**Section 1: Statement of the problem to be solved**

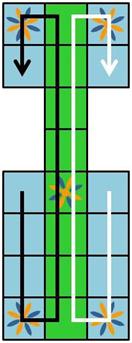
Problem: Final Project – Game of Ur with GUI.

Our group is going to be making a board game heavily based on the 5000 year old game, The Royal Game of Ur.

* The game will launch and the user will be presented with two options, to either start a new game or to quit, which will close the applicaiton. Once the user clicks on new game, the user will see the game board with a row of seven, round white pieces to the top and seven, round black pieces to the bottom of the board. The user will also see a menu bar at the top of the window, with different options for the user to select, such as a “Rules” button. Below the board the user will see a dice roller as well as a button to pass their turn to the opponent either at the end of their turn, or if they cannot move any pieces with the number they rolled.
* The user will always go first at the start of the game
* The goal of the game is to race the other player to get all your pieces off the board before they get all of theirs off the board
* To move, the player will roll four dice, each with four sides. Each dice only has one number on it, the number one. The max number a user can get from a roll is a Four.
* Once the player has rolled, they can click on their piece to see if they can move their piece. If they can move the piece, a red circle will appear on the board where the piece can move based on the number that they rolled. If there is no option available, a message will apear telling the user that there is no movement option and to pass their turn.
* The pieces will move in a elliptical shape around the board (see picture below). The first four tiles as well as the last two tiles that the user will move their pieces on are considered a “safe” space. The opponent cannot enter those tiles and vice versa. All the other tiles that run through the middle of the board are a “combat” zone.
* If the user rolls a number that would land their piece on a tile that the opponent currently occupies and they choose to move their piece to that tile, the opponents piece will be knocked off the board and will be forced to restart from the beginning. The users piece will then occupy the tile that the opponent was knocked off of.
* There will be around five (subject to change between now and the final product) special tiles that the user can land on. These tiles will enable them to roll again and either move the same piece or move a completely different piece. While four of these tiles are in the “safe” space for each player, one of them is in the middle of the board and can be occupied by either user. The tile in the middle also serves as another “safe” space for the players. Pieces that stay on this tile cannot be knocked off the board by an opponents piece as long as the piece stays on that tile.
* To get a piece off of the board, the user must roll the exact number of moves that it would take to move the piece off the board. For example, if the user has a piece one tile away from the end of the board, the user MUST roll a one to get that piece off of the board
* Once a player gets all their pieces off of the board, a window will open with a message that will be displayed. The message will either read “You Win” or “You Lose” depending on whether or not the user won or lost. After a brief pause, the user will be presented with the option to either play again, which will start a new game, or to quit, which will close the application.

Picture of the game board

The blue area represents the “safe” spaces. The green area represents the “combat” area.



**Section 2: List of Inputs, Outputs, And Resources Used**

INPUTS

* The GameDriver Main class will include a GUI-based user interface (UI) to display the welcome screen.
* UI to start the 2-Player Point and Click Game and display the Game Board.
* UI for Player to start Turn, RollDice, select Piece and Tile for PieceMove.
* UI for prompting next Player Turn, if game is not finished.

OUTPUTS

* Use JFrame/JPanel GUI for well-formatted, user friendly display of GameBoard, Player Pieces, and Dice information in response to player point and click action events.
* Save the state of the game following each Turn.

RESOURCES USED

* Used Stack for team communications.
* Used GitHub for code source control.
* Used Astah for UML class diagram.

**Section 3: Identification of classes needed and a list of their individual responsibilities**

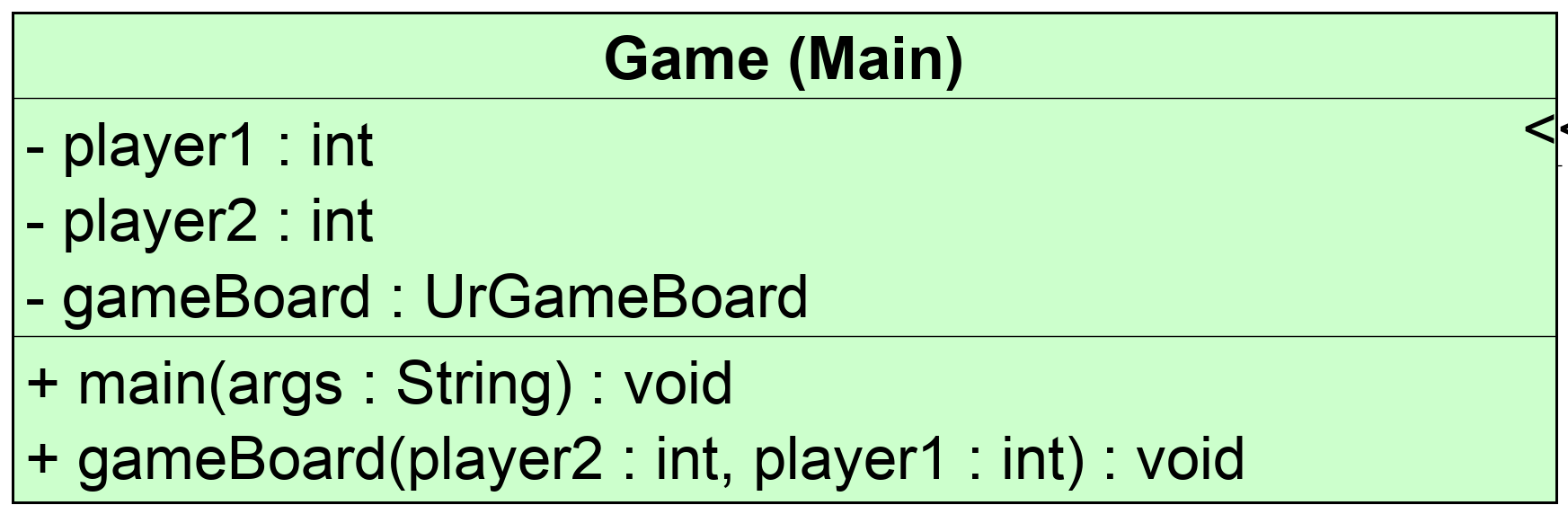
**Class name:** **GameDriver (Main)**

Responsibilities:

* Display Game GUI inteface.
* Start game play.
* Save the current state of the Game using serializable.

