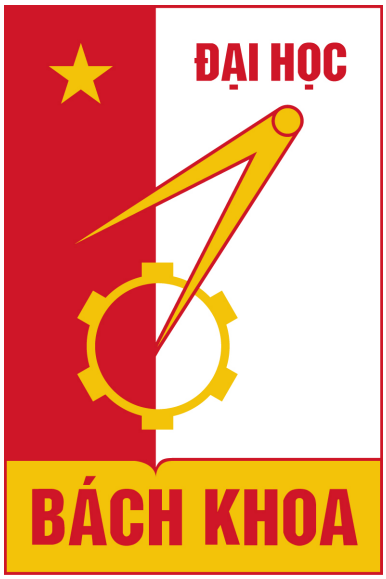
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

——————— o0o ———————



Mandarin Square Capturing Game

Object-oriented Programming

Lecturer : Tran The Hung

Students : Nguyen Huy Hoang - 20226041
Nguyen Huu Nam Hoa - 20226040
Le Hoang Huy - 20225976
Tran Viet Anh - 20226012

Class : ICT-01 K67

Hanoi - 2024

Contents

1 Team Members

Our team consists of four members, each given specific task with different completion level. This is our honest and unanimous assessment with each of the team member. We strive to make unbiased assumptions and review of each other work so that the assessment can be as accurate as possible

Members	Student ID	Tasks	Completion level	Assessment
Nguyen Huu Nam Hoa	20226040			
Nguyen Huy Hoang	20226041			
Tran Viet Anh	20226012			
Le Hoang Huy	20225976			

2 Introduction

Mandarin square capturing, also known as Ô ăn quan in Vietnamese, is a fascinating traditional children's board game with origins dating back centuries. Beyond its simple rules and materials, the game offers a surprising depth of strategy, requiring players to carefully plan their moves to capture their opponent's pieces and ultimately achieve victory. This report delves into the world of Mandarin Square Capturing by designing a digital implementation of the game using the principles of Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) is a powerful programming paradigm that allows for the creation of reusable and modular code by focusing on real-world entities and their interactions. By applying OOP principles to Mandarin Square Capturing, we can create a well-structured and maintainable digital representation of the game. This report will explore how various OOP concepts, such as classes, objects, methods, and inheritance, can be effectively utilized to model the game board, pieces, player actions, and overall game logic.

This project not only serves as a way to preserve and share this traditional Vietnamese game but also provides an opportunity to showcase the power and versatility of OOP in building interactive applications. Throughout this report, we will detail the design process, outlining the classes and their functionalities, the relationships between them, and how they collectively recreate the strategic essence of Mandarin Square Capturing in the digital realm.

3 Game Description

Game board: The game board can be made on the ground, a sidewalk, or a flat piece of wood. Its size is variable as long as it can be divided into enough squares to accommodate the pieces without being too big to make movement of the pieces difficult. After being drawn as a rectangle, the game board is divided into 10 squares, with five symmetrical squares on each side. Draw two squares that are semicircular or arc-shaped and oriented outward on the rectangle's two shorter sides. The two semicircular or arc-shaped squares are known as Mandarin squares, and the other squares are known as citizen squares.

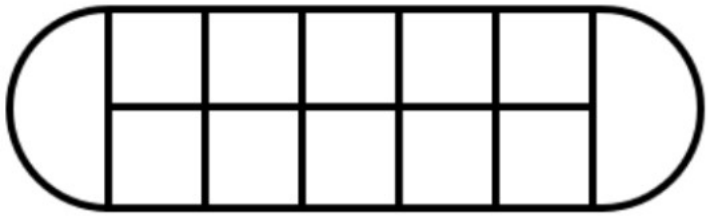


Figure 1: Game board

Pieces: Mandarin pieces and citizen pieces. They are composed of a variety of materials and have stable shapes and moderate sizes, allowing players to hold and grasp multiple pieces with one hand while playing. Crucially, the Mandarin pieces are notably bigger than the citizen pieces so that they may be quickly identified apart. The pieces can be made artificially from hard materials, most typically plastic, or they can be pebbles, bricks, stones, or the seeds of some fruits. According to the game's regulations, the citizen pieces can have any number of pieces, however the most popular quantity is 50. The number of Mandarin pieces is always 2.

Pieces arrangement: the citizen pieces are arranged in squares with five pieces per square, while the Mandarin pieces are arranged in two semicircular or arc-shaped squares, one Mandarin piece per square. We can substitute a Mandarin piece by adding the same number of citizens pieces to the Mandarin square if you don't want one or can't locate one that suits.

Players: Usually two, each on the outside of the board's longer side. The player on that side controls the squares on that side.

4 Game Mechanics

4.1 Setup

Each player places one big stone (named the "Mandarin piece" and it is equal to five or ten citizen pieces) in the Mandarin square and five small stones (named the "citizen pieces") in each of the rice field squares.

