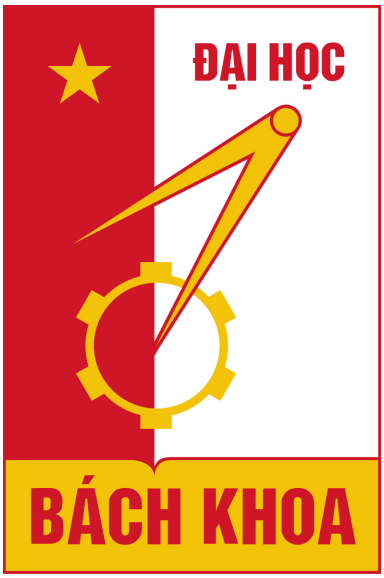
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



OBJECT-ORIENTED PROGRAMMING - IT3100E

CAPSTONE PROJECT: Traditional game "Ô Ăn Quan"

Instructor: Ph.D. Tran The Hung
Class: 147839
Students: Phung Duc Dat-20226025
Ha Minh Duc-20226027
Nguyen Trung Kien-20226052

Hanoi, June-2024

Contents

1 Assign of member

Team Member	Responsibilities
Nguyen Trung Kien (Leader, 34%)	• Frontend Development: Collaborate on UI design, complex components, performance optimization. • Backend Development: Collaborate on game logic, complex algorithms, special case handling. • Project Leadership: Coordinate tasks, manage timelines, facilitate communication, ensure overall project success. • Quality Assurance: Thoroughly test the game, identify and fix bugs/issues.
Phung Duc Dat (33%)	• Frontend Development: Design and implement JavaFX UI (game board, pieces, buttons, score display, interactive elements). • UI Enhancements: Collaborate on animations, visual effects. • Presentation: Design and create project presentation slides.
Ha Minh Duc (33%)	• Backend Development: Implement game logic (rules, scoring, turn management, win/loss conditions). • Data Management: Design and handle data storage (game states, scores, etc.). • UML Diagrams: Create UML diagrams (class, sequence, etc.) to visualize system structure.

2 Project description

2.1 Project overview

"Ô Ăn Quan" is a two-player that consists of one board with 10 squares, divided into 2 rows, and 2 half-circles on the two ends of the board. Initially, each square has 5 small gems, and each half-circle has one big gem. Each small gem equals 1 point and each big gem equals 5 points. The first player to start the game is chosen randomly.

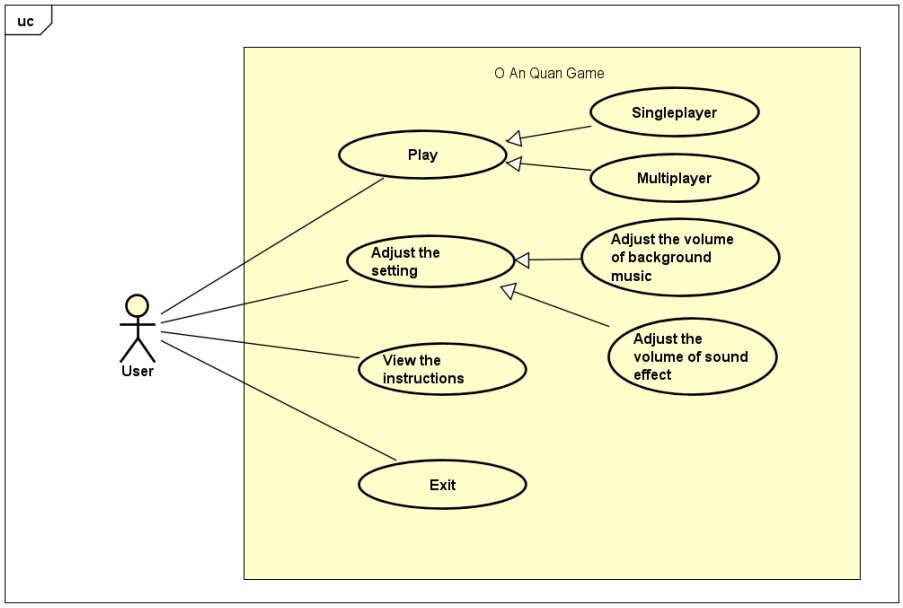
Each player possesses 5 squares on their side and can start their turn from any of those squares, pick up all gems from this square and spread them in either direction: clockwise or counterclockwise. The player must drop one gem into every cell on the path and continue until there is no gem left in their hand. Starting turns from any of the half circles is prohibited.