Main Method:

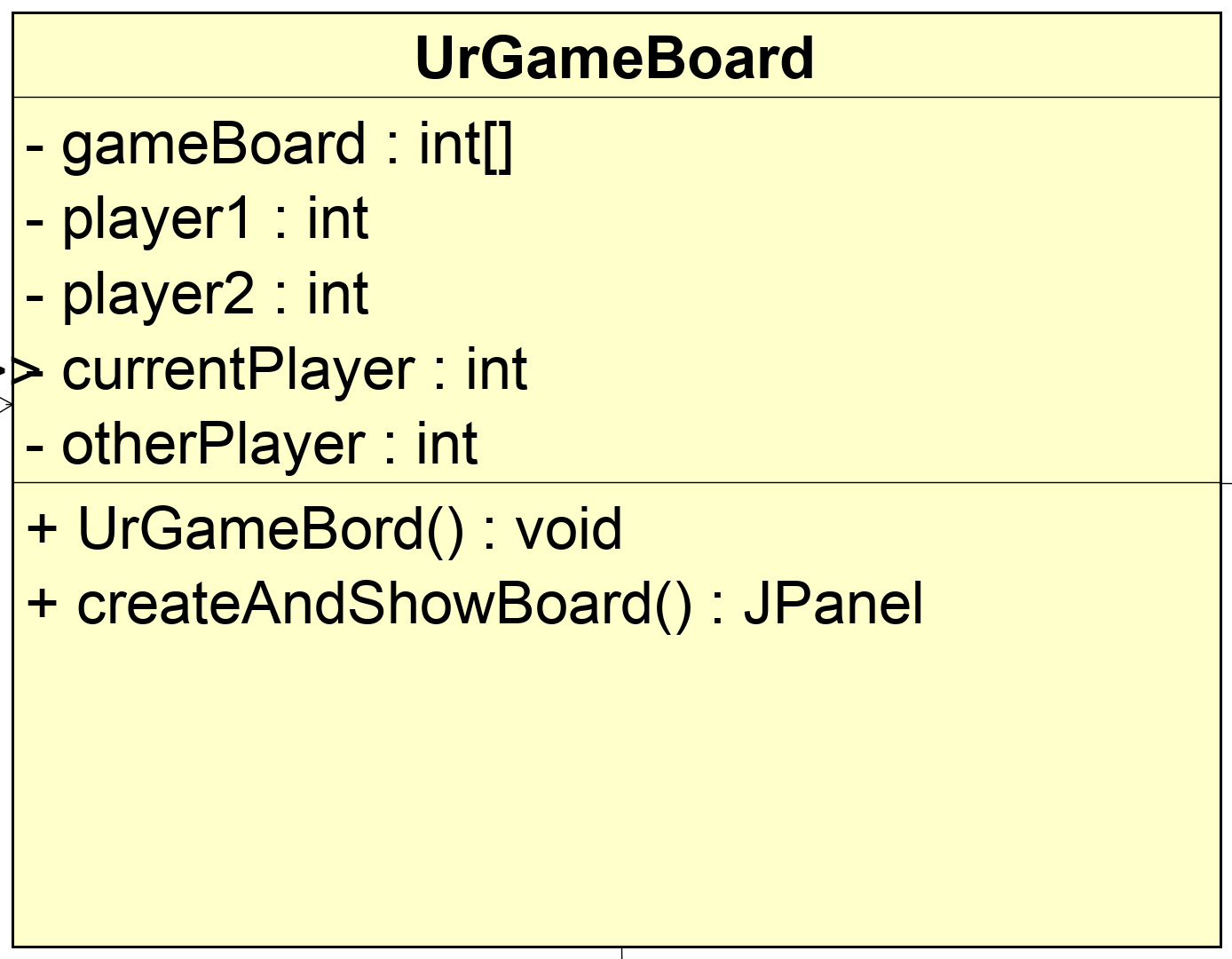
* Start the Game GUI interface.
* Initiate the UrGameBoard object to set up the game board Tiles and Players.
* Save the current game to file.



