# OBJECT-ORIENTED PROGRAMMING MINI-PROJECT REPORT SEMESTER 20212

# TEAM 26 – O AN QUAN GAME

Instructor: Prof. Nguyen Thi Thu Trang

Hoang Long Vu – 20204897 Pham Vu Huyen Trang – 20207997

[vu.hl204897@sis.hust.edu.vn](mailto:vu.hl204897@sis.hust.edu.vn) [trang.pvh207997@sis.hust.edu.vn](mailto:trang.pvh207997@sis.hust.edu.vn)

Nguyen Huu Vuong – 20204898

[vuong.nh204898@sis.hust.edu.vn](mailto:vuong.nh204898@sis.hust.edu.vn)

## 1. Assignments of members

General contribution:

1. Hoang Long Vu: 50%
2. Pham Vu Huyen Trang: 50%
3. Nguyen Huu Vuong: 0% (\*)

*(\*): Team members could not contact with this member, either by direct messages or direct talking on classes as he does not seem to attend any lectures, thus did not receive any response on the decision to contribute to this Project*. *With only two members, the tasks are not easy to be assigned separately but we mostly work together through this project, thus many tasks are overlapped.*

The details of tasks are described in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Member/Task | Problem Modelling | Model | View | Controller | Other tasks |
| Pham Vu Huyen Trang | Brain-storming and constructing the diagrams for the problem. [1] | - Gem  - Player (60%) – spreadGem, and other methods except pickUpGemFrom and dropGemInto - GamePlay [3] | - PlayScreen.fxml  - GamePlayScreen [4] | GamePlayScreen Controller (80%) - handling almost everything in GamePlayScreenController except exitButton and displayEndGameScreen after the game has finished  [4] | - Design Slide  - Fix report  - Music handling  - Record demo video |
| Hoang Long Vu | Brain-storming and constructing the diagrams for the problem. [1]­­­ | - Board, Cell  - Player (40%) - pickUpGemFrom, speadGem, dropGemInto  - BoardTest, PlayerTest [3] | - IntroScreen.fxml  - IntroScreen  - HelpScreen.fxml  - HelpScreen  [4] | - IntroScreen Controller  - Help Screen Controller  - GamePlayScreenController (20%) – handing exitButton and displayEndGameScreen after the game has finished  [4] | - Fix Slide  - Write Report  - Background picture design [2] |

References:

1. Ideas for the general structure of the program:

[<https://github.com/linusericsson/oware>]

2. Ideas for the (aesthetic) design of user interfaces:

[[Game Ô ăn quan - Dân gian - Game Vui](https://gamevui.vn/o-an-quan/game)]

3. Game rules:

[<https://hocvienboardgame.vn/huong-dan-tro-choi-o-an-quan/>]

4. Ideas for designing GUI structure (based on the slight similarities in the board component):

[<https://github.com/Querz/chess>]

## 2*.* Mini-project Description

### 2.1. About the game

This two-player game 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 in either direction: clockwise or counter-clockwise. The player must drop one gem into every cell on their 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 is called “terminal”. If “terminal” has gems, then the player can continue using the gems in this square to spread in the same direction. A spread is finished if the “terminal” is empty.

After finishing spreading, if the “terminal” (empty) is followed by a square with gems, the player can earn all gems inside this square. If there is another empty square after the earned square, followed by a square with gems, then the player can continue the streak in the same fashion.

During one turn, the players cannot change their direction at any time.

The score is evaluated by the number of gems earned by each player (a small gem equals to 1 point, and one big gem equals to 5 points).