After spreading all gems in hand, the square next to the final square on the player's path has gems, then the player must continue using the gems in this square to spread in the same direction. The spreading process is finished if the square next to the final square is empty. If this square is followed by a square with gems, the player can earn all gems inside this square. If there is another empty after the earned square, followed by a square with gems, the player can earn this square too. This is called streak. If there is no square left to be earned, the player's turn will end. If the square next to the final square is followed by an empty square or a half-circle, the player's turn will end.

If there is no gems left in 5 squares of the player's side, the player must use 5 of his/her captured gems and distribute them to all 5 squares, each square has 1 gem. Now the player can choose a square and a direction to spread a gem. If the player has no captured gems left, He/She must borrow 5 gems from the opponent. Every time the player has to use captured gems or borrow gems from the opponent, the score will be decrease by 5

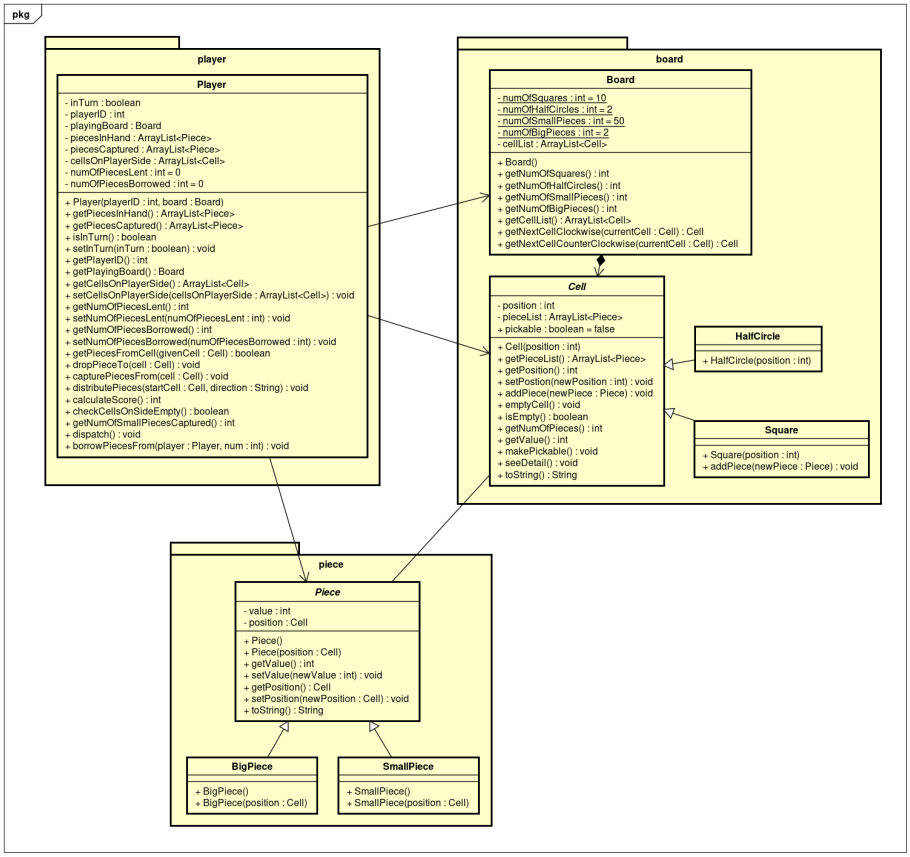
The game only end in one case: All gems in both half-circles is earned. The winner will be decided after calculating the score based on the number of gems earned by each player(1 small gem equals 1 point, 1 big gem equals 5 points)