**Class name:** **UrGameBoard**

Responsibilities:

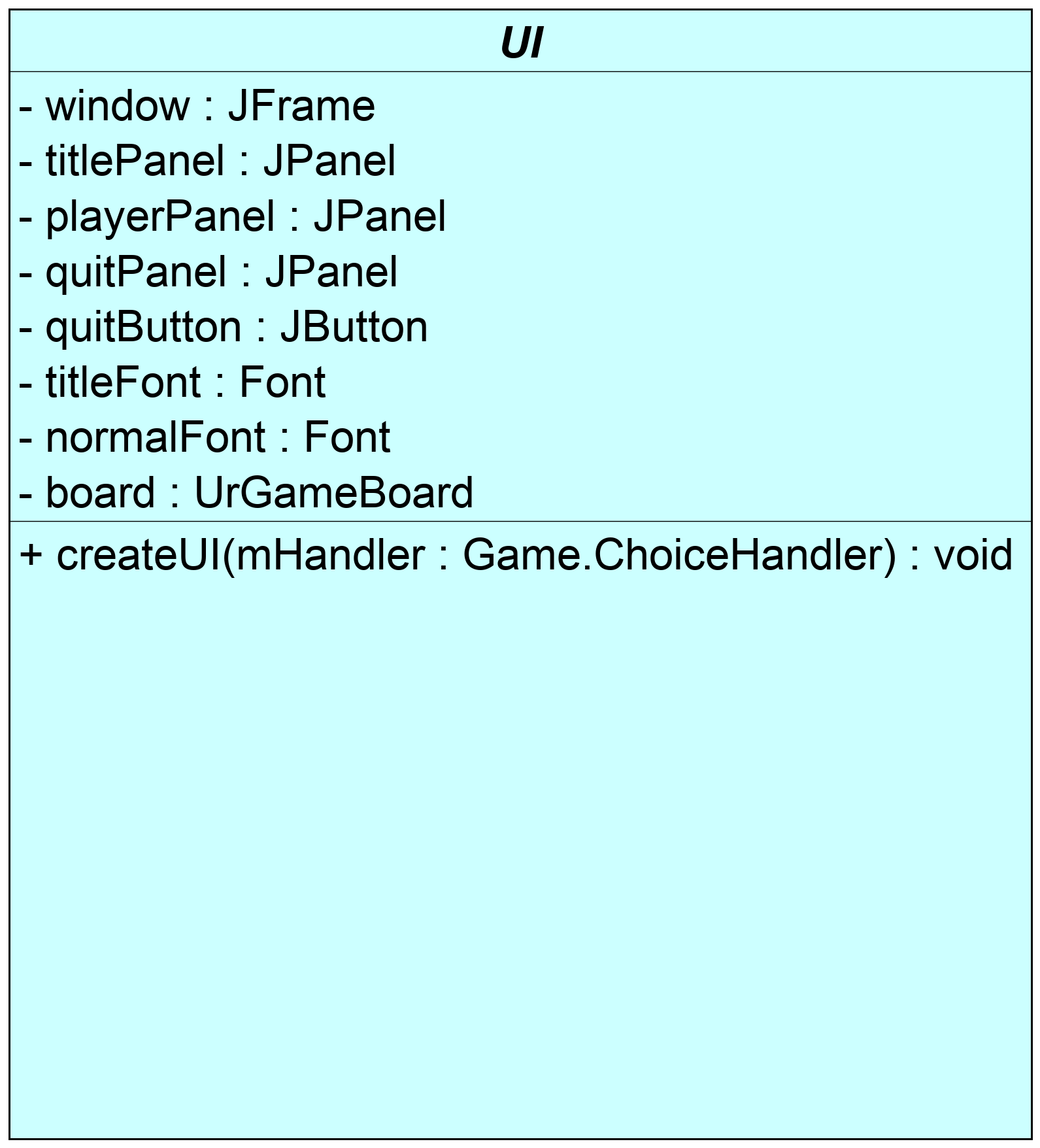
* Constructor for initializing two Players and the gameboard array of 14 Tiles with specific fields that define the state (see Tile class) of each board space as the game progresses.



**Class name:** **UI**

Responsibilities:

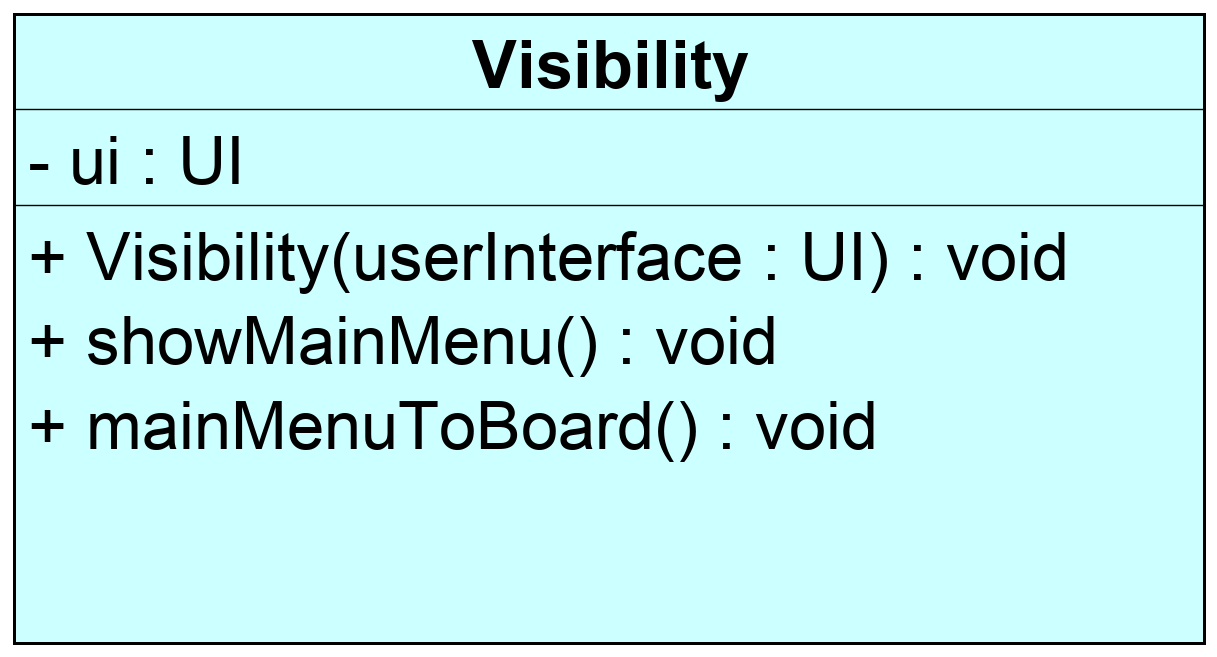
* Abstract class that sets up GUI frame and components that interface code.



