CS 319

Object- Oriented Software Engineering

Design Report

IQ Puzzler Pro

Unknown - Group 1-I

Derviş Mehmed Barutçu

Elif Beril Şaylı

Giray Baha Kezer

Zeynep Ayça Çam

Zeynep Hande Arpakuş

Table of Contents

1.Introduction

IQPuzzlerPro is a puzzle game that consists of 12 different pieces, and 3 different gameplay options: blackboard, grey board, and 3D pyramid option. The players should follow the instructions to fill the level requirements and they should fill the board with rest pieces or construct the pyramid by using pieces.

1.1. Purpose of the System

IQPuzzlerPro is a combined mind game with its 2D and 3D extensions. Its design was implemented in a way to broaden your thinking. Users could get higher satisfaction if they pass the levels step by step from beginner to expert. Iq Puzzler Pro is thought to be created as a user-friendly and challenging game which consists of different boards and different paths. The player’s target is to accomplish level requirements in his way. His way should be on stage so as to do what related level requires, also there need to be faster time counts for the scoreboard. Except for the game’s high performance it also offers user-friendly and pleasing UI.

1.2. Design Goals

The design is another important point in order to create the system, it is much helpful for guiding what to focus on that system. There are some lower efficient requirements as we wrote in our analysis report they are our focus objects. Following steps will be part of our design goals.

1.2.1 Criteria

End User Criteria

Usability:

IQPuzzlerPro is a form that combination of entertainment and mind tricks in 2D and 3D space. It can be seen as a much more harder form of 3D tetris on a flat board except the players select the piece and put them on each other to finalize pyramid shape. Our game will provide users user-friendly, funny interface, related noisy sounds for incorrect tries so it is funny to play and easy to use the game. For instance, imagine an understandable game home menu that shows how to play, settings, directly music on-off buttons on the settings button. In the game panel rotations can be seen by users, thus they don't have to imagine how to rotate in that piece in that way or in that way it is an easier rotation for users. The section levels can be chosen easily by the user before the game beginning. The rotates and picked suitable holes can be seen on game illustration synchronously while playing with pieces and board holes. Ease of Learning for how to play can be counted as another ease of use for users since pictures for different boards and pieces can be seen before the game via that button. The user can use a keyboard for the exit for directly quit button for quitting easily.

Performance:

Performance could be seen as another important design goals either, which is necessary for games itself. We are using JavaFX library for our game’s 3D parts with JAVA source codes on NetBeans 8.2 environment. Thanks to using this programming language and it's library support we can have better platform performance and ease of use. We will use JAVA codes to program our game by using JavaFX in 2D and 3D modes.

Maintenance Criteria

Extendibility:

The game design provided for our game we will contact the users via e-mails related their valuable feedbacks to progress the game. We write our e-mail addresses on the game credit part to be easily connected them. Also, we could advance the levels and pieces in difficulty for expert game users if they demand.

Modifiability:

The IQPuzzlerPro game is designed for both multi or single gamers. Two game-playing modes are offered them on different boards and levels. These features could be easily modified for different levels or different volume pieces. Thereby, thanks to extensions or directly changes our game can be modified without putting so much effort on it.

Reusability:

There are few subsystems in our game, which can be integrated into other different games with logic. For example, our game can be seen as 3D flat board with rotating and selecting but not falling pieces. It means our game logic and features can be integrated some part of likely tetris games. These kinds platforms and rotates and pieces selection games or directly mind trick games can be used for integration our features.

1.3 Definitions, Acronyms and Abbreviations

User Interface (UI): UI includes features in the application which directly interact with the user.

Model View Controller (MVC): MVC is a design pattern that separates an application to tree main domain: model, view and controller.

JavaScript Object Notation (JSON) : JSON is syntax to store and exchange data.

JavaFX: JavaFX is a software platform which is used for creating software applications.