The game ends if one of the following cases happens:

1. Both of the half circles are empty.

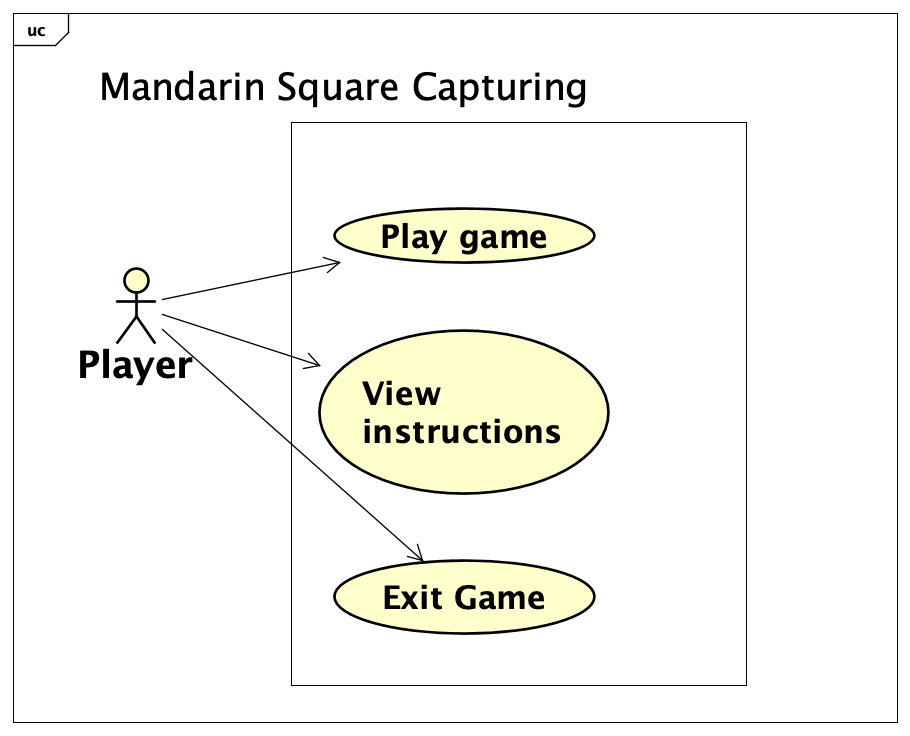
2. If a player is in turn, but all the squares that player possesses are empty.

### 2.2. Project requirements

On the main screen, the game should have the following buttons:

* **Start:** Start the game, no need to create different difficulties.
* **Exit:** Exit the program, be sure to ask users if they want to quit the game.
* **Help:** Show guide for playing the game.