**Class name:** **Visibility**

Responsibilities:

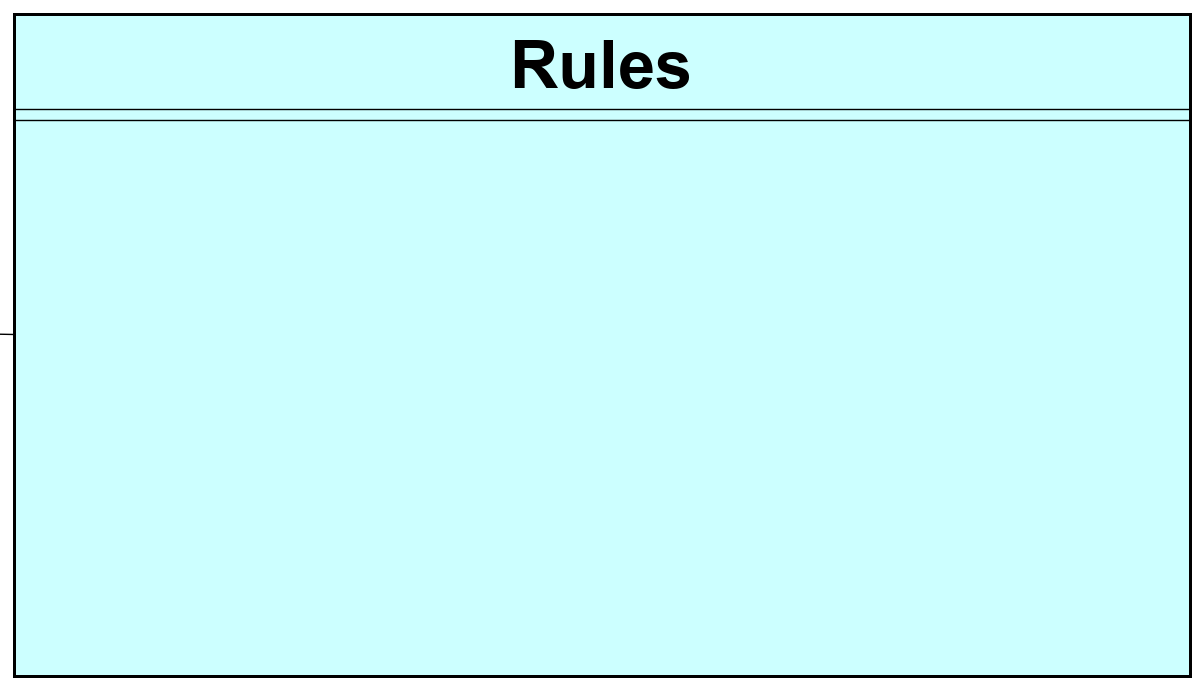
* Method Name: mainMenuToBoard.
* Method Purpose: This method makes sure that the board IS visible to the user while also making sure that the main menu is NOT visible to the user.

****

**Class name:** **Rules**

Responsibilities:

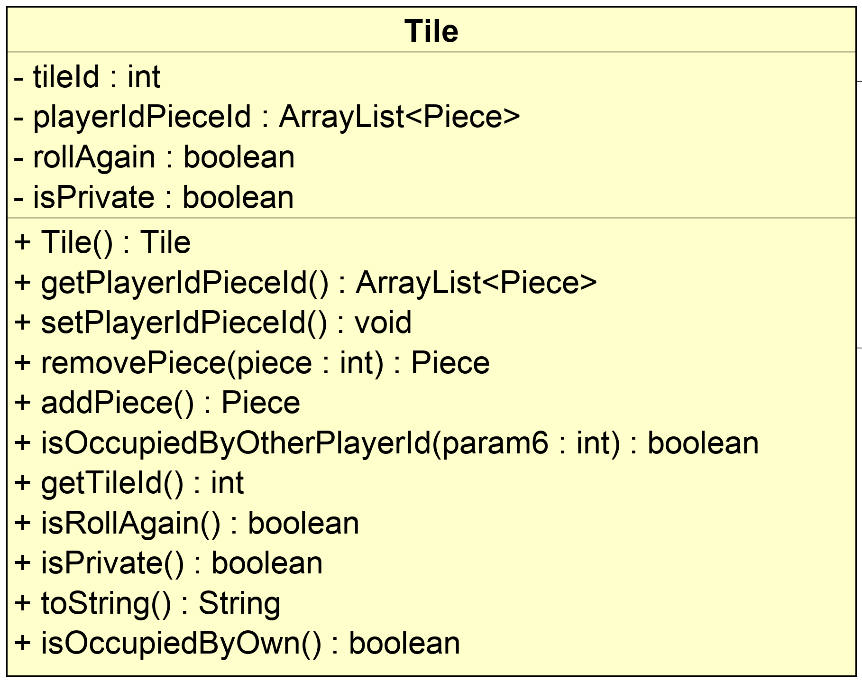
* The Rules.java class that creates and displays an image with a brief description of the rules when the "Rules" button is clicked by the user .