2. High-Level Software Architecture

This section is created to demonstrate software architecture of the application. Subsections are related to subsystem decomposition, hardware/software mapping, software and hardware features, persistent data management, access control, and security and finally, boundary conditions.

2.1 Subsystem Decomposition

In this section, we will decompose our system into subsystems. During the decomposition of our system, we decide to use the MVC design pattern and we decompose our subsystems according to that design pattern. We decide to use MVC pattern since it fits our game and it provides an easy change in UI in the future. We have 3 subsystems. These are Game Management (controller), User Interface (view) and Models (model) subsystems. By dividing subsystems like this, we want to achieve flexibility in the future.

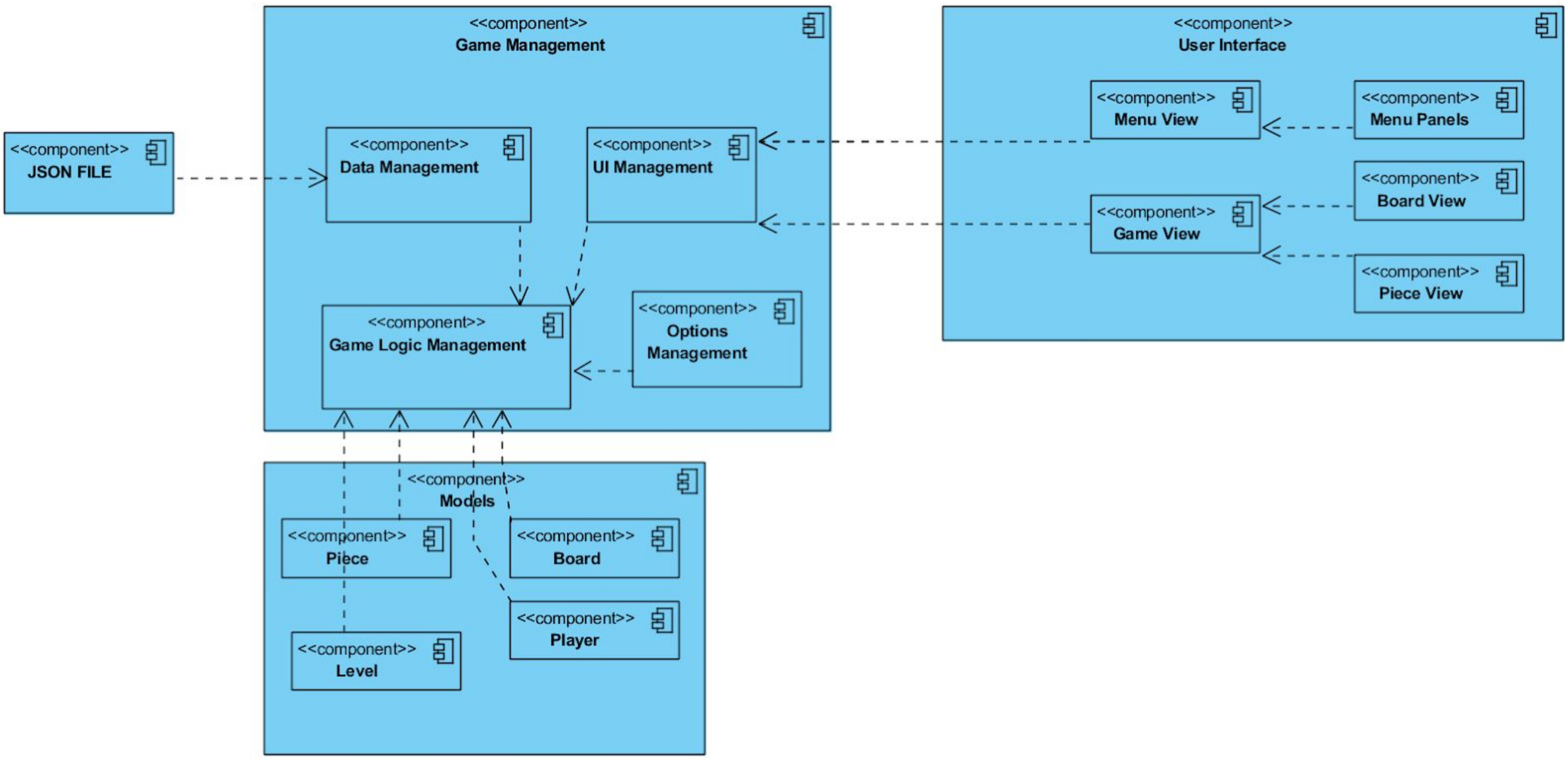


Figure 1 : Deployment Diagram of IQPuzzlerPro

Game Management Subsystem consists of controller classes for our game. It consists of data management, UI management, and game logic management. Data management class basically handle with interaction game between UI and data. UI management class controls view of all game, includes the view of menu panels and game objects. Game logic controls all functionalities related to game objects which are pieces and board.

User Interface Subsystem consists of view classes for our game. It includes the view of game objects, which are piece and board, it also includes the view of menu panels.

Model Subsystem consists of model classes for our game. It includes models of the piece, board, player, and level.

2.2 Hardware / Software Mapping

2.2.1 Software

Our project is being developed in Java programming language. We are using NetBeans 8.2 environment and JavaFX libraries[1] to program and create UI and game interfaces, thereby we decided to use Java.

2.2.2 Hardware

As for hardware, the player uses the mouse for selecting, dragging, placing, rotating and flipping. As to the computer part, most of the computers should run our game because it does not require high specification.

2.3 Persistent Data Management

Although our program doesn't require any complex data systems, our program keeps players data, which are players names, time, moves, and scores of that player. To store this kind of data which are simple data structures and objects, we will use JSON file.

2.4 Access Control and Security