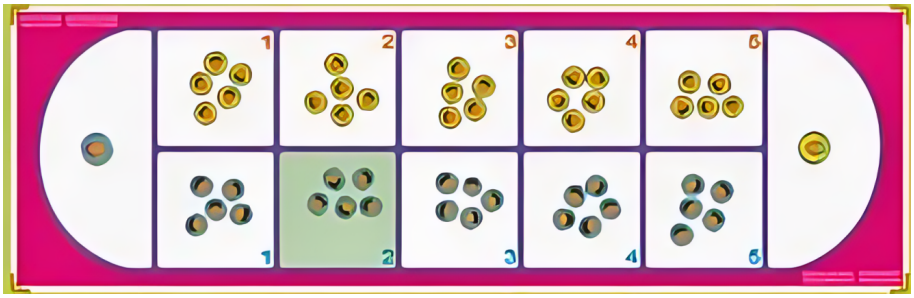


Figure 2: Game setup

4.2 Scattering

1. The first player will be chosen randomly
2. The first player takes up all the pieces of any citizen square on his/her side of the board

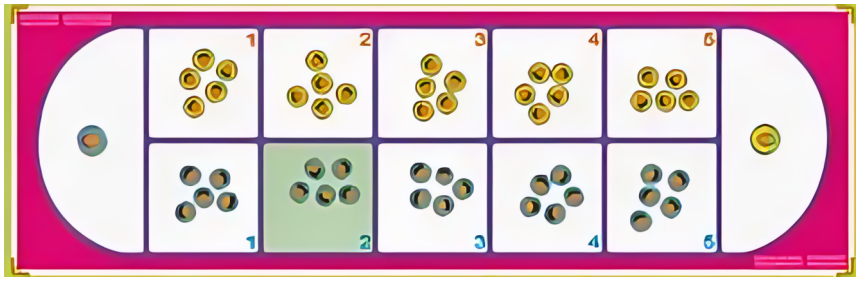
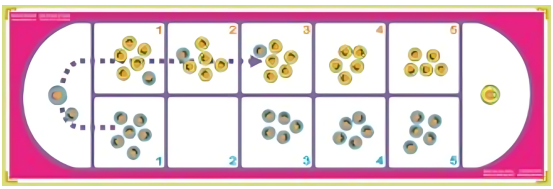
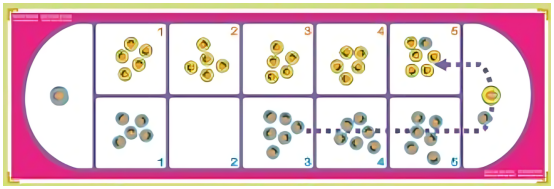


Figure 3: First player takes turn

3. Distributes one piece for each square, starting at the next square in either direction
4. When all pieces are distributed, the player repeats by taking up the pieces of the following square and distributing them in the same direction.



(a) Player distribute pieces to the left direction



(b) Player distribute pieces to the right direction

Figure 4: Player distributes pieces

4.3 Capturing

5. When the next square to be distributed is empty, the player wins all the pieces in the square after that and take them out of the table.

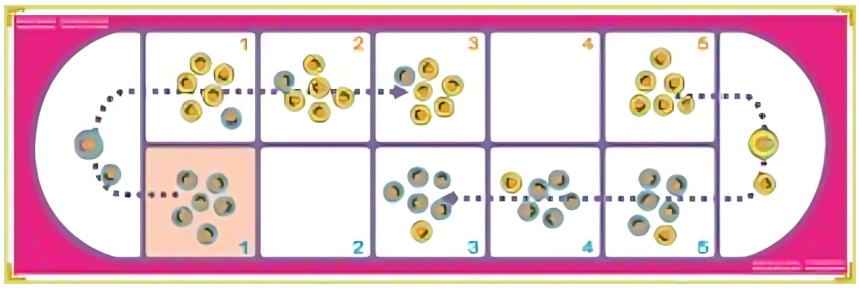


Figure 5: Capture

* When the square next to the won square is empty, the player also wins the all the pieces in the square after that

4.4 Passing

You will lose your turn and the other player can start in the following case:

- Your next distribution is the Mandarin Square
- Your next distribution turn are two empty squares
- After winning all available squares

4.5 Dispatching

Before beginning their turn, a player must use their five previously acquired pieces to place one 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 in order to count the points.)

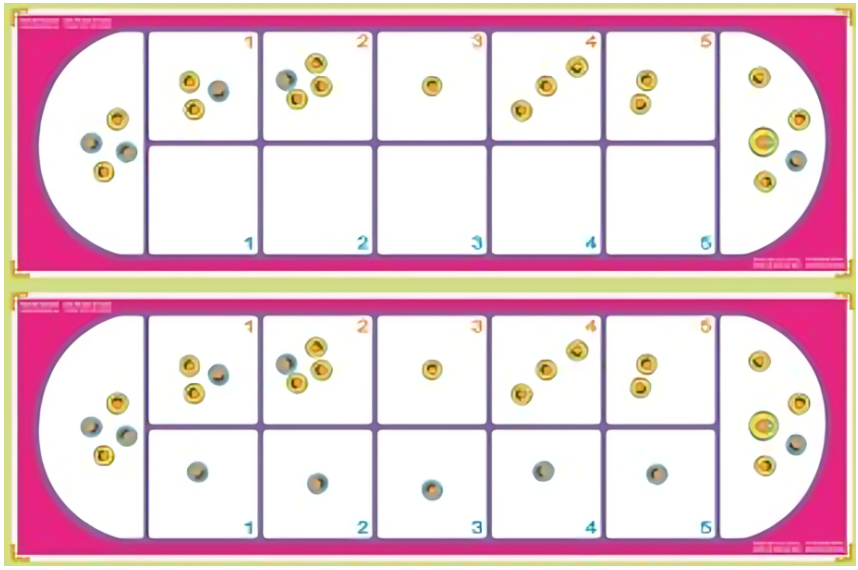


Figure 6: Player places pieces after before starting his/her turn

4.6 Winning

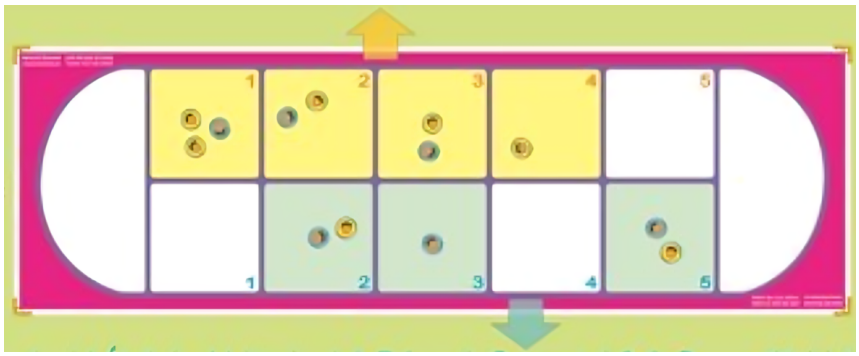


Figure 7: Game over

The game ends when all the pieces are captured. Whichever player has more pieces is the winner or if both Mandarin pieces are captured, the remaining citizen pieces belong to the player controlling the side that these pieces are on. Then you say:

"***hết quan, tàn dân, kéo về***"
("Mandarin is gone, citizen dismisses, retreat")

5 Architectural design

MVC Framework The Model-View-Controller (MVC) framework is a powerful and versatile architectural pattern used in software development to separate concerns and enhance the modularity of applications.
As indicated by its name, the MVC framework contains three main components: Model, View and Controller

1. **"Model"** plays the role of governing the underlying rule and business logic of the application. It handles all the processing and implementing the functions and features of the application designed by the programmer. We can also view Model as the "Brain" of the application that handles all the information and thinking for the main Body, that is the rest of the application. Most of this processing will happen in the background and will provide no actual confirmation for the user and will instead rely on the GUI to display its results for the user. In the context of our game, the "Model" has a different name that is a "Logic" package. Here it stores the Java file of all the necessary logic components of the game including Player, Board, CitizenSquare, MandarinSquare and many other. Its main role is to process the game Logic according to user interaction given to it by the GUI

2. **"View"** is mostly known as the GUI, however, it is much more complex than that. View contains every component that deals with displaying or rendering art and media to the screen and give the user visual response of the application. View does not handles any logic related to the game. Its main focus lies on displaying the results of the game logic or handling the interaction of User with the GUI, which then feeds it back to the Model via the Controller. In the aspect of games, one key feature is the frame rate. Within our game, we have implemented this with the idea of Timeline , which is a javafx class. This Timeline allows us to control the speed at which the GUI refresh itself to update new frames. This updating speed is so fast that the user do not notice it and perceive the frames as continuous instead of discrete frames being updated after a specified amount of time. In our game the View model is simply named as GUI due to its main role as the Graphic Interface.

3. **"Controller"** plays the role of the intermediary between the View and Model. Specifically it can be view as an interface acting as the one to handle the interaction between View and Model. When the Model has finished processing of the logic and is ready to show results to GUI, Controller will make call to View component to display the corresponding results. In contrast, when View detects interaction from the user with the GUI and need to invoke the underlying logic of the application to handle the user's request, the Controller will receive and pass on the call accordingly to the Model component. Within our game, this "Controller" is mostly implemented with the Menu class, which handles the call from GUI to Model. However, for the duration of our development, we notice that the View component can take up more

spaces than needed. Therefore we decided to implement the update of frames with Menu for ease of bug fixing and speeding up the development process. In our game the "Controller" is the Control package which contains the Menu class.

The MVC framework's separation of concerns allows us to work on individual components independently, promoting cleaner code, easier maintenance, and improved scalability. This decoupling enables a more organized approach to handling complex applications, facilitating parallel development, and making it simpler to manage changes over time.

6 UML Design

6.1 Usecase diagram

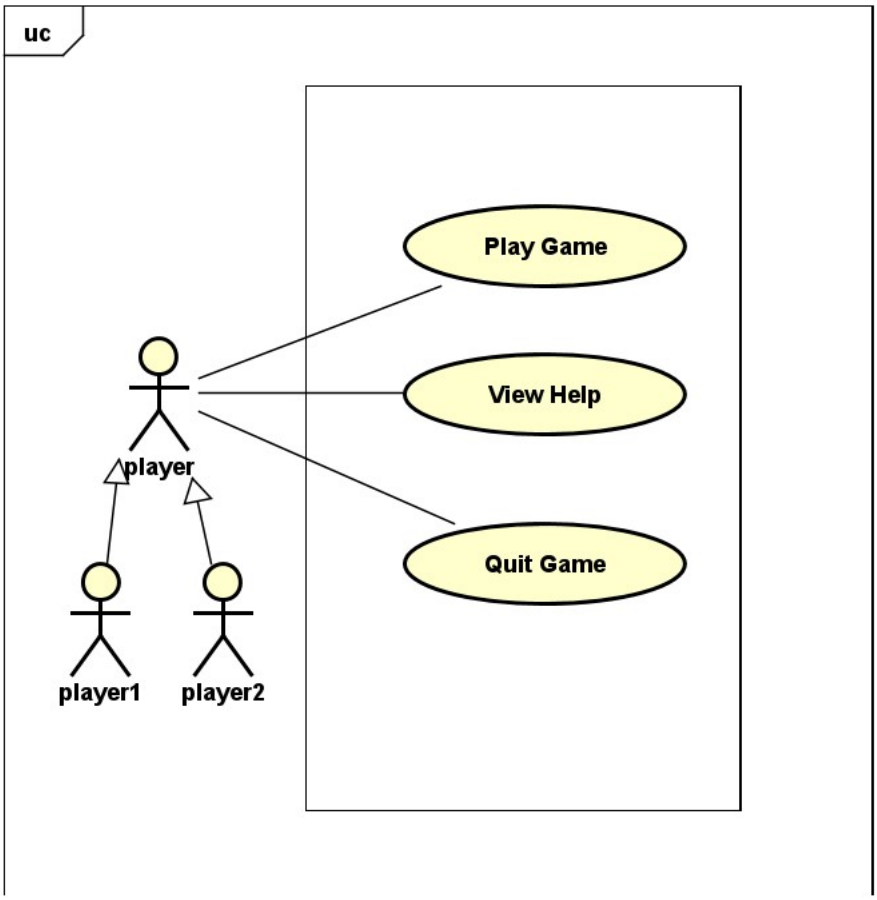


Figure 8: Usecase diagram

In terms of our Use-case diagram, we have identify three main needs of the user for our game

- First of all is to start a new game, whether it is multiplayer or singleplayer
- The second use-case is to view the help documents to learn more about how to play the game
- Lastly, it is to exit the game when the player no longer wants to continue the game.

