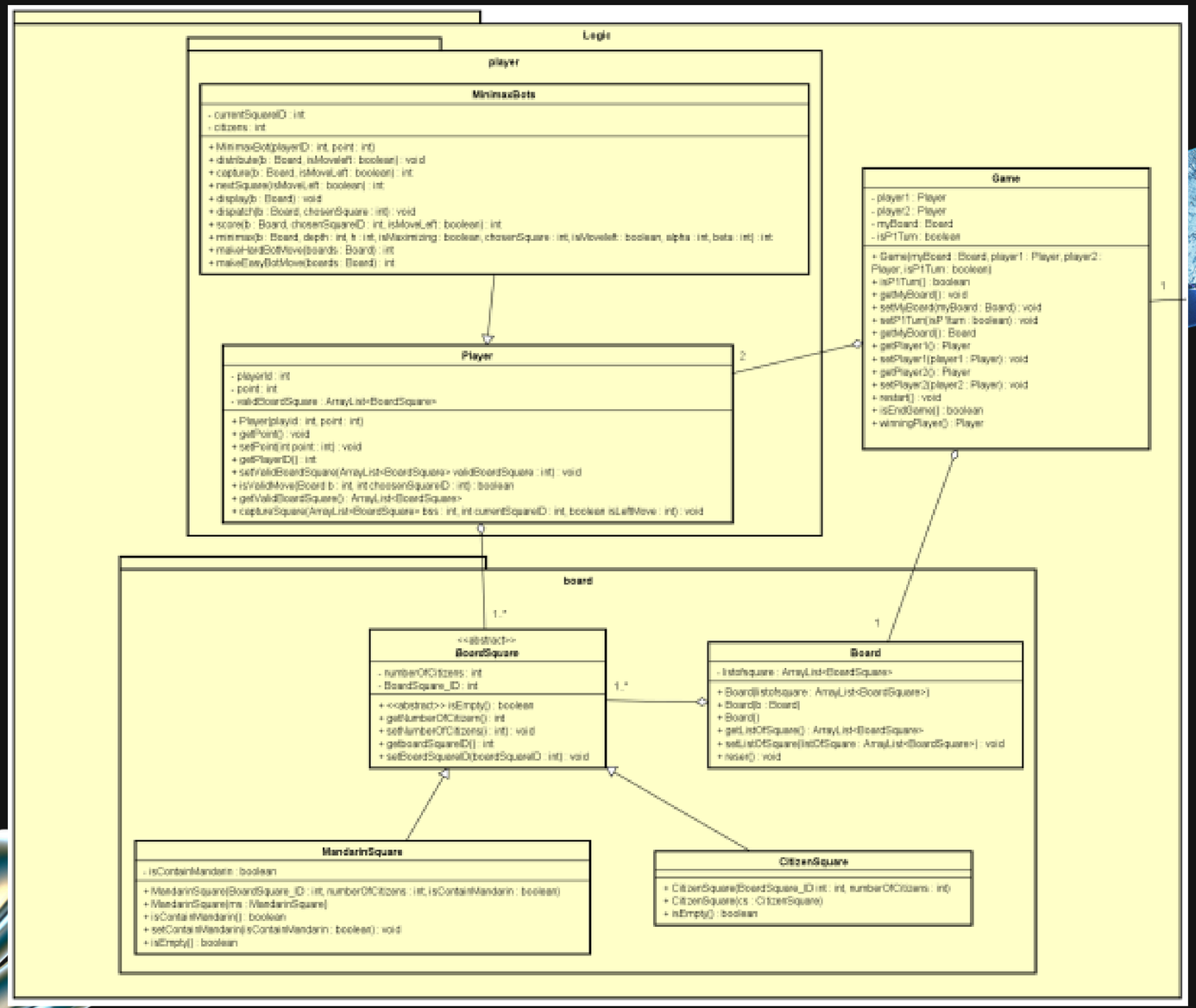
Usecase Diagram



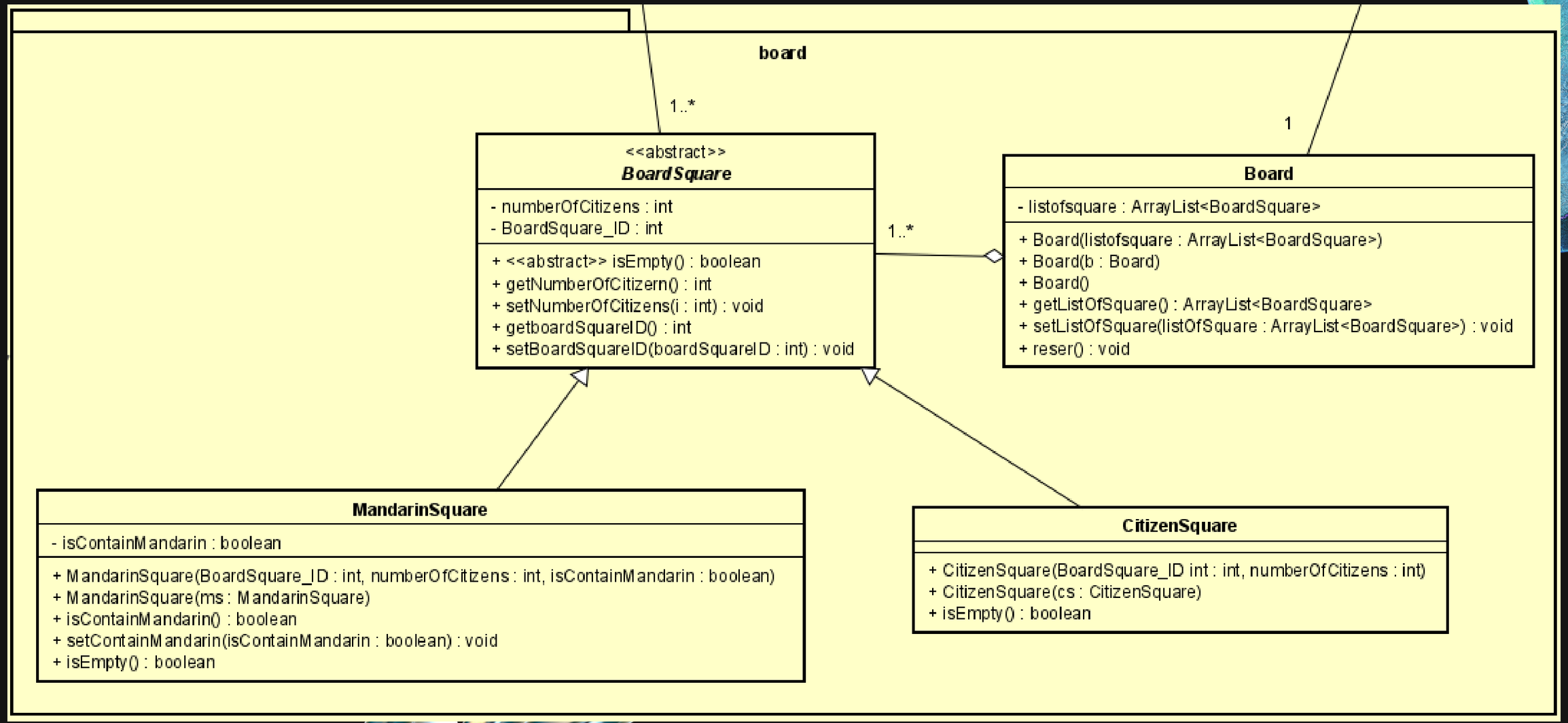
Class Diagram



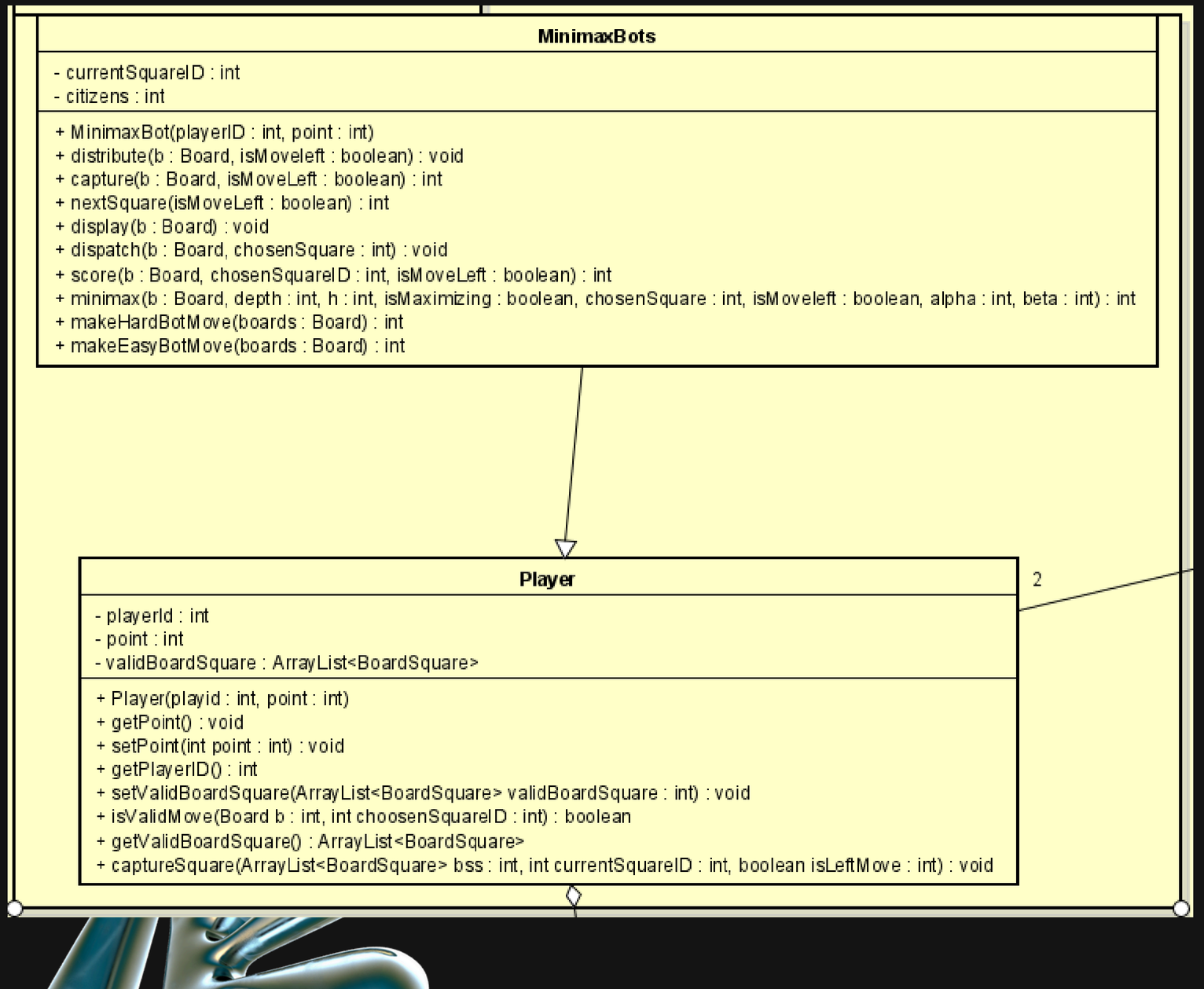
Logic Module