Our project does not require an internet connection so outside of that computer, nobody can change our game data or specifications of levels. As some of the necessary variables were defined as constant or private, other classes cannot see or modify that variable. In addition, our project doesn't hold sensitive information of the user, thus we provide some access control by creating some variables and functions as private.

2.5 Boundary Conditions

Our game has an executable jar and will be executed from .jar. This provides the game to portability. The game can be transfer from a computer to others so it is ready to play. To start the game, the user has to register system by providing his/her nickname data.

To startup game, the level class takes data from JSON file to create game's fundamentals.The game can be terminated when the user click the "Exit" button in the main menu. If the user wants to quit during playing game, firstly, user pause the game and return the main menu. Then, the user can terminate the game by clicking the "Exit" button.Also, the game can be terminated when the user clicks "X" button which is on the right top of the frame. Moreover, if the time exceeds one hour , the game finishes automatically.

When the user accomplishes one level, game store level's information. If game collapses during one level, the whole level's data won't be lost. The accomplished last level's data remains and can be taken from JSON file.

If the user accomplishes all levels and completes the game, the game return to the main menu again.If one subsystem terminates, other subsystems can affect and terminate.

3. Subsystem Services

In this section, we show general representation of subsystems. They are user interface layer, models layer and game management layer.

3.1 User Interface Layer

Our User Interface package has BoardViewerPanel, HowToPlayPanel, LeaderPanel, SettingPanel, MenuPanel, PanelViewer, BoardSelectionPanel, ModeSelectionPanel classes inside. These panels are for the essence of that particular screens and they are replaced by the PanelViewer class.

PanelViewer class basically takes the current panel and replace it with the new panel when it is necessary.The difference between BoardSelectionPanel and BoardViewerPanel is that BoardSelectionPanel is for choosing the between three boards, black, grey, and pyramid. According to selection in this panel game will start accordingly. The BoardViewerPanel is the creating the board for the game, chosen in BoardSelectionPanel.