6.2 Class diagram

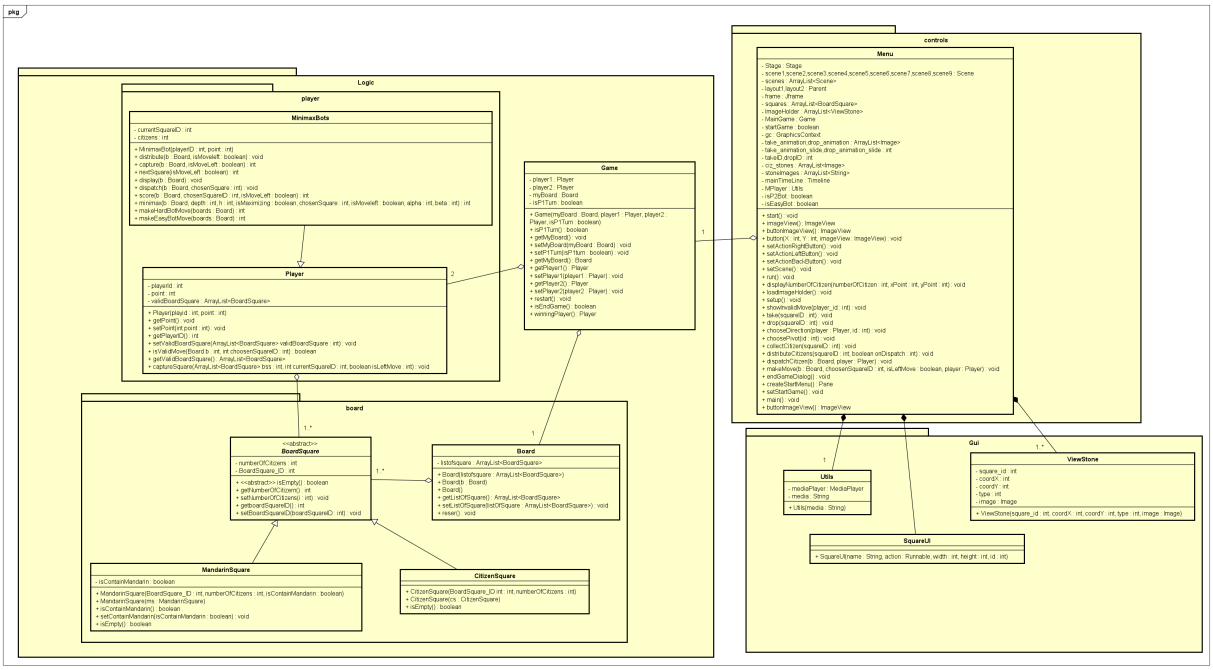


Figure 9: Class diagram

This is our Class diagram which showcase three main modules of our game. They are Logic, GUI and Control which translate to the three main components of MVC architecture. These modules combine to create a complete loop of interaction from player to game logic and vice versa.