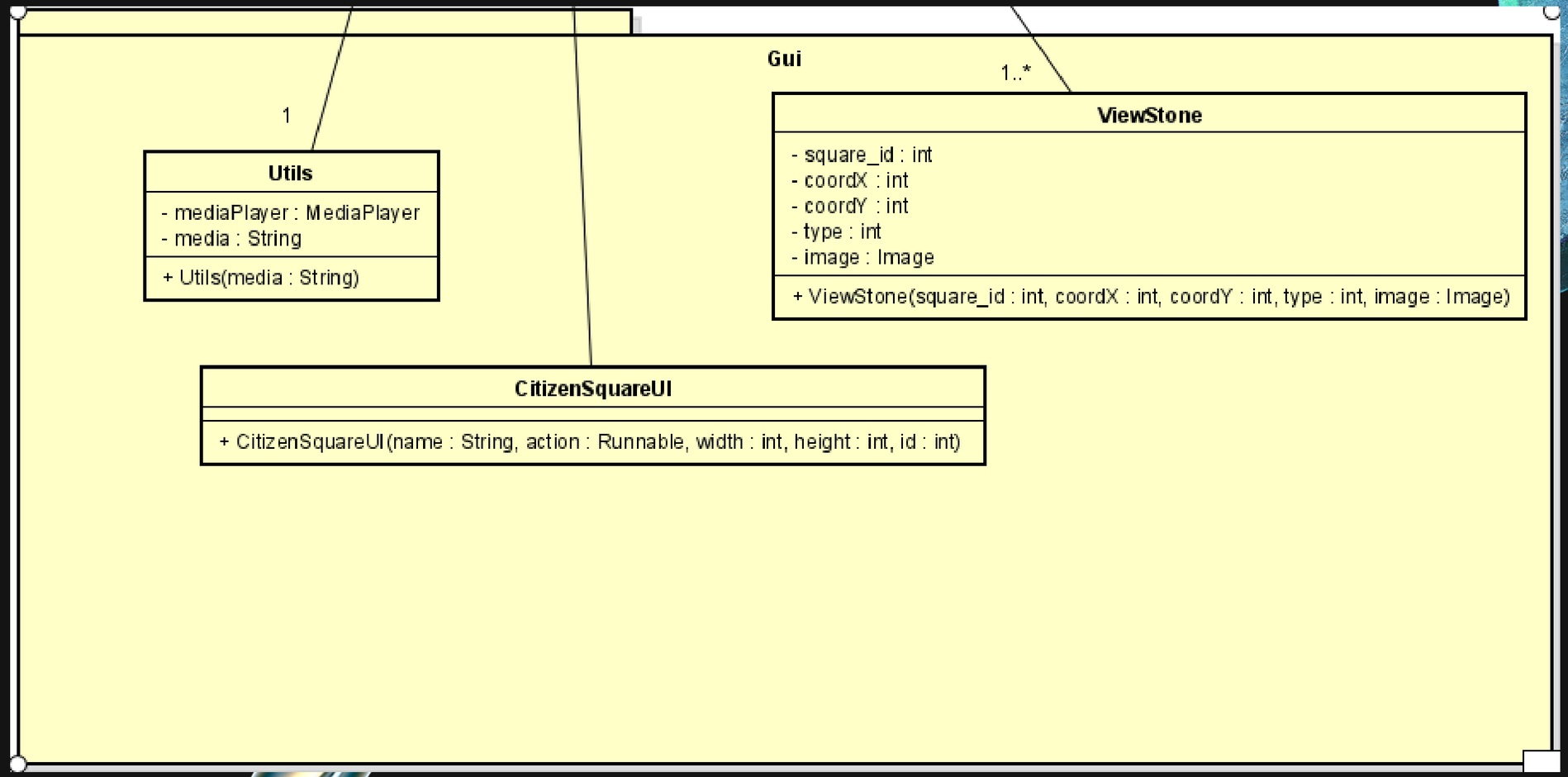
Board Package



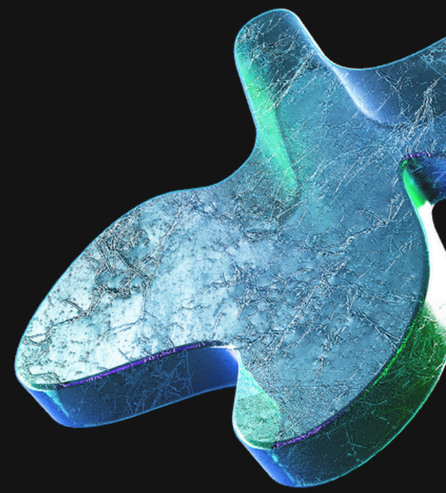
Player Package

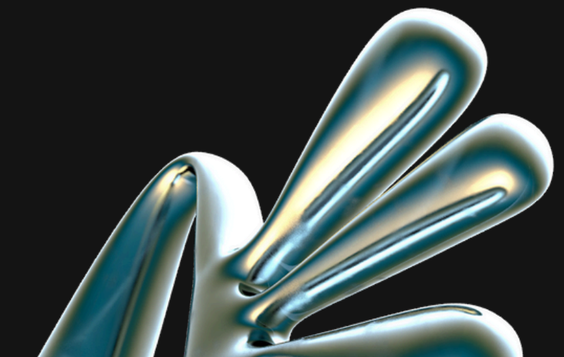


GUI Module


Gui
1
1..*
Utils
- mediaPlayer : MediaPlayer
- media : String
+ Utils(media : String)
ViewStone
- square_id : int
- coordX : int
- coordY : int
- type : int
- image : Image
+ ViewStone(square_id : int, coordX : int, coordY : int, type : int, image : Image)
CitizenSquareUI
+ CitizenSquareUI(name : String, action : Runnable, width : int, height : int, id : int)