****

**Class name:** **Tile**:

Responsibilities:

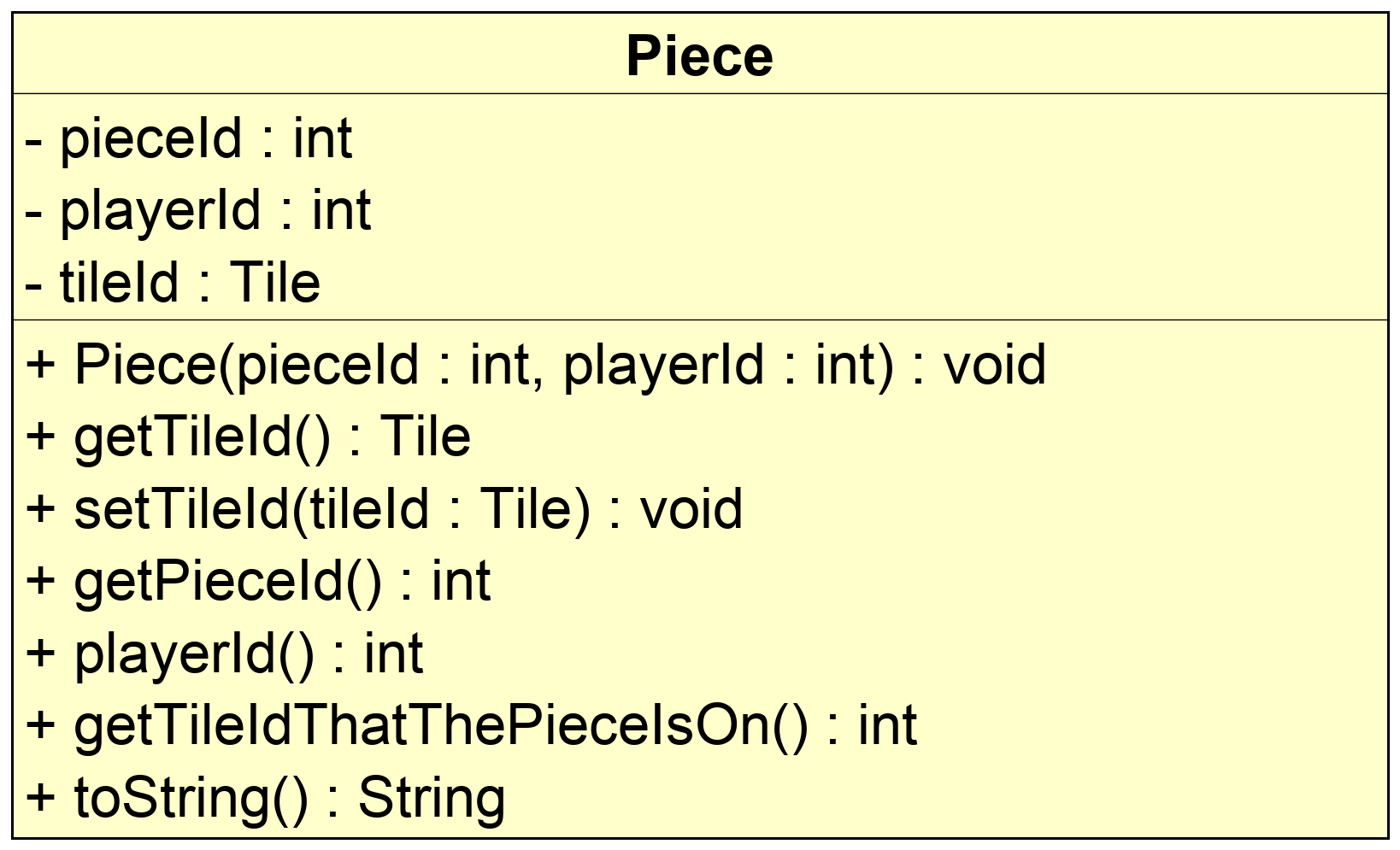
* The Tile class tileId field instance variable (type int) is used to assign the order position of the 14 board spaces.
* Each Tile object is assigned a value between 1 and 14 and establishes the board order of the Tiles (spaces on the board).
* Each Tile object includes a properties of the space. One property is the shared space (isPrivate - true); that is, both players can occupy the space at the same time. A second property is a space not shared by both players (isPrivate – false); that is, only one player at time can occupy the space. This is a space that one player can bump the other player piece off the board.
* Another property is the roll again space (either shared or not shared) that allows the player to roll again.
* Each Tile object (tileID) also includes an ArrayList of Piece objects (playerIdPieceId) that stores information on the Player on that space (playerID) and unique Piece ID on that space (pieceId) during game play.
* The methods include getters and setters, as well as add a piece to the space, check to see if space is occupied by current player or other player, and removing piece from the space.