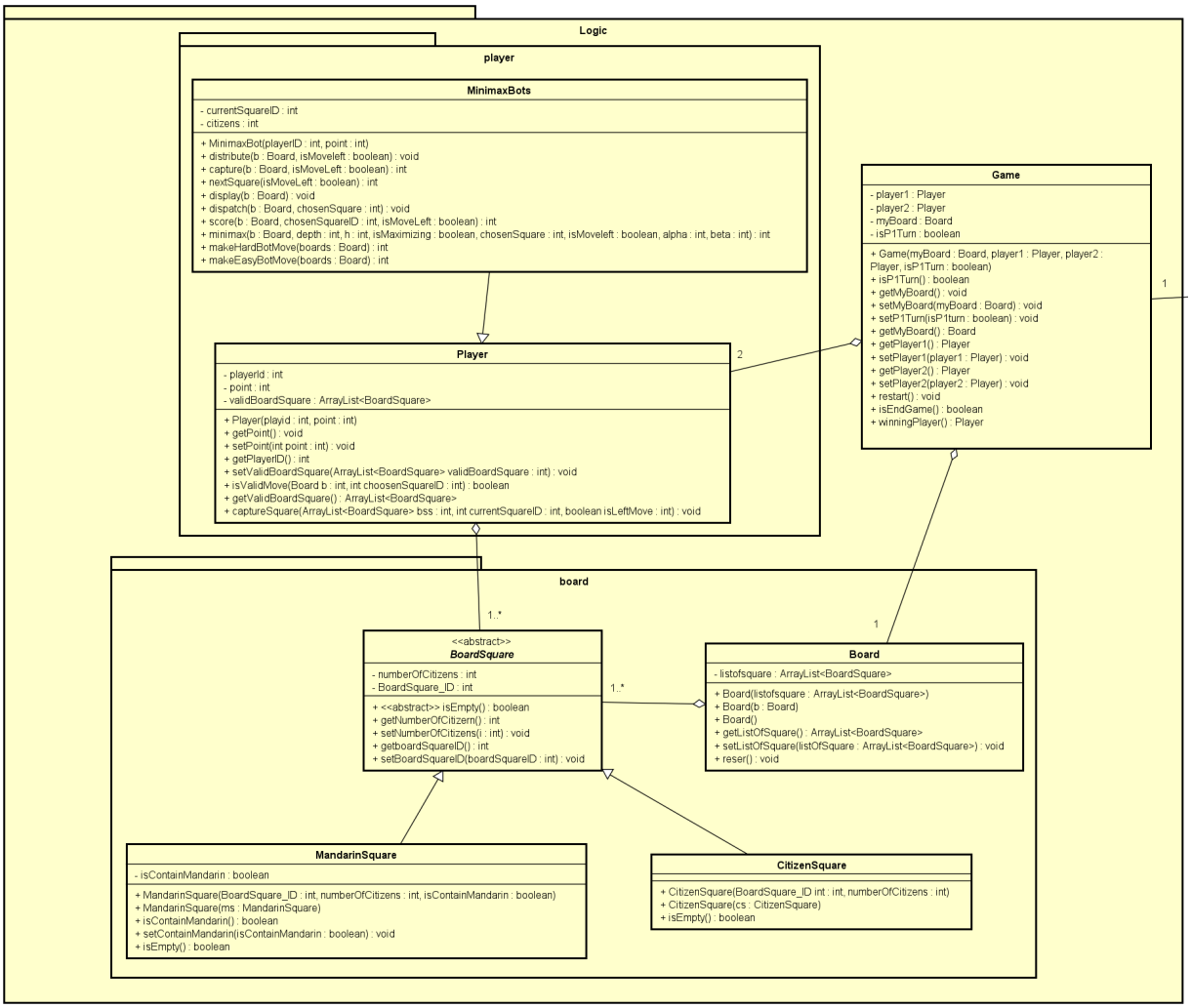


Figure 10: Logic module

The first module, Logic, contains the most important classes that handles the game Logic. These are: Game, Player, MinimaxBot, Board, CitizenSquare and MandarinSquare. If you recall the game mechanics of the game , our design follows the same concept and implements each of the component of the real game into our video game. The package Logic is , however, reorganized into two subpackage “board” and “player” , and a Game class

Here our implementation also display some characteristics of OOP which is **Inheritance** and **Abstraction**.

6.3 Logic module

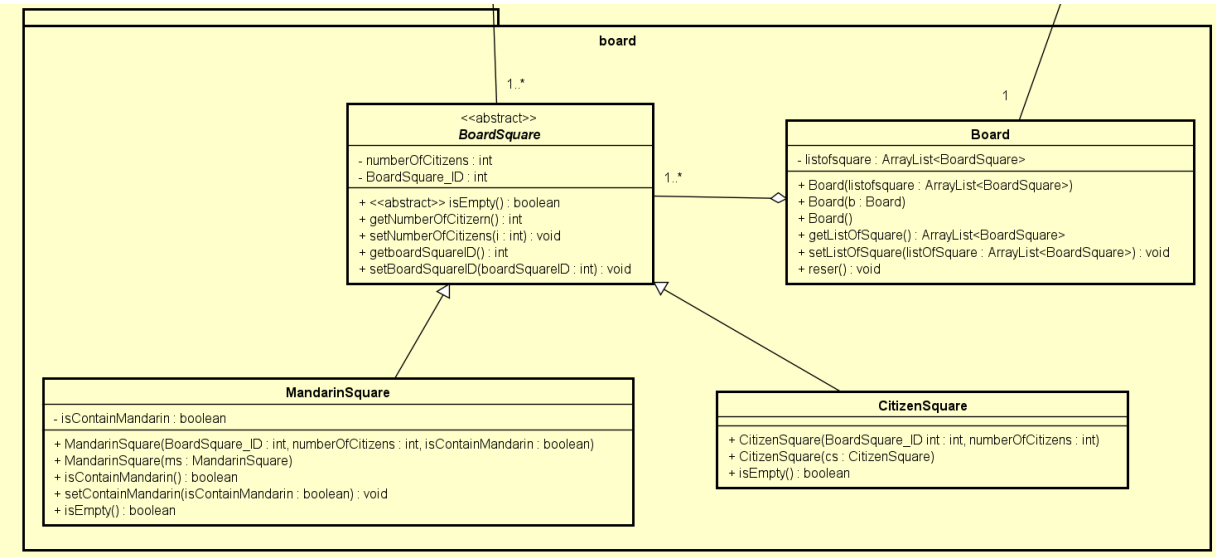


Figure 11: Board package

To be more specific, in terms of **Abstraction** our BoardSquare class is an abstract class which serves as the blueprint for the two class MandarinSquare and CitizenSquare. This can be understood as both Mandarin squares and Citizen squares in the real game are all considered playable squares and share some similar attributes, while only differ in roles with regards to the rule of the game.

- **Board** class is the representation of the playing board. It has one main attribute which is the list of squares belong to it and an important method which is "reset" to help re-organize the stones in the square to start a new game
- **BoardSquare** is the parent class of MandarinSquare and CitizenSquare, it is quite simple , only containing getter and setter.
- **MandarinSquare** and **CitizenSquare** are the inherited class of BoardSquare. CitizenSquare facilitate direct usage of user and each is assigned to each player while MandarinSquare do not.