2.2 Project requirement



- Play: Start the game, the player can choose between 2 modes: Multiplayer and Singleplayer
  - Gameboard: The gameboard consists of 10 squares, divided into 2 rows, and 2 half- circle on the 2 ends of the board. Initially, each square has 5 small gems, and each half- circle has 1 big gem. Each small gem equals 1 point, and each big gem equals 5 points. The player plays the game following the rules stated above until the game is finished. The winner will be decided base on each player's score.
  - The player can choose to go exit the game to go back to main menu or mute sound effect while playing
- Exit: Exit the program, the program will show a confirmation dialog to make sure to ask the users if they really want to quit
- View the instructions: Player can press Help to access the Help Menu from the Intro Screen to read the instructions, the program should display a board showing rules of the game.
- Adjust the setting: Player can adjust the volume of sound effect or background music

3 Class diagram

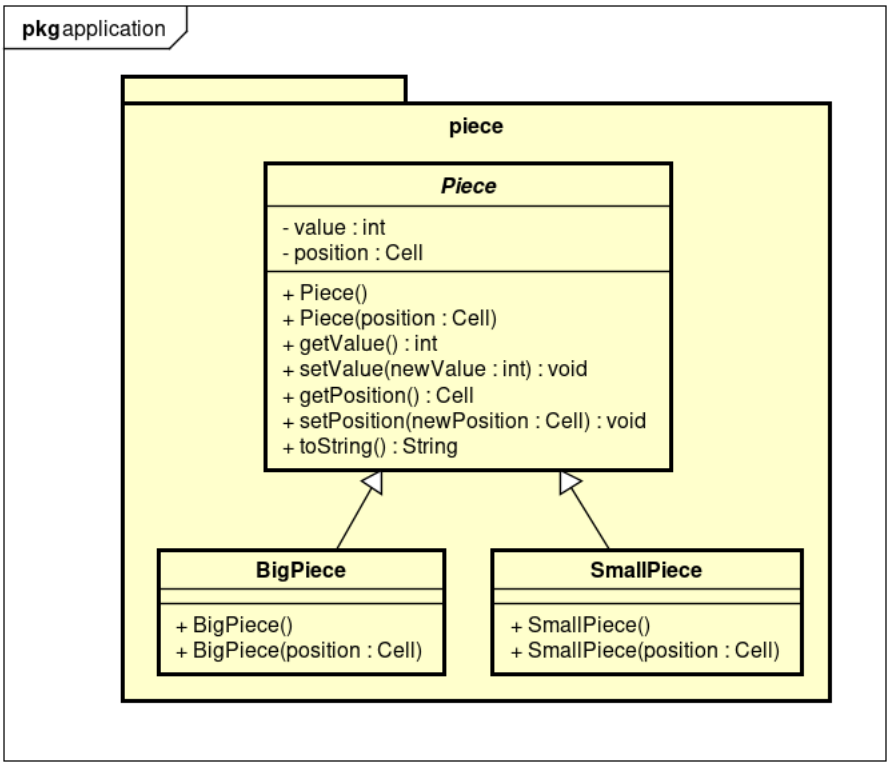


3.1 Class and package overview

3.1.1 piece package

3.1.1.1 Piece class

Piece is an abstract class of the **piece** package. It is the most basic class in the program, representing the pieces(or gems) that the player use to interact with other base classes



1. Attributes:
    - value: Represents the value of the piece
    - position: Represent which cell the piece is in on the game board
2. Methods:
    - Piece: Constructor
    - getValue, setValue: Getter and Setter method for value
    - getPosition, setPosition: Getter and Setter method for position
    - toString: Returns the position and the value of the piece