**Class name:** **Piece**

Responsibilities:

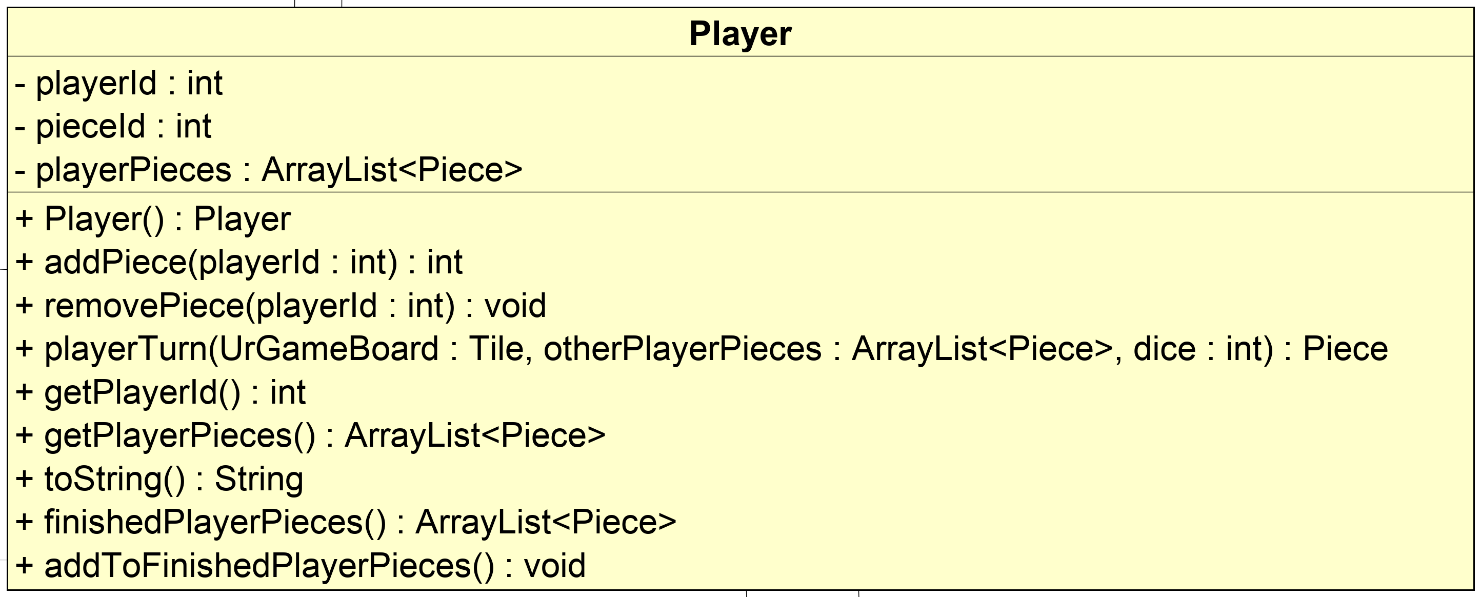
* Piece class is used to create and assign 7 pieces (Piece objects) for player 1 and 7 pieces for player 2. In turn, each piece is assigned and moved from TileId to TileID by being added to and removed from the Tile field ArrayList of playerIdPieceId Piece objects.
* If a neither player’s piece is a board spaces (Tiles), then Tile field ArrayList of playerIdPieceId holds a value of null.
* The methods include getters and setters.



**Class name:** **Player**

Responsibilities:

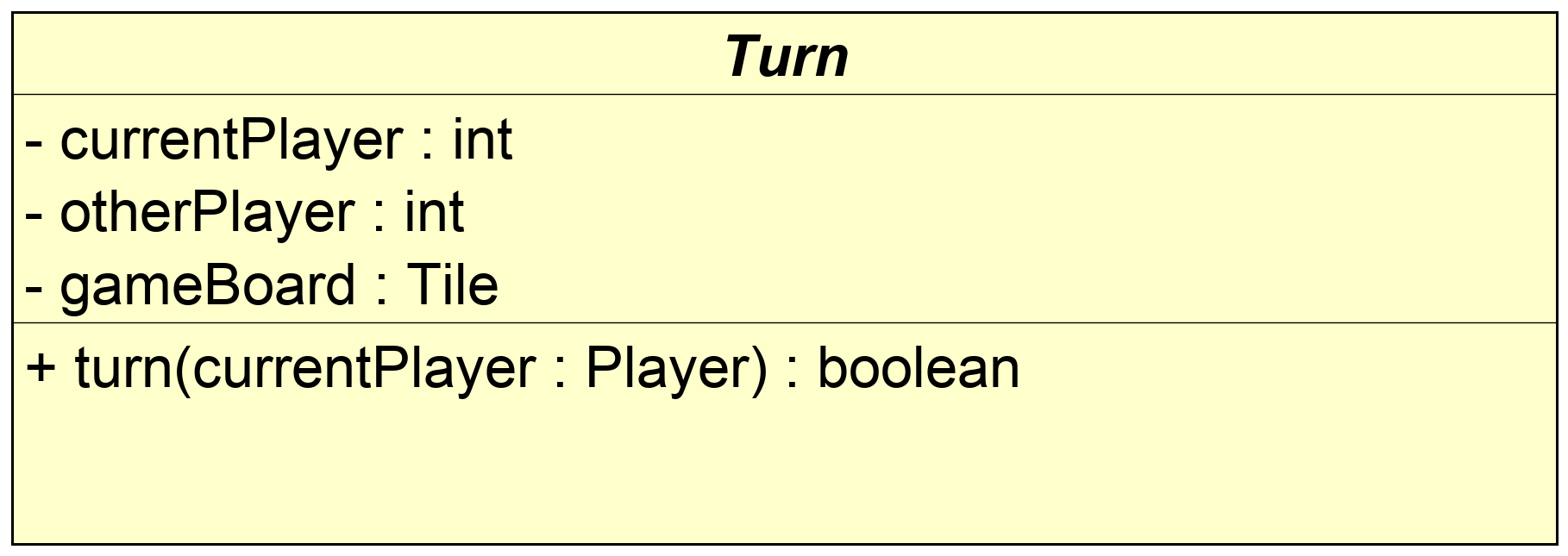
* Player class is used by the UrGameBoard to create 2 players, player1 and player2.
* Each player in turn will be associated with 7 pieces (Piece object with pieceId number), a playerPieces ArrayList, that keeps track of when the player moves onto the board (removePiece) or gets bumped off the board by the other player (addPiece).
* A second ArrayList of finishedPieces is added to as a player moves a pice off the board at the finish line (if ArrayList<Piece> finishedPieces.length() is greater than 7, the the player wins.
* The pieceId assigned during the construction of the player object will be associated with the tileId as it moves on to the game board, around the game board, and, ultimately, off the game board.
* The methods include getters and setters, as well as playerTurn and those mentioned above.