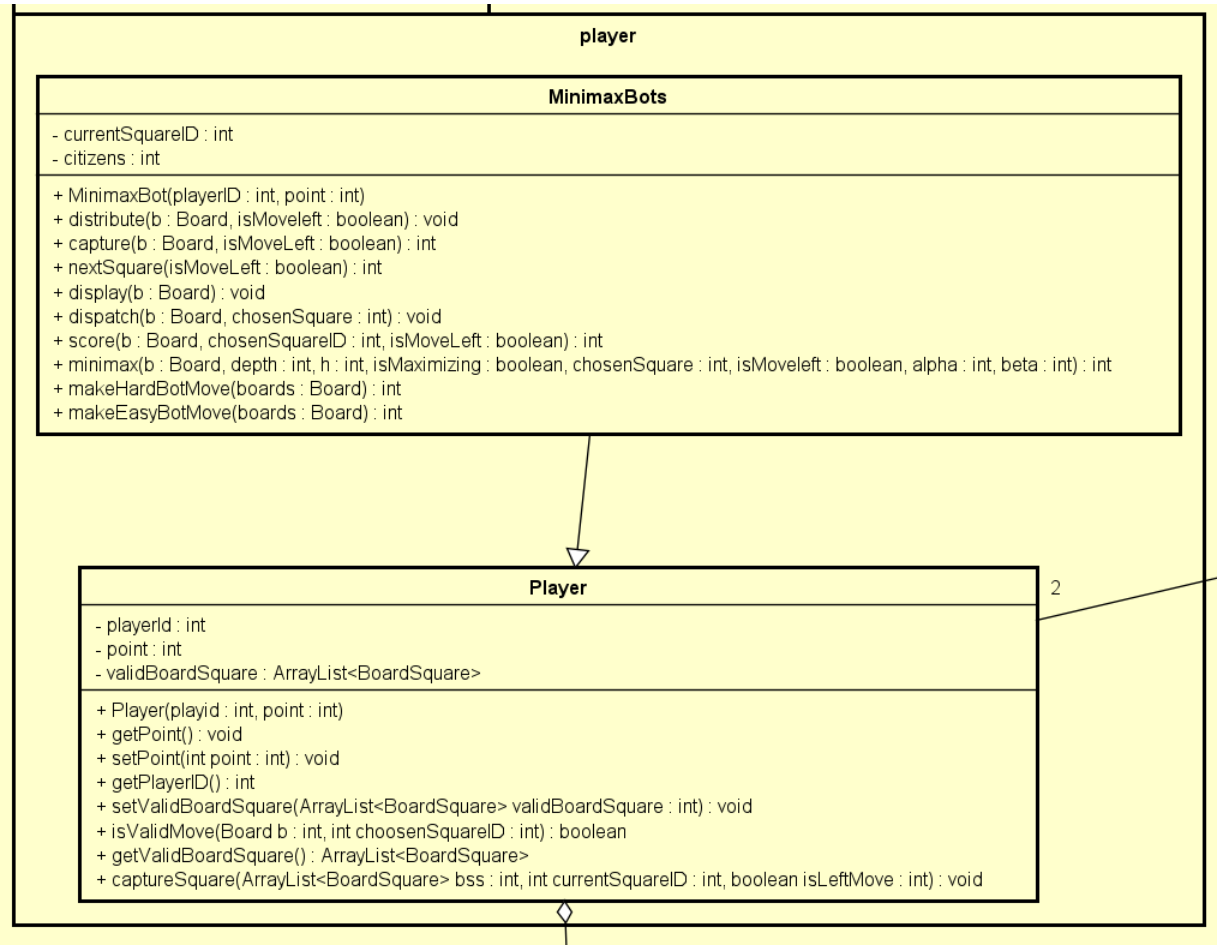


Figure 12: Player package

On the other hand, in terms of **Inheritance**, MinimaxBot inherit from the Player class. This can be understood as the Bot also plays the similar role of a second player. And the difference is in the processing and decision making of the Bot. In our game, we implement the Bot with Minimax algorithm and this requires our game to process its every move, which resembles the same way humans make decisions.

6.4 GUI module

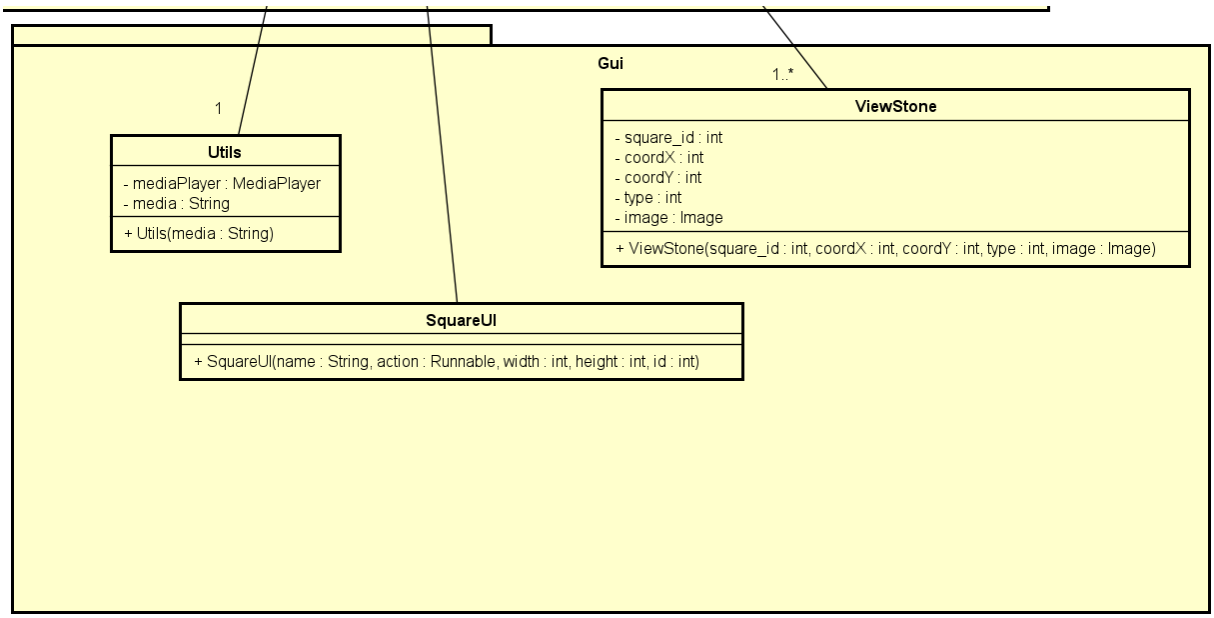


Figure 13: Control module

The second module, GUI, contains the graphic component necessary for the interaction with the user. This contains the Graphical interface of the Board which displays points and react to Mouseclick interaction from the Player. ViewStone class plays the role of graphical representation of the stones. The Utils play the role of adding soundtrack onto the game and keeping it running in parallel.