3.1.1.2 SmallPiece class

Inherited class of the Piece abstract class. The value of a small piece is 1

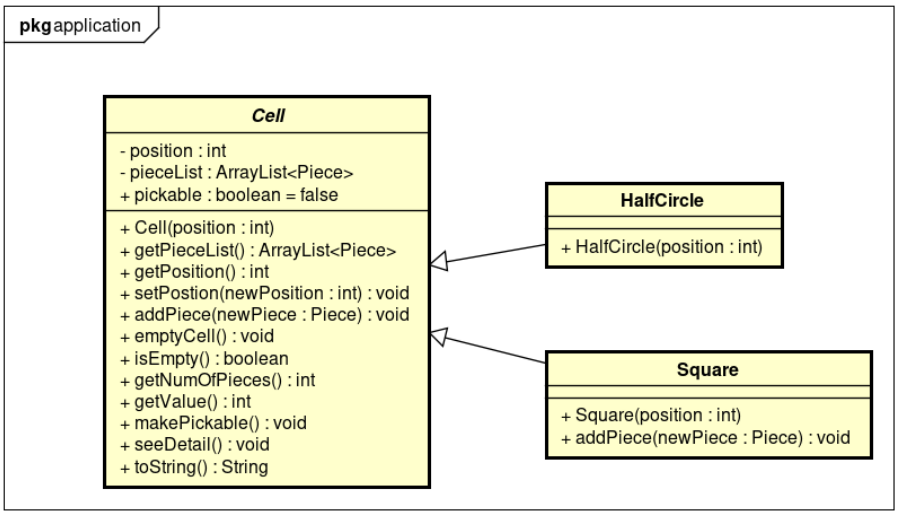
3.1.1.3 BigPiece class

Inherited class of the Piece abstract class. The value of a big piece is 5

3.1.2 board package

3.1.2.1 Cell class

Cell is an abstract class of the **board** package. It represents the cells inside the game board that player interacts while plating. A cell can either be a square(stores smallPiece) or a half-circle(stores big Piece)



1. Attributes:
    - position: The position of the cell in the game board
    - pieceList: An ArrayList of Piece containing all pieces currently in the cell
    - pickable: A boolean value to decide if the player can pick this cell to spread pieces in play turn. Default: false
2. Methods:
    - Cell: Constructor
    - getPieceList: Getter method for pieceList
    - getPosition, setPosition: Getter and Setter method for position
    - addPiece: Add a new piece to the cell
    - emptyCell: Empty the cell
    - isEmpty: Check if the cell is empty or not
    - getNumOfPieces: Get the number of pieces currently in the cell
    - getValue: Get the value of the cell
    - makePickable: Make the cell pickable in play turn

- seeDetail: Return the position of the cell and the list of pieces in the cell
- toString: Return the position of the cell

3.1.2.2 HalfCircle class

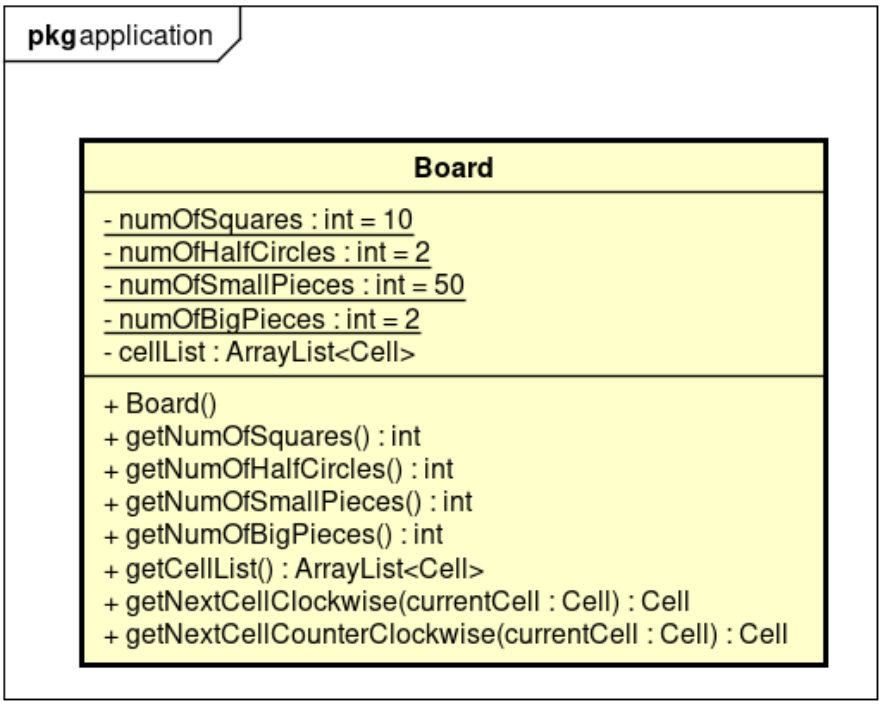
An inherited class of the Cell abstract class, represents a half-circle on the game board