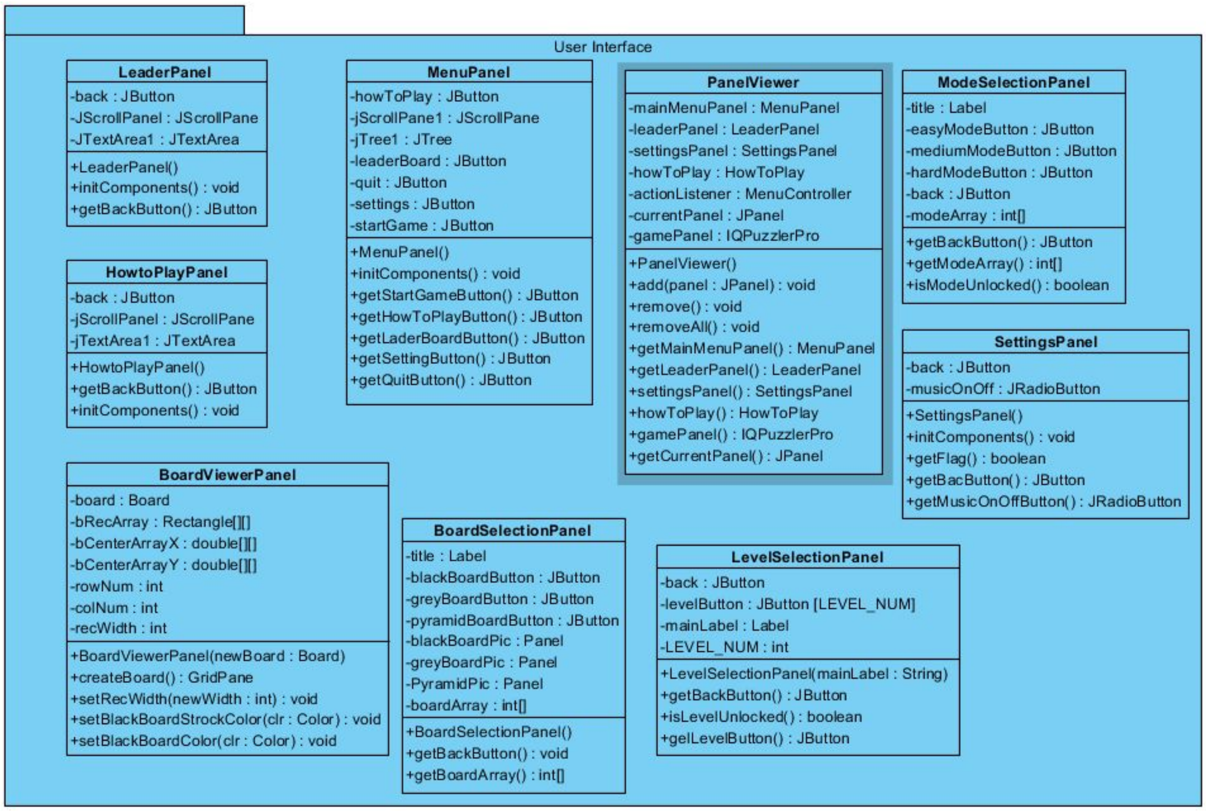

User Interface
LeaderPanel
-back : JButton
-JScrollPanel : JScrollPane
-JTextArea1 : JTextArea
+LeaderPanel()
+initComponents() : void
+getBackButton() : JButton
HowtoPlayPanel
-back : JButton
-jScrollPanel : JScrollPane
-jTextArea1 : JTextArea
+HowtoPlayPanel()
+getBackButton() : JButton
+initComponents() : void
MenuPanel
-howToPlay : JButton
-jScrollPane1 : JScrollPane
-jTree1 : JTree
-leaderBoard : JButton
-quit : JButton
-settings : JButton
-startGame : JButton
+MenuPanel()
+initComponents() : void
+getStartGameButton() : JButton
+getHowToPlayButton() : JButton
+getLaderBoardButton() : JButton
+getSettingButton() : JButton
+getQuitButton() : JButton
PanelViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-actionListener : MenuController
-currentPanel : JPanel
-gamePanel : IQPuzzlerPro
+PanelViewer()
+add(panel : JPanel) : void
+remove() : void
+removeAll() : void
+getMainMenuPanel() : MenuPanel
+getLeaderPanel() : LeaderPanel
+settingsPanel() : SettingsPanel
+howToPlay() : HowToPlay
+gamePanel() : IQPuzzlerPro
+getCurrentPanel() : JPanel
ModeSelectionPanel
-title : Label
-easyModeButton : JButton
-mediumModeButton : JButton
-hardModeButton : JButton
-back : JButton
-modeArray : int[]
+getBackButton() : JButton
+getModeArray() : int[]
+isModeUnlocked() : boolean
SettingsPanel
-back : JButton
-musicOnOff : JRadioButton
+SettingsPanel()
+initComponents() : void
+getFlag() : boolean
+getBacButton() : JButton
+getMusicOnOffButton() : JRadioButton