**Class name:** **Turn**

Responsibilities:

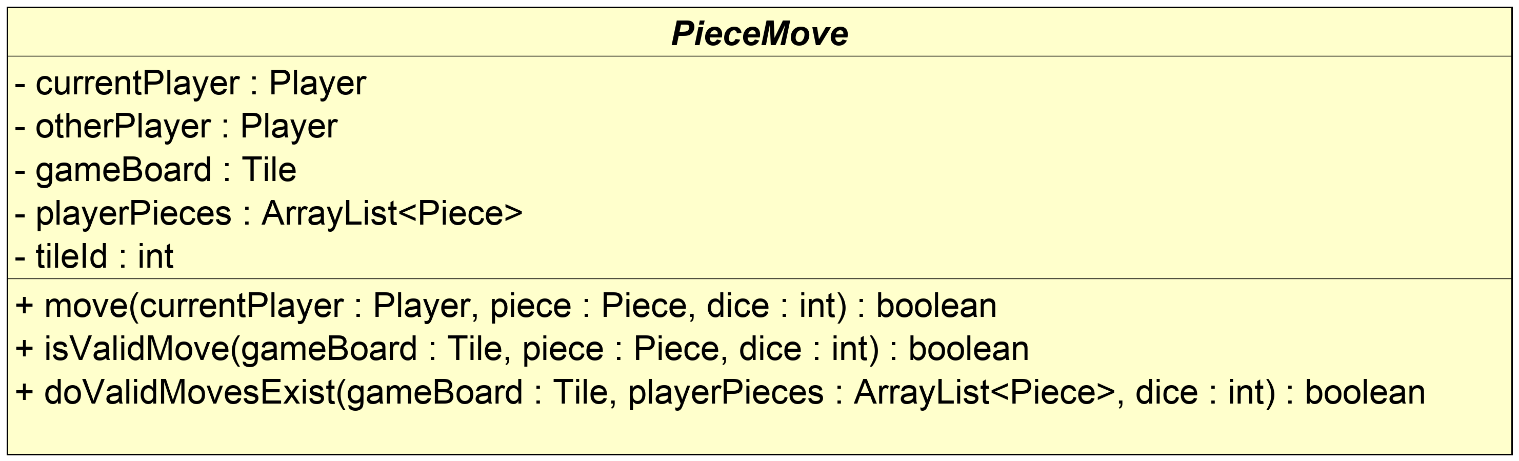
* This abstract class includes a fully implemented method that takes care of one player’s turn. It first changes the active player for the next turn, then rolls a dice, then ask what the player wants do do, if there is a possible move and finally makes that move, if possible.



**Class name:** **PieceMove**

Responsibilities:

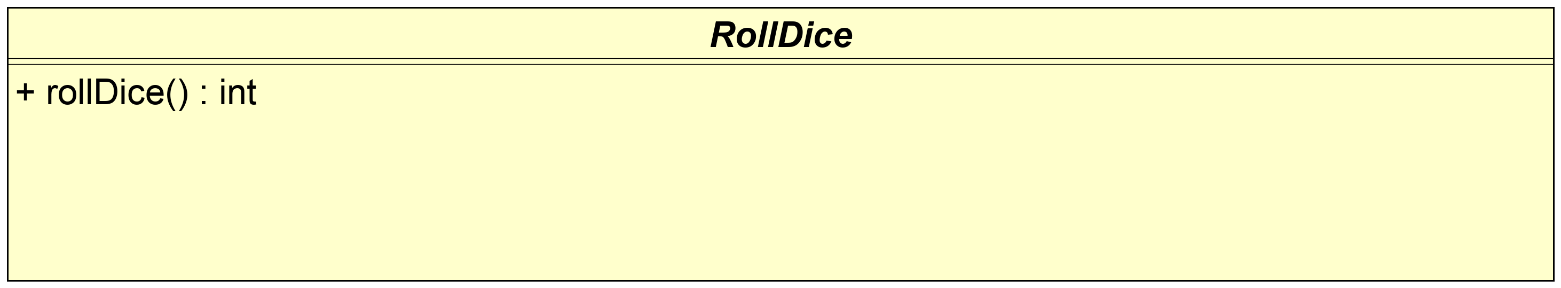
* This abstract class includes a fully implemented method that moves piece a piece if possible.
* Other methods include checking if a contemplated move is valid and checking for existamce of any valid moves.