#### 2.2.1. Use cases



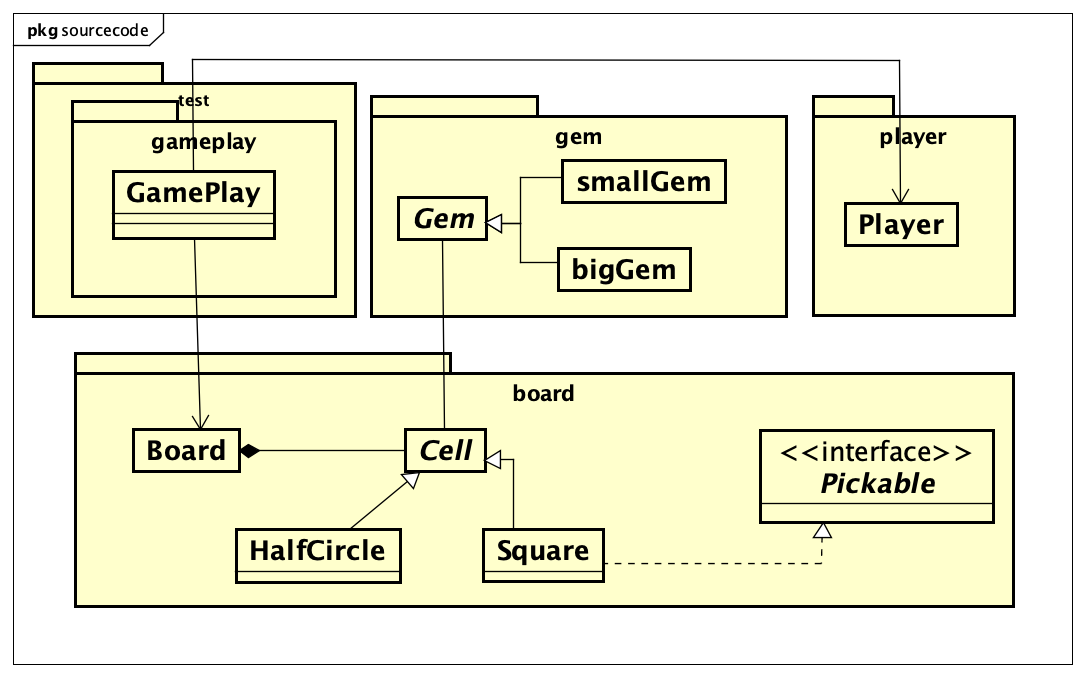
*Use case diagram*

The user (*Player*) in this program has two use cases: Play game, and Show instructions. Specifically:

* Play game: When play choose to start the game (by clicking the **Start** button in the Intro Screen), the program displays a playing board, the player then plays the game following the rules stated in (2.1) until the game is finished. The player can choose to exit the game while playing, on which occasion the program will ask for confirmation and quit the program based on the decision from the user. The player can choose to exit at the introduction screen, too.
* View instructions: Player can press Help to access the Help Menu from the Intro Screen to read about the instructions, the program should display a board showing rules of the game.
* Exit Game: In the intro screen or while playing, the player can click the Exit button to quit the program. The program needs to show a dialog for confirmation. The player can exit the program after clicking the OK button in the dialog, else choose Cancel to stay in the program.

### 1.1. Design

#### 1.1.1. General class diagram and classes relationship



O An Quan Use Case Diagram

We divide our program into two main parts: the “active” part (components that initiate the actions in the program, e.g. *Player* class) and the “passive” part (components that are initiated in the program, e.g. *Board*, *Cell*, and *Gem* class). The *Cell* and *Board* classes are placed in the *board* package. The *Gem* is placed separately in the *gem* package. Similarly, the *Player* class is contained in the *player* package.