BoardViewerPanel
-board : Board
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-rowNum : int
-colNum : int
-recWidth : int
+BoardViewerPanel(newBoard : Board)
+createBoard() : GridPane
+setRecWidth(newWidth : int) : void
+setBlackBoardStrockColor(clr : Color) : void
+setBlackBoardColor(clr : Color) : void
BoardSelectionPanel
-title : Label
-blackBoardButton : JButton
-greyBoardButton : JButton
-pyramidBoardButton : JButton
-blackBoardPic : Panel
-greyBoardPic : Panel
-PyramidPic : Panel
-boardArray : int[]
+BoardSelectionPanel()
+getBackButton() : void
+getBoardArray() : int[]
LevelSelectionPanel
-back : JButton
-levelButton : JButton [LEVEL_NUM]
-mainLabel : Label
-LEVEL_NUM : int
+LevelSelectionPanel(mainLabel : String)
+getBackButton() : JButton
+isLevelUnlocked() : boolean
+gelLevelButton() : JButton

Figure 2 : Diagram of user interface layer

3.2 Models Layer

Our model package has board, level, player and pieces classes inside. Board class has the column, row numbers, rectangle width and arrays with pieces coordinates and total arrays to compare completed or not. The level class has id and create level operation inside to integrate level to player selection. Player class holds names, time, scores inside for leaderboards, also it is usable for other classes thanks to getter and setter operations inside. Piece class has coordinates and shapes as attributes and its operations can be used for other classes, they can take pieces shape and can detect intersections and create pieces.