3.1.2.3 Square class

- Methods
  - Square: Constructor. Make the square pickable
  - addPiece: Override the addPiece method of the Cell abstract class. Add new piece to a square

3.1.2.4 Board class

Board is a class of the **board** package. It is the base component of the program, representing the game board for the player to interact with. The Board class contains a cellList and other attributes related to the initial state of the game.



1. Attributes:

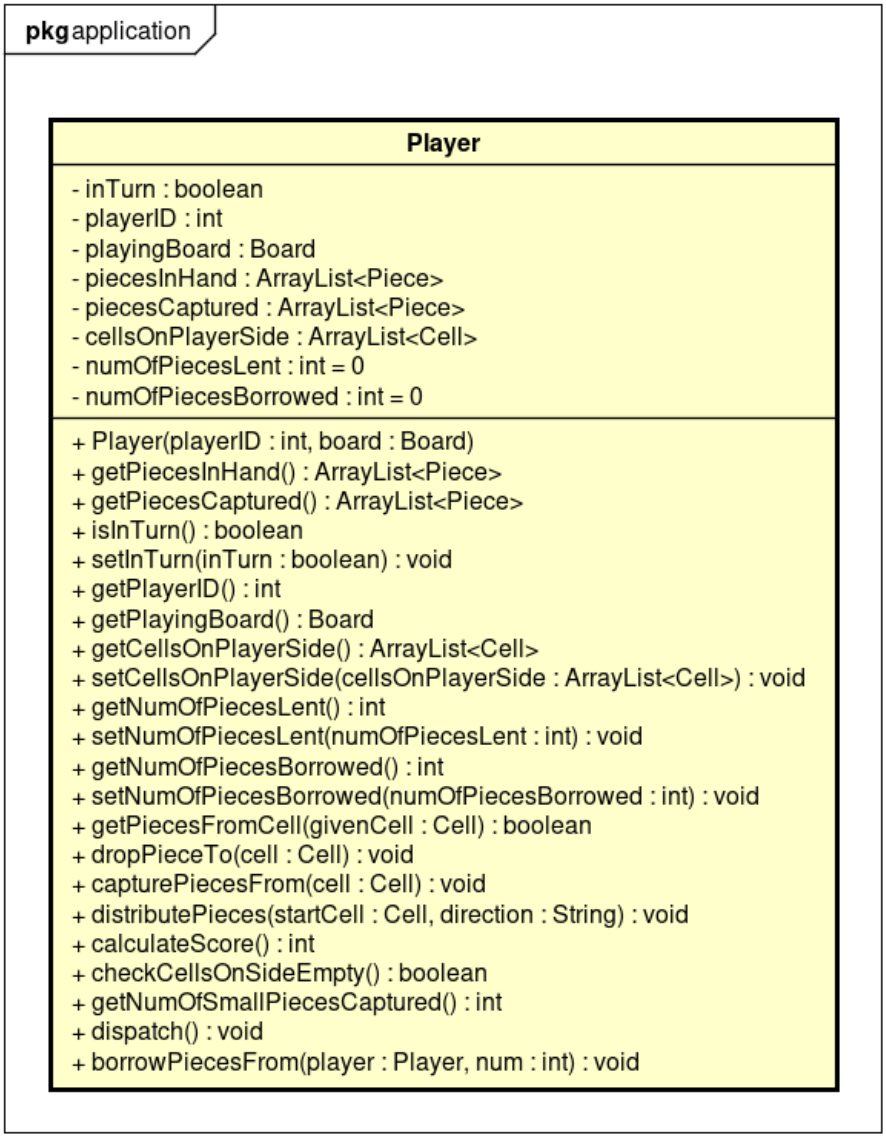
- numOfSquares: A static variable represents the number of squares on the game board. There are 10 squares on the game board
- numOfHalfCircles: A static variable represents the number of half-circles on the game board. There are 2 half-circles on the game board
- numOfSmallPieces: A static variable represents the number of small pieces on the game board at the start of the game. There are 50 small pieces on the game board at the start of the game equally distributed into 10 squares, each square has 5 small pieces
- numOfBigPieces: A static variable represents the number of big pieces on the game board at the start of the game. There are 2 big pieces on the game board at the start of the game equally distributed into 2 half-circles, each square has 2 big pieces
- cellList: An ArrayList of Cell represents all cells on the game board