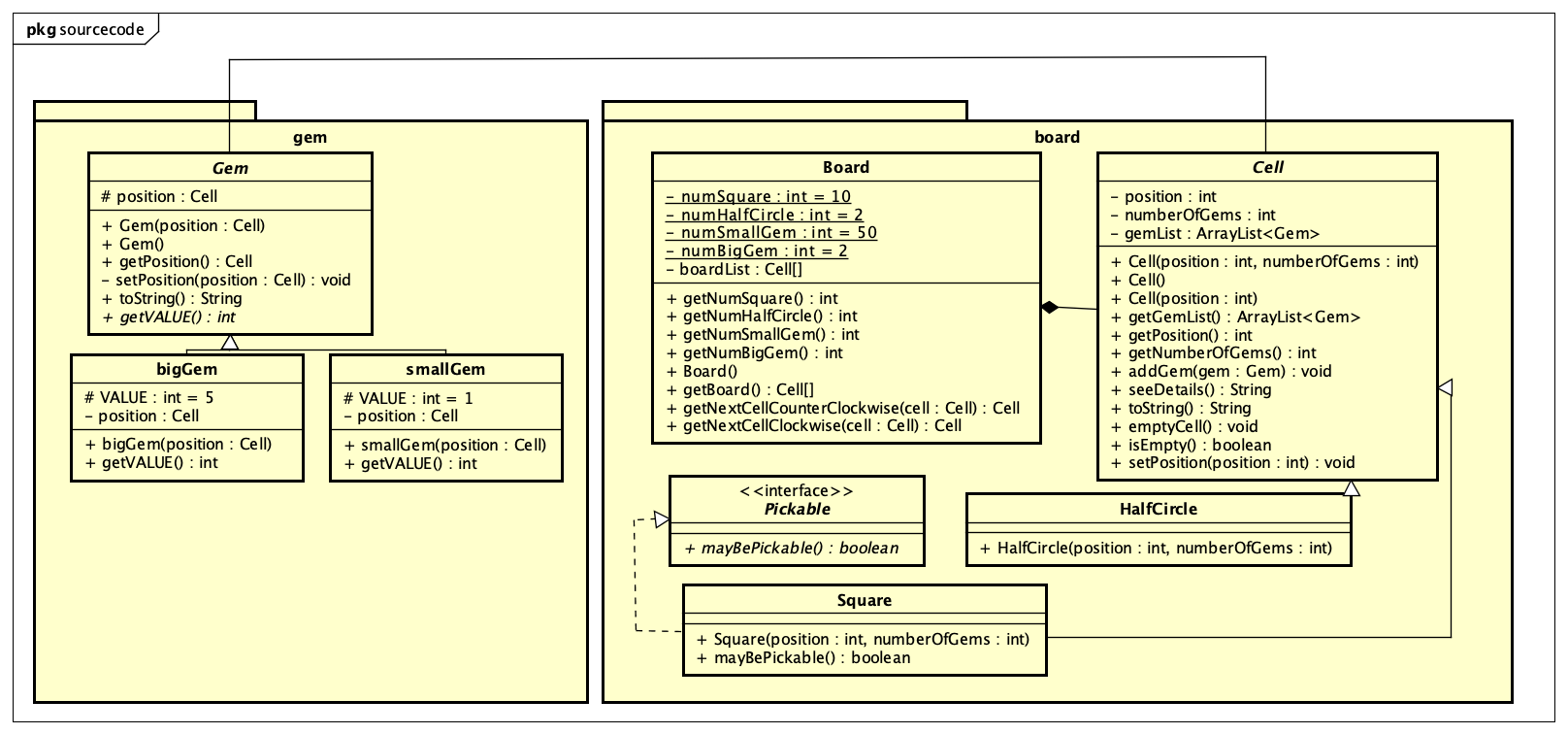
**Relationships:** The abstract *Gem* class has two inherited classes: *smallGem* and *bigGem*, this *Gem* class aggregates the abstract class *Cell* (with two inherited classes: *HalfCircle* and *Square*). The *Cell* class then aggregates the *Board* class to construct the main board for players to interact with.

The *GamePlay* class is used for testing the actions and procedures of the game (initiating players, board; picking up, spreading and eating the gems; etc.)

More information on the details and relationships between classes is discussed in the sections below.

#### 1.1.2. Detailed class diagrams

**Firstly**, we consider the “passive” components in the program, i.e. the components that players manipulate during the game.



Class Diagrams for board and gem packages (with Gem, Board, Cell as parent classes)

Class *Gem:*

This is the most basic class in our program, representing the gems (or stones) that players use to interact with other base classes (*Cell* and *Board*). Objects in this abstract class have one single attribute: position, which represents the cell containing this gem. The *Gem* object has two other inherited classes: *bigGem* and *smallGem*, where both have another VALUE attribute to determine how valuable this gem is. Traditionally (and in this program), big gem is worth 5 points, and small gem is worth 1 point. These points are then converted to scores for the players.

The **polymorphism** can be applied here. The *bigGem* and *smallGem* can be both called as an object of the *Gem* class, but has different value of the VALUE attribute. The score of the player is calculated by the value of the Gem, which should have different value depending on the type of the gem.

Moreover, toString method has been overridden in class smallGem and bigGem. This is also an application of **polymorphism**.

Class *Cell:*

This abstract class represents the cells inside the board, with which users interact while playing. The *Cell* class has three attributes: position (integer, representing the “index” of the cell on the board), numOfGems (integer, representing the number of gems currently exist in this cell), and gemList (an ArrayList, containing list of *Gem* objects currently exist in this cell). There are two classes that inherit this class *Cell*, that are *HalfCircle* and *Square*. The *Square* class implements the *Pickable* interface, allowing users to pick gems from *Square* objects (picking gems from any half circles is prohibited).