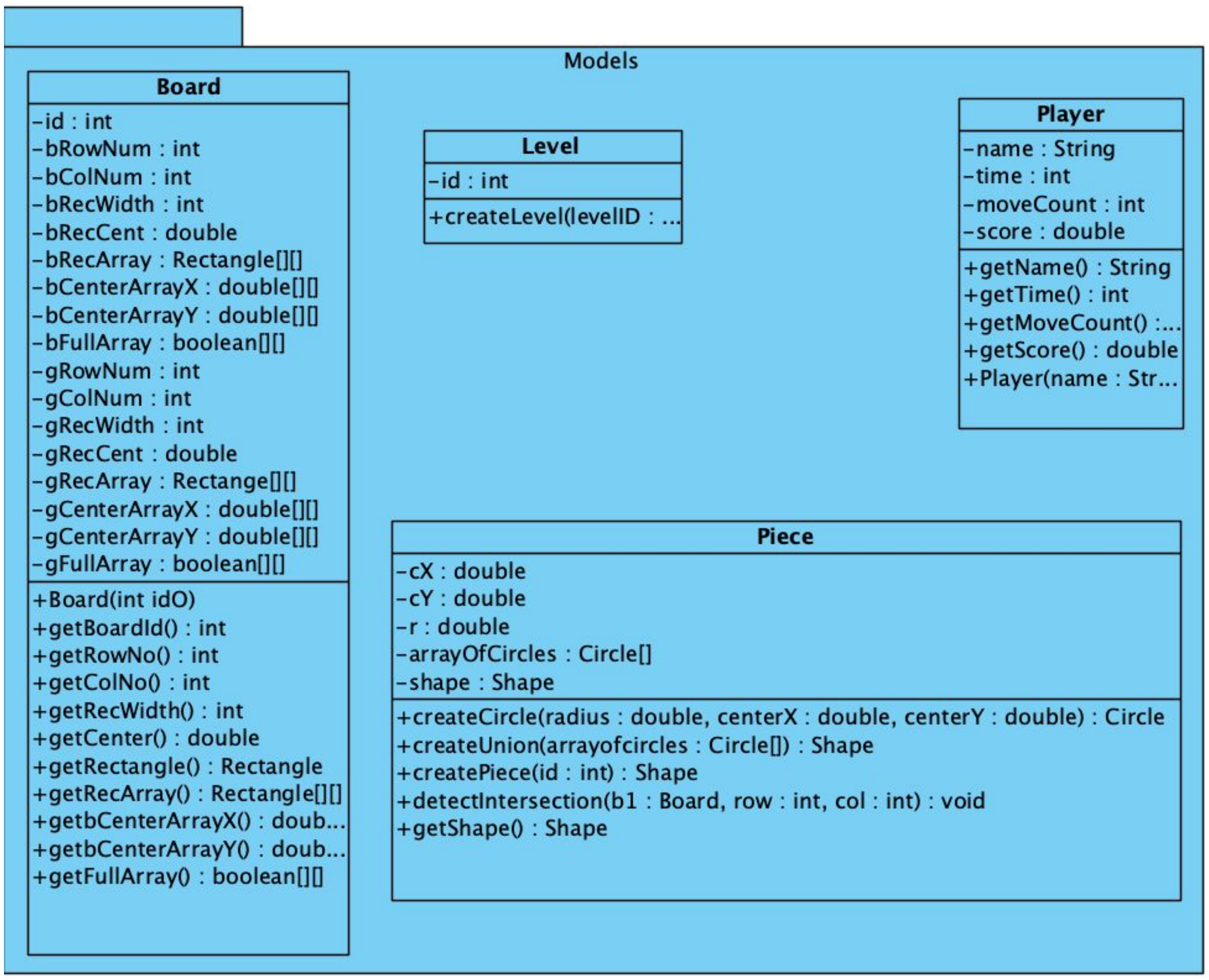

Models
Board
-id : int
-bRowNum : int
-bColNum : int
-bRecWidth : int
-bRecCent : double
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-bFullArray : boolean[][]
-gRowNum : int
-gColNum : int
-gRecWidth : int
-gRecCent : double
-gRecArray : Rectange[][]
-gCenterArrayX : double[][]
-gCenterArrayY : double[][]
-gFullArray : boolean[][]
+Board(int idO)
+getBoardId() : int
+getRowNo() : int
+getColNo() : int
+getRecWidth() : int
+getCenter() : double
+getRectangle() : Rectangle
+getRecArray() : Rectangle[][]
+getbCenterArrayX() : doub...
+getbCenterArrayY() : doub...
+getFullArray() : boolean[][]
Level
-id : int
+createLevel(levelID : ...
Player
-name : String
-time : int
-moveCount : int
-score : double
+getName() : String
+getTime() : int
+getMoveCount() :...
+getScore() : double
+Player(name : Str...
Piece
-cX : double
-cY : double
-r : double
-arrayOfCircles : Circle[]
-shape : Shape
+createCircle(radius : double, centerX : double, centerY : double) : Circle
+createUnion(arrayofcircles : Circle[]) : Shape
+createPiece(id : int) : Shape
+detectIntersection(b1 : Board, row : int, col : int) : void
+getShape() : Shape

Figure 3 : Diagram of models layer

3.3 Game Management Layer

In our game management package, we have GameManager, SinglePlayerGame, MultiPlayerGame, SoundManager, Leaderboard, MenuController and Pieces classes inside. The player could change music, turned on or off functions thanks to that class. Pieces class is the controller of the Piece class, moving of pieces, flipping, and rotation of them. Time manager class has time elements time is updated there. Multi or single player game classes provides player different game modes, with level, boards, and difficulties levels. Leaderboard class has player object and player based leaderboard list functions as adding and removing them into the board. Related their scores and challenge time they are placed on the leaderboard list.