6.5 Control

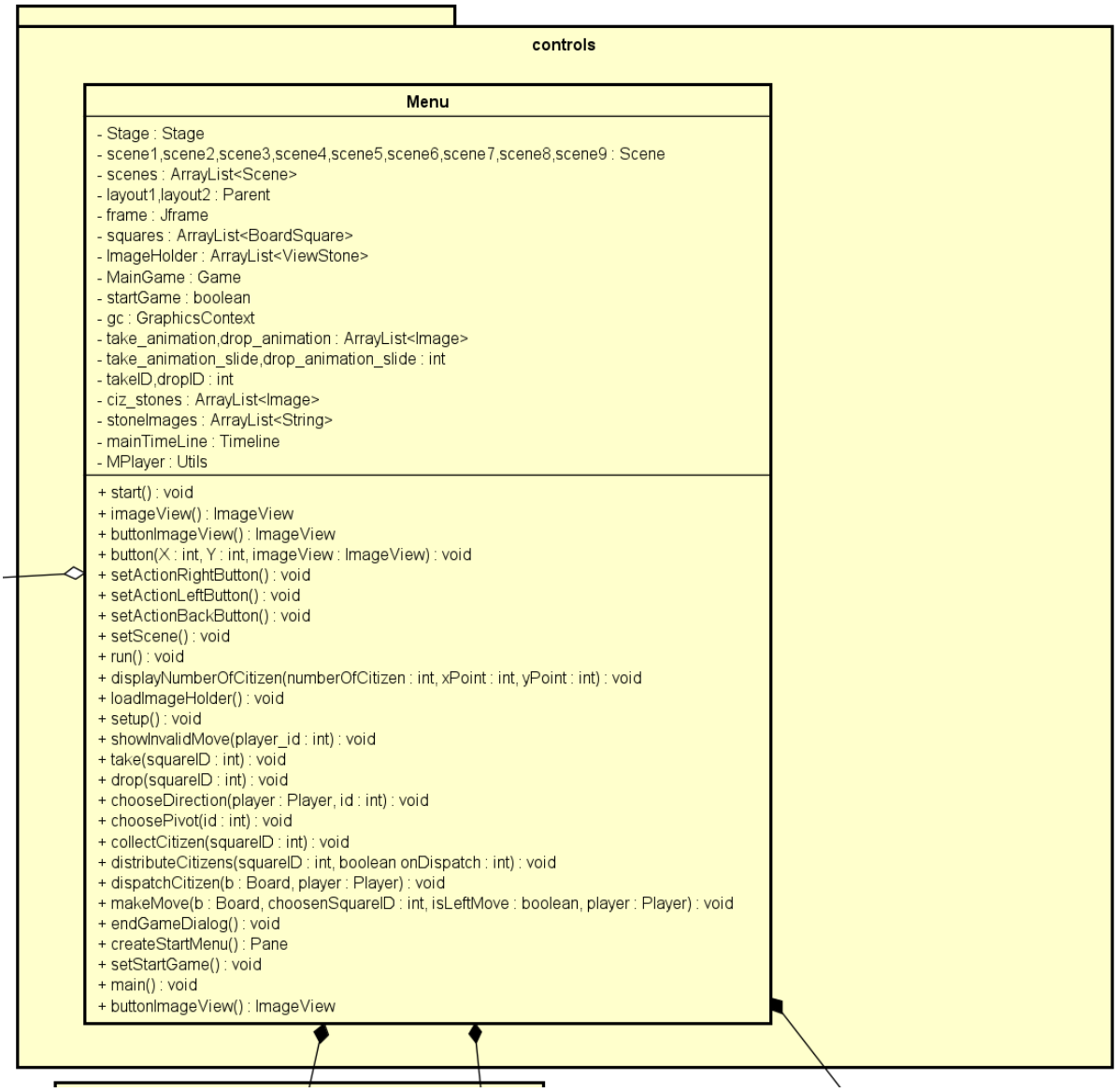


Figure 14: Control module

Last but not least, Control, contains the most important class that help facilitate the connection between Logic and GUI, the Menu class. This Menu class will handle many different part of the Game, including reacting to user interaction from GUI and then invoking the corresponding method from Logic module to process the response. Another important task of the Menu class is to update the frame after every specific amount of time. This means the game screen is updated frequently and create the illusion of continuous interaction.

7 Object-Oriented Design

BoardSquare

This *abstract* class represents a general square of the game, either a Citizen square or a Mandarin square. It hold some information of the general square (board square ID, number of citizens) and the state of the board square (empty)
Here are the atributes and methods of this class:

- **boardSquareID**: the board square ID
- **numberOfCitizens**: the citizens in the board square
- **isEmpty()**: checks if a square is empty

CitizenSquare

This class represents a specific citizen square. It is inherited from the BoardSquare class. Therefore, it has the same atributes and methods of the BoardSquare class.