A cell can either be a half circle or a square, where a half circle is the house to store big gems. The **polymorphism** can also be applied here since only the *Square* is “pickable”. When looping through the cells (*Cell* objects) in a board, depending on the type (*Square* or *HalfCircle*), the program will decide if the player can pick up gems from that cell or not.

The objects created by *HalfCircle* and *Square* together construct the complete *Board* for users to play. Traditionally (and in this program), one board contains 10 squares divided into 2 rows, along with 2 half circles on two ends of the board.

We use Gem as an attribute of the Cell class and use Cell as an attribute of the Gem class, thus the class *Cell* is said to associate the *Gem* class and versa.

Class *Board:*

This is the base component in our program, representing the board for players to interact. The *Board* class contains objects of *Cells* (which can be half-circles or squares). This class contains attributes related to the initial state of the game (number of squares, number of half-circles, number of small gems, number of big gems) and one list of cells (*board*) for convenient manipulation of the *Board* objects.

In the Constructor of *Board:*

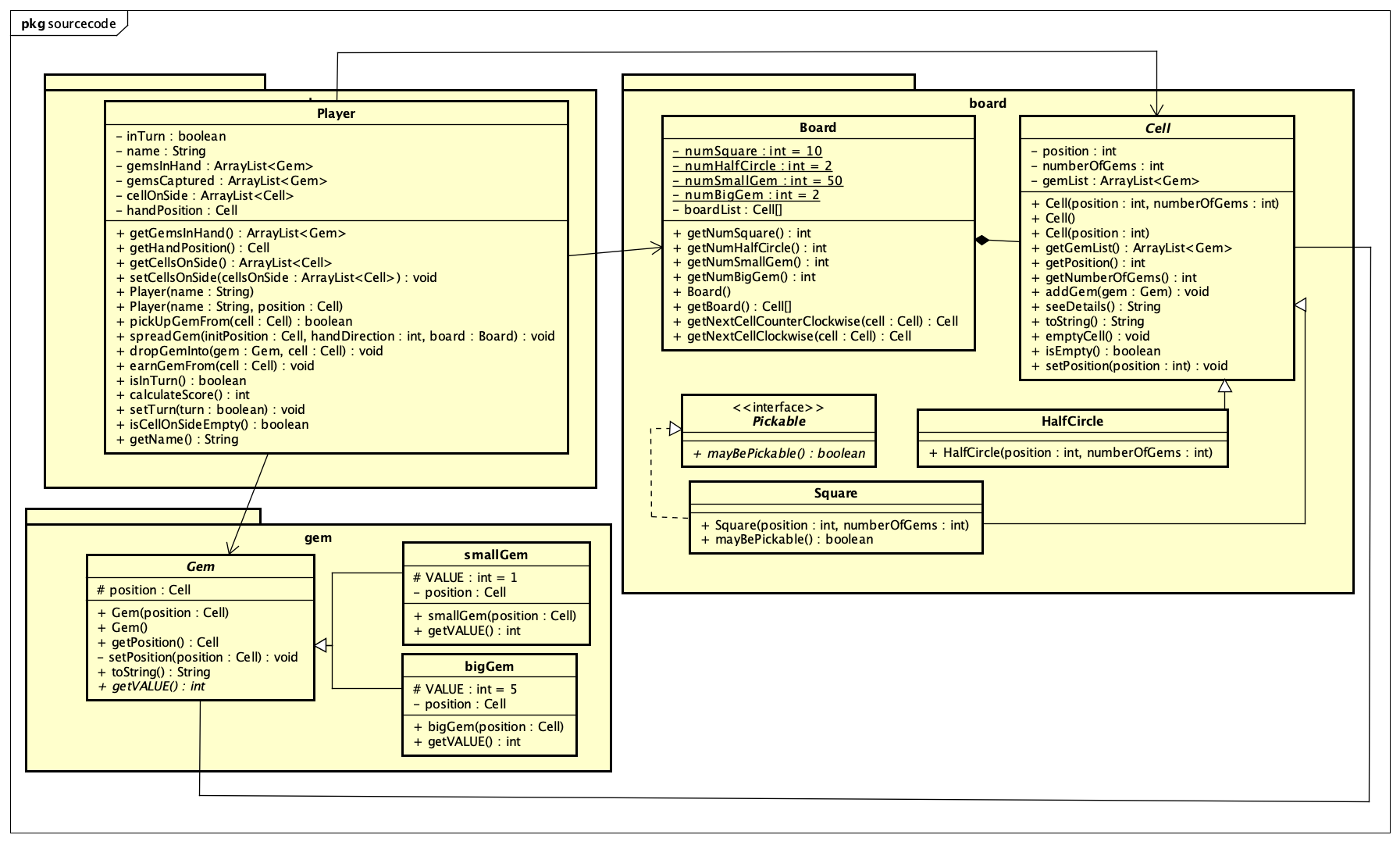
* The big gems are initially placed in the half circles (which has the position of 0 and 6 in our board). Other squares will have 5 gems each.
* We can get the next cell of one cell, considered in clockwise or counter-clockwise direction. These methods are useful in the action of spreading.