Control





controls

Menu

- Stage : Stage
- scene1,scene2,scene3,scene4,scene5,scene6,scene7,scene8,scene9 : Scene
- scenes : ArrayList<Scene>
- layout1,layout2 : Parent
- frame : Jframe
- squares : ArrayList<BoardSquare>
- ImageHolder : ArrayList<ViewStone>
- MainGame : Game
- startGame : boolean
- gc : GraphicsContext
- take_animation,drop_animation : ArrayList<Image>
- take_animation_slide,drop_animation_slide : int
- takeID,dropID : int
- ciz_stones : ArrayList<Image>
- stoneImages : ArrayList<String>
- mainTimeLine : Timeline
- MPlayer : Utils

+ start() : void
+ imageView() : ImageView
+ buttonImageView() : ImageView
+ button(X : int, Y : int, imageView : ImageView) : void
+ setActionRightButton() : void
+ setActionLeftButton() : void
+ setActionBackButton() : void
+ setScene() : void
+ run() : void
+ displayNumberOfCitizen(numberOfCitizen : int, xPoint : int, yPoint : int) : void
+ loadImageHolder() : void
+ setup() : void
+ showInvalidMove(player_id : int) : void
+ take(squareID : int) : void
+ drop(squareID : int) : void
+ chooseDirection(player : Player, id : int) : void
+ choosePivot(id : int) : void
+ collectCitizen(squareID : int) : void
+ distributeCitizens(squareID : int, boolean onDispatch : int) : void
+ dispatchCitizen(b : Board, player : Player) : void
+ makeMove(b : Board, choosenSquareID : int, isLeftMove : boolean, player : Player) : void
+ endGameDialog() : void
+ createStartMenu() : Pane
+ setStartGame() : void
+ main() : void
+ buttonImageView() : ImageView

Object-Oriented Design

BoardSquare

- boardSquareID
- numberOfCitizens
- isEmpty()

MandarinSquare

- inherited from BoardSquare
- isContainMandarin

Player

- player_id
- points
- validBoardSquare
- captureSquare()

Menu

- Stage
- gc
- ImageHolder
- mainTimeLine
- start()
- setup()
- choosePivot
- chooseDirection
- collectCitizen
- distributeCitizen

CitizenSquare

- inherited from BoardSquare

Board

- listOfSquare
- reset()

MinimaxBot

- inherited from Player
- minimax()
- makeHardBotMove()
- makeEasyBotMove()

Game

- myBoard
- player1, player2
- isP1Turn
- Game(Board myBoard, Player player1, Player player2, boolean isP1Turn)
- Getters, Setters
- restart()
- isEndGame()
- winningPlayer()

BOT game

Minimax algorithm

- Maximizing heuristic value translates to minimizing opponent's heuristic
- Minimax algorithm works by constructing a game tree - this tree represents all possible future states after each move by both players, each node in tree represents a specific state

1. **Maximizing Player**: minimax explores all possible moves, then evaluating each resulting position using **heuristic evaluation function -** choose the move leading to the **highest evaluated score**
2. **Minimizing Player:** minimax assumes opponent will play optimally to minimize their score; it explores all possible moves and choose the one resulting in the **lowest evaluated score**

Limitations:

- Exponential complexity
- Heuristic Evaluation Function

BOT game

Minimax with Alpha-Beta Pruning

Alpha-beta pruning is an optimization technique that significantly reduces the number of nodes explored in the game trees

1. **Alpha(α):** represents the highest guaranteed score the maximizing player can achieve from a particular node onwards; any branches leading to scores lower than alpha can be pruned
2. **Beta(β):** represents the lowest guaranteed score the maximizing player can achieve from a particular node onwards; any branches leading to scores higher than alpha can be pruned

Max Nodes: alpha is updated with the maximum score encountered along a branch; if a score exceeds the current beta, the entire branch can be pruned

Min Nodes: beta is updated with the minimum score encountered along a branch; if a score falls below the current alpha, the entire branch can be pruned

By dynamically adjusting alpha and beta values, alpha-beta pruning eliminates unnecessary evaluations

Demo gameplay