MandarinSquare

This class represents a specific Mandarin square. It comes from the BoardSquare class as well. It features several extra attributes to hold certain unique properties in addition to the attributes and methods it inherited from the BoardSquare class.
Here is the additional atributes of this square:

- **isContainMandarin**: checks if a Mandarin square still contains Mandarin

Board

This class represents of the game board itself. it hold information about the board layout (number of squares)
Attributes and methods:

- **listOfSquare**: the list of squares of the board
- **reset()**: reset the board itself

Player

This class represents a player of the game. It hold some information about the player (player ID, points, valid square) and a method of a player (capture square) Attributes and methods:

- **player_id**: player id

- **points**: points that a player gets
- **validBoardSquare**: a list of valid board squares that a player can control **captureSquare()**: capture available squares after distributing citizen pieces

MinimaxBot

This class represents a bot using *minimax* algorithm. It is inherited from Player class. Attributes and methods

- **minimax()**: calculate the heuristic score of a square.
- **makeHardBotMove()**: choose the square that has the worst heuristic score
- **makeEasyBotMove()**: choose the square that has the best heuristic score

Game

This class, named Game, represents the core logic of a two-player board game. Here's a breakdown of the code:

Attributes:

- **myBoard**: a game board.
- **player1** and **player2**: the two players in the game.
- **isP1Turn**: indicating whose turn it is (true for player 1, false for player 2).

Methods:

- **Game(Board myBoard, Player player1, Player player2, boolean isP1Turn)**: initializes the game with a board, two players, and the starting player turn.
- **Getters and Setters**
- **restart()**: restart the game
- **isEndGame()**: checks for the end game condition
- **winningPlayer()**: determines and returns the winning player

Menu

This is perhaps one of our most complex class. Attributes:

- **Stage**: the main Stage to set up graphical component on top of it. This is a standard way to render game art assets and provide visual representation of the entities in a game
- **gc**: a GraphicsContext attribute, which plays the role of rendering and updating the frames after each specified amount of time. This helps the game to update its game screen and provide real time interaction with player
- **ImageHolder**: this is an array which holds the list of the images of stone that has been rendered out to the screen. We want to keep this list so that the images cannot be changed from frame to frame, except when collected or distributed explicitly by the game
- **mainTimeLine**: this attribute regulates the amount of time the frames can be updated. Furthermore we can also call the method "mainTimeline.stop()" or "mainTimeline.play()" to either cease the refreshing or start it. This is implemented because we want to make sure we do not pre-render real-time game assets in prior to save memory space for Java

Methods:

- **start()**: this method is a method of the Application class. It sets up the necessary assets and functions for the game to run. The main() method in javafx is only used to invoke this start() functions which will set up and set the application going.
- **setup()**: this method setup all the Game's instances such as Player, Game, Square, as well as art assets such as the images of Stones, Squares and other visual indicator
- **choosePivot**: this function creates the response of the application with the MouseClick interaction of the player when they choose a square to collect the citizens. One key notice is that the player here can be either the real user or the bot.
- **chooseDirection**: this function is used when the interaction is from a real player. In this case , a dialog box pops up to confirm the direction that the user wants to distribute the stones to.
- **collectCitizen**: perform the collecting citizens process for selected squares and invoke the call to the GUI function to delete the images of the collected stones
- **distributeCitize**: perform adding new stones to citizen squares and invoke the call to render those stones.

8 BOT game

8.1 Minimax algorithm

Minimax is a fundamental algorithm in game theory, particularly for two-player zero-sum games like Mandarin Square Capturing. A zero-sum game implies that one player's gain is directly equivalent to the other player's loss. In this game, maximizing heuristic value translates to minimizing the opponent's heuristic value, as the ultimate goal is to win.

The minimax algorithm works by constructing a game tree. This tree represents all possible future states of the game after each legal move by both players. Each node in the tree represents a specific board state.

1. **Maximizing Player**: At maximizer's turn (represented by a "max" node), minimax explores all possible legal moves. It then evaluates each resulting position using a *heuristic evaluation function* [1]. The algorithm chooses the move that leads to the highest evaluated score.

2. **Minimizing Player**: At your minimizer's turn (represented by a "min" node), minimax assumes their opponent will play optimally to minimize their score. It explores all their legal moves and chooses the one that results in the lowest evaluated score for them (effectively maximizing their score).

Limitation of Minimax algorithm:

- **Exponential Complexity**: The number of possible game states grows exponentially with each move. A full minimax search becomes computationally expensive for deep evaluations.

- **Heuristic Evaluation Function**: The accuracy of minimax heavily relies on the heuristic function used to evaluate board states. The more accuracy heuristic score is, the better minimax algorithm is.

[1] In this game, the heuristic evaluation function of the minimax algorithm is based on the difference between the points of the maximizer and the minimizer. The maximizer will try to maximize the difference and vice versa.

8.2 Minimax with Alpha-Beta Pruning

Building upon the minimax algorithm, alpha-beta pruning is an optimization technique that significantly reduces the number of nodes explored in the game tree. This allows for deeper searches within the allotted time, leading to more strategic moves by the game bot.

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, as they cannot possibly influence the final decision.

2. **Beta (β)**: Represents the lowest guaranteed score the minimizing player can achieve from a particular node onwards. Any branches leading to scores higher than beta can be pruned, as the opponent wouldn't choose a move resulting in a worse outcome.

During the minimax search:

1. **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 as it cannot affect the final decision.

2. **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, focusing the search on the most promising branches. This allows the game bot to explore deeper into the game tree, considering more future possibilities and leading to stronger play.

9 Technology Used

- **Astah**: creating UML diagrams
- **JavaFX** and **Swing**: build GUI
- **Java Coding Convention**: Classes, packages, variable names, functions are name according to standards.
- **Git & Github**: Manage source code via Github, see source code