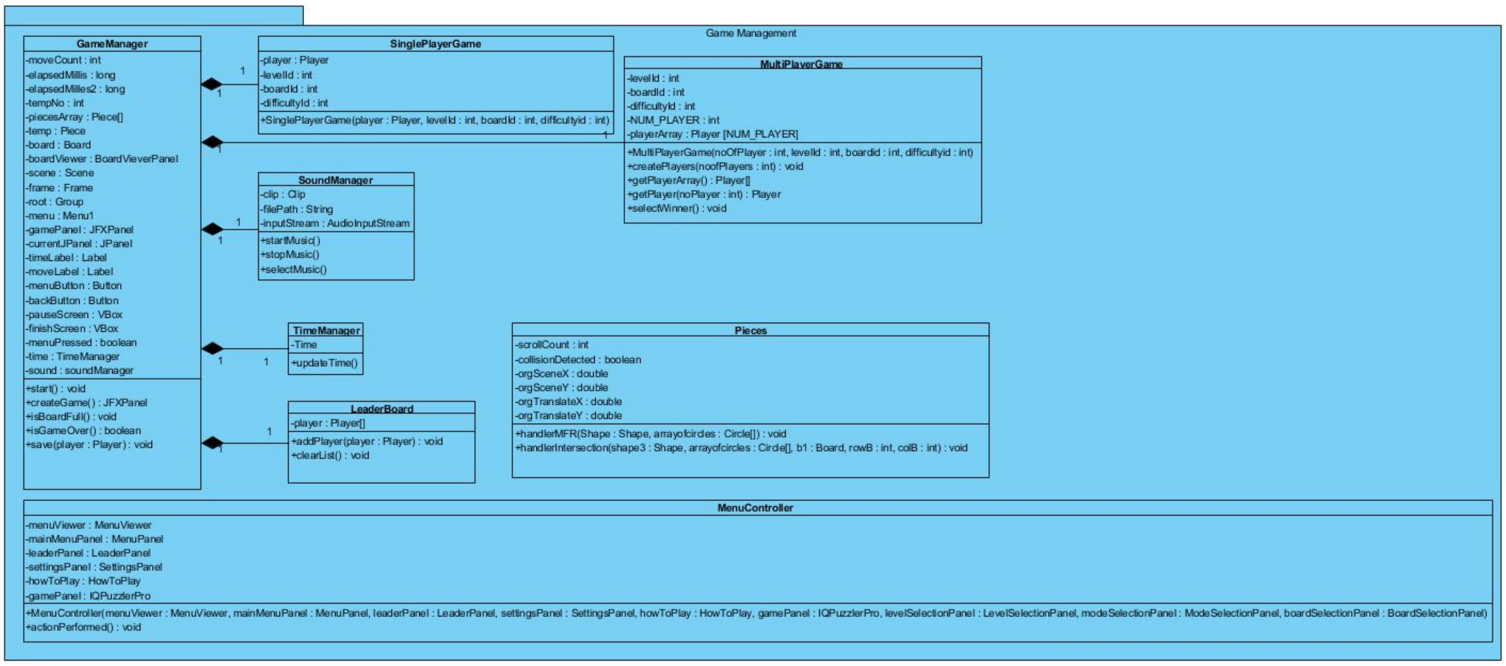

Game Management
GameManager
-moveCount : int
-elapsedMillis : long
-elapsedMilles2 : long
-tempNo : int
-piecesArray : Piece[]
-temp : Piece
-board : Board
-boardViewer : BoardVieverPanel
-scene : Scene
-frame : Frame
-root : Group
-menu : Menu1
-gamePanel : JFXPanel
-currentJPanel : JPanel
-timeLabel : Label
-moveLabel : Label
-menuButton : Button
-backButton : Button
-pauseScreen : VBox
-finishScreen : VBox
-menuPressed : boolean
-time : TimeManager
-sound : soundManager
+start() : void
+createGame() : JFXPanel
+isBoardFull() : void
+isGameOver() : boolean
+save(player : Player) : void
SinglePlayerGame
-player : Player
-levelId : int
-boardId : int
-difficultyId : int
+SinglePlayerGame(player : Player, levelId : int, boardId : int, difficultyid : int)
MultiPlayerGame
-levelId : int
-boardId : int
-difficultyId : int
-NUM_PLAYER : int
-playerArray : Player [NUM_PLAYER]
+MultiPlayerGame(noOfPlayer : int, levelId : int, boardid : int, difficultyid : int)
+createPlayers(noofPlayers : int) : void
+getPlayerArray() : Player[]
+getPlayer(noPlayer : int) : Player
+selectWinner() : void
SoundManager
-clip : Clip
-filePath : String
-inputStream : AudioInputStream
+startMusic()
+stopMusic()
+selectMusic()
TimeManager
-Time
+updateTime()
LeaderBoard
-player : Player[]
+addPlayer(player : Player) : void
+clearList() : void
Pieces
-scrollCount : int
-collisionDetected : boolean
-orgSceneX : double
-orgSceneY : double
-orgTranslateX : double
-orgTranslateY : double
+handlerMFR(Shape : Shape, arrayofcircles : Circle[]) : void
+handlerIntersection(shape3 : Shape, arrayofcircles : Circle[], b1 : Board, rowB : int, colB : int) : void
MenuController
-menuViewer : MenuViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-gamePanel : IQPuzzlerPro
+MenuController(menuViewer : MenuViewer, mainMenuPanel : MenuPanel, leaderPanel : LeaderPanel, settingsPanel : SettingsPanel, howToPlay : HowToPlay, gamePanel : IQPuzzlerPro, levelSelectionPanel : LevelSelectionPanel, modeSelectionPanel : ModeSelectionPanel, boardSelectionPanel : BoardSelectionPanel)