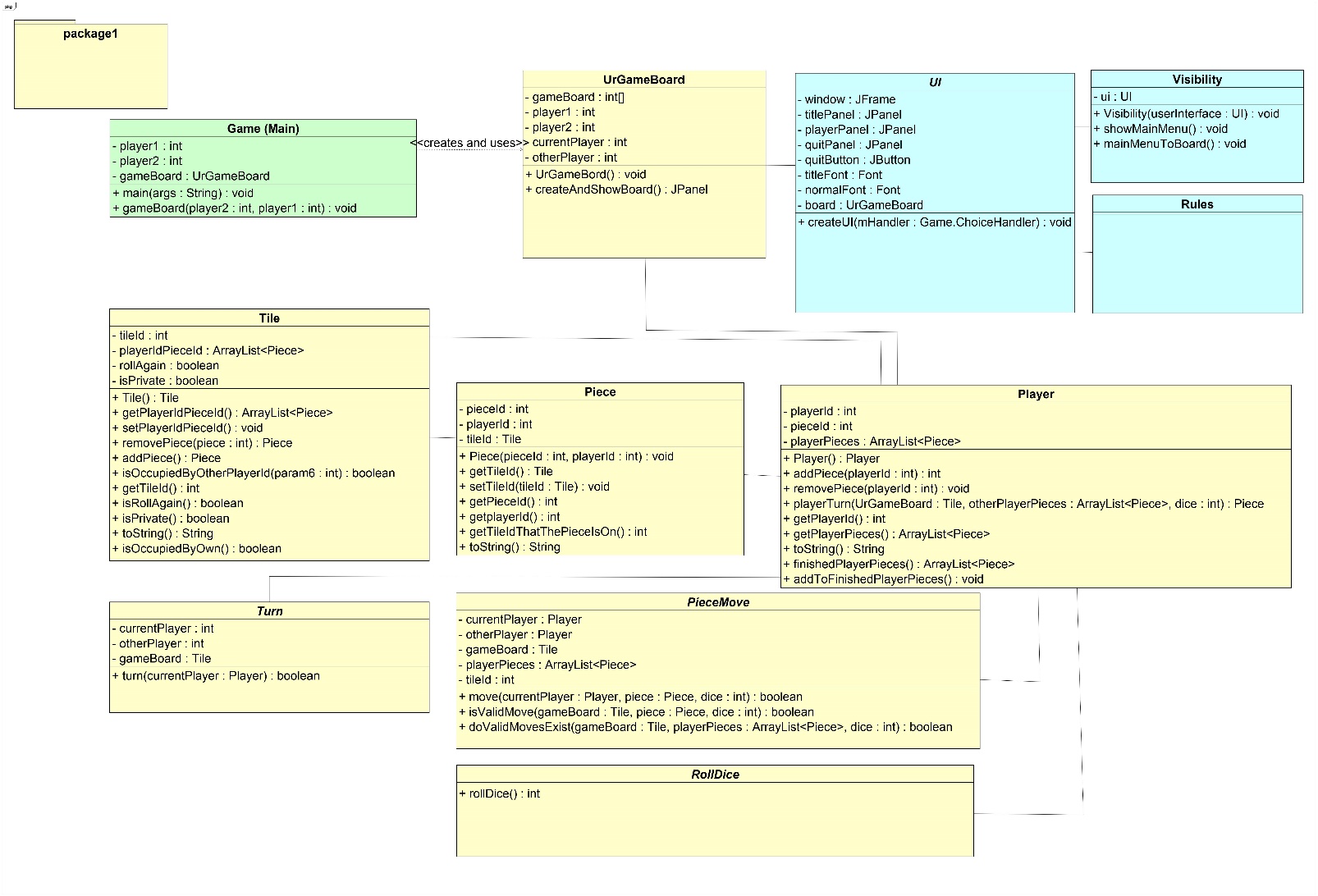
**Class name:** **RollDice**

Responsibilities:

* This abstract class includes a fully implemented method that simulates the roll of 4 dice.



**Section 4: UML Astra class diagram showing the classes and depicting the relationships among the classes. We used the free student registration and license**



**Section 5: Design of the** **main and major algorithms (see code files)**

**Section 6: Summary of test plan and List of test cases: the data and expected results used to verify the final product is working as expected**

**Summary of the JUnit-base Test Plan**

We plan to use JUnit tests to test our Game of Ur application.

The JUnit test data values include good values, bad values, and boundary values.

The test plan includes a tabulation of values or results that we expect to get (asserted expected results).

Finally, the actual values or results that the coded methods return will be compared with the asserted expected values or results to verify a passed test.

Based on our latest UML class diagram, we will prepare JUnit testing for the following classes:

UI (GUI-based User Interface)

Visibility (Display the Main GUI-based Menu)

UrGameBoard

Player

Tile

Piece

Turn

RollDice

PieceMove

Game (Start Main Driver)

A general definition of unit testing is a piece of code written that executes a specific functionality in the code to be tested and asserts a certain behavior or state of a small unit of code, such as a method or a class.

More formally, “A JUnit test is a method contained in a class which is only used for testing. This is called a Test class. ... You use an assert method, provided by JUnit or another assert framework, to check an expected result versus the actual result.”

Source: <https://www.vogella.com/tutorials/JUnit/article.html>

Our test plan is to use the following JUnit test methods to test our code classes and methods to affirm that actual values and object types generated by the code equal the expected values and object types:

assertArrayEquals()

assertEquals()

assertTrue(), assertFalse()

assertNull(), assertNotNull()

assertSame(), assertNotSame()

An example of the test code is:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testConcatenate() {

MyUnit myUnit = new MyUnit();

String result = myUnit.concatenate("one", "two");

assertEquals("onetwo", result);

}

}

List of Test Cases (as part of test-driven development):

We did not have time to complete formal testing .