Both *HalfCircle* and *Square* classes inherit from the *Cell* class, sharing the same three attributes: position, numOfGems and gemList. Half circles and squares have distinct roles and position characteristics: half circles are located on two ends of the board, and this is the only instance that can contain big gems; whereas squares can only contain small gems.

The *Cell* objects are parts of a *Board* instance, thus the *Board* class is said to composite the Cell class.

The Gem object is an attribute of Board class, then the Board class is said to associate the Gem class.

**Secondly**, we come to the “game-related” components of the program: players – which can be considered the “manipulator” part of the program.



Class Diagram for Player class, along with its relationship with other aforementioned classes

Class *Player:*

This is the initiator of actions in the program, representing the players of the game. The class has the following attributes:

* inTurn (Boolean): determine whether this player is in turn or not.
* score (int): the score of the player
* name (String): name of the player
* id (int): id of the player
* gemsInHand: an ArrayList containing the gems that are currently possessed by the player.
* gemsCaptured: the gems eaten by players, which can be used to determine the player’s score by referring to the value of the gem.
* handPosition: the current cell that the player’s hand is pointing to.
* cellsOnSide: ArrayList of gems in the cells possessed by the player’s side.

Explanation for methods in this class:

* pickUpGemFrom(Cell cell): pick up all gems from the cell *cell*. By the rules, players cannot pick up gems from any of the half-circles on the board.
* spreadGem(Cell initPosition, int handDirection, Board board): starting from the cell *initPosition*, the player will spread gems on the board *board* in the direction of *handDirection*. This method also allows players to continue spreading in their turn if applicable (i.e. the “terminal” is not empty); and allow players to eat all the gems in one cell if applicable.
* dropGemInto(Gem gem, Cell cell): drop each *gem* into the cell *cell*, used in the procedure of spreading where users need to drop the gems into the cells on their path.
* earnGemFrom(Cell cell): earn all the gems inside the cell *cell.*
* calculateScore(): calculate the score of players based on the value of the gems eaten.

*The Player class associates the Cell, Board and Gem Class because Gem, Board and Cell are attributes of Player class.*

Beside designing class describing the “board-related” components, we also design:

*Class* *GamePlay:*

This is the main test class of the program, initializing the necessary components: board, players, decide the cells possessed by the players, determining if the game is over or not. After the game is finished, the program will display the winner of the game (i.e. the one with higher score).

The Gem, Board, Cell, Player class associate class GamePlay.

***Note (Polymorphism in GUI):*** In the isPaneEmpty method of the class GamePlayScreenController, the children of the Pane is retrieved via the built-in method getChildren, in which only the Label objects can implement the getText method.