2. Methods:

- Board: The constructor of the class. Initially placed 1 big piece in each half-circle and 5 small pieces in each square
- getNumberOfSquares: Getter method for numOfSquares
- getNumOfHalCircles: Getter method for numOfHalfCircles
- getNumOfSmallPieces: Getter method for numOfSmallPieces
- getNumOfBigPieces: Getter method for numOfBigPieces
- getCellList: Getter method for cellList
- getNextCellClockwise: Get the next cell in clockwise direction
- getNextCellCounterClockwise: Get the next cell in counterclockwise direction

3.1.3 player package

1. **Player** class is a class of the **player** package. It represents the player of the game.



(a) Attributes:

- inTurn: Determine whether the player is in turn or not
- playerID: The ID of the player
- playingBoard: Represents the game board that the player is playing on
- piecesInHand: An ArayList of Piece, containing all pieces that are currently possed by the player on his/her side of the game board

- piecesCaptured: An ArrayList of Piece, containing all pieces that have been captured by the player, which ban be used to determine player's score by referring the value of the piece
- cellsOnPlayerSide: An ArrayList of Cell, containing 5 squares in his/her side of the game board
- numOfPiecesLent: The number of pieces the player has lent the opponent if the opponent has no piece left to distribute in turn. The final score of the player will be decreased by this number
- numOfPiecesBorrowed: The number of pieces the player has borrowed from the opponent if he/she has no piece left to distribute in turn. The final score of the player will be decreased by this number

(b) Methods:

- Player: The constructor of the class, initially set playerID and playingBoard
- getPiecesInHand: Getter method for piecesInHand
- getPiecesCaptured: Getter method for piecesCaptured
- isInTurn: Return the inTurn state of the player
- setInTurn: Setter method for inTurn
- getPlayerID: Getter method for playerID
- getPlayingBoard: Getter method for playingBoard
- getCellsOnPlayerSide, setCellsOnPlayerSide: Getter and setter method for cellsOnPlayerSide
- getNumOfPiecesLent, setNumOfPiecesLent: Getter and setter method for numOfPiecesLent
- getNumOfPiecesBorrowed, setNumOfPiecesBorrowed: Getter and setter method for numOfPiecesBorrowed
- getPiecesFromCell: Get all pieces on a cell for distrbuting
- dropPieceTo: Drop a piece from hand to a cell when distributing
- capturePiecesFrom: Captures all piece from a cell
- distributePieces: Choose a cell to get pieces from, a direction and distributing pieces
- calculateScore: Calculating the score of the player based on number of pieces captured
- checkCellsOnSideEmpty: Check if all 5 squares on player's side is empty or not
- getNumOfSmallPiecesCaptured: Get the number of small pieces captured
- dispatch: If all cells on player's side are empty, take 5 captured pieces and distribute each one them in a cell
- borrowPiecesFrom: If the player has no captured pieces left to dispatch, he/she must borrow 5 pieces from the opponent

3.2 OOP techniques

3.2.1 Encapsulation

Encapsulation: Encapsulation is the technique of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also involves restricting direct access to some of an object's components, which is achieved using access modifiers like 'private', 'protected', and 'public'. Encapsulation is demonstrated in **Piece**, **Cell**, **Board**, **Player** classes by:

- Making attributes **private** to restrict direct access.
- Providing public getter and setter methods to access and modify private attributes.
- Ensuring that the internal state of the objects can only be changed through well-defined methods, maintaining data integrity and hiding implementation details from the outside world.

3.2.2 Inheritance

Inheritance: Inheritance is a mechanism where one class (child/subclass) inherits the attributes and methods of another class (parent/superclass). This is how inheritance is demonstrated in the project:

- Abstract classes: **Cell** and **Piece** are abstract classes that define common properties and methods for their subclasses
- The **HalfCircle** class extends the **Cell** abstract class, inheriting its properties and methods
- The **Square** class extends the **Cell** abstract class, inheriting its properties and methods, overriding the **addPiece()** method
- The **SmallPiece** class extends the **Piece** abstract class, inheriting its properties and methods
- The **BigPiece** class extends the **Piece** abstract class, inheriting its properties and methods

3.2.3 Polymorphism

Polymorphism: Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. It is often achieved through method overriding and interface implementation. Classes applied Polymorphism:

- The **addPiece()** method is overriden in the **Square** class. While **Cell's addPiece** method allows adding any **Piece**, **Square** restricts this to only instances of **SmallPiece**.
- The **toString** method is overridden in various **Cell** class and **Piece** class, providing class-specific string representations.

3.2.4 Abstraction

Abstraction: Abstraction involves creating simple models representing more complex underlying code and data. It helps in focusing on what an object does instead of how it does it. **Cell** and **Piece** are abstract classes that define common attributes and methods for their subclasses. The specific details of **Cell** types (**HalfCircle** and **Square**) and **Piece** types (**BigPiece** and **SmallPiece**) are abstracted out

3.3 Objects relationship

3.3.1 Association

Association:Association represents a relationship between two separate classes that are aware of each other. It is the most general form of relationship where one object can be associated with another

- Each **Piece** object holds a reference to the **Cell** objetc, each **Cell** object also holds a reference to the **Piece** object so **Piece** associates **Cell** and vice versa
- Each **Player** object holds a reference to the **Board** object, **Cell** object and **Piece** object so **Player** associates **Board**, **Cell** and **Piece**

3.3.2 Aggregation

Aggregation: A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts. An aggregation is an "is a part-of" relationship. A **Board** has a list of **Cell** objects, but a **Cell** can exist independently of a **Board**. We can say that **Board** aggregates **Cell**

3.3.3 Composition

Composition: Composition is a strong form of aggregation where one object "owns" another object. The child object cannot exist independently of the parent object. If the parent is destroyed, the child is also destroyed. The **Cell** class and the **Piece** class have a composition relationship. A **Cell** contains **Piece** objects. When a **Cell** is destroyed, its **Piece** objects are also destroyed. We say that **Cell** composites **Piece**

4 Demonstration

Setup instructions

1. Go to project repository `https://github.com/kazuyuki114/Mandarin_Square_Capturing`

2. Clone the project to your local machine by running this command in the terminal: "git clone https://github.com/kazuyuki114/Mandarin_Square_Capturing.git"
3. Follow instructions in `https://openjfx.io/openjfx-docs/` to install Java and JavaFX
4. Open the project in an IDE that supports Java
5. Run the application in Main class /OAnQuan/src/application/Main.java
6. Finally, have fun playing the game

Demo video

5 References

1. Game rules in English: `https://shorturl.at/BguHT`
2. Bot implementation: `https://github.com/WHKnightZ/Graphics_O_An_Quan`