+actionPerformed() : void
1
1

Figure 4 : Diagram of game management layer

4. Low-Level Design

4.1 Object Design Trade-Offs

Usability vs Functionality :

The purpose of the system is to entertain the user so we prefer to increase usability rather than functionality. The game has a simple interface and basic instructions rather than complex menu choices and various features. Therefore, we had to sacrifice some functionalities to be more usable. Our game has the most 4 steps to play game and in-game, which are first they have to select if they want to play singleplayer or multiplayer, then they choose difficulty, board and lastly which level they want to play. There are few functions about game logic and these are described as “How to Play” section and video which demonstrates gameplay. The user can reach them before he or she starts the game.

Performance vs. Memory:

We choose Java programming language to use JavaFX library for our game structures. It is created on the platform with JAVA source codes on the NetBeans environment, also through its dynamic, there is no unnecessary memory allocation to avoid time complexities and lower efficiency.

We avoid using unnecessary attributes to prevent unnecessary memory allocations. Performance is important for us because it makes the game playable and increases usability. Throughout the whole coding session, we avoid big time complexities.

We store the necessary information in JSON files, it does not take much space to store and the players will not be affected by the size of the game.

Development Time vs User Experience :

For game development, graphics are very important to make the game more attractive. As it was mentioned before, we decided to use JavaFX because its libraries use 32 bits for resolution so JavaFX provides better resolution than C++ and

Java. In fact, JavaFX is intended to replace Swing as the standard GUI library for Java SE [2].

It is harder for us because although the team has experience with Java libraries, JavaFX is completely new for us. Also, JavaFX resources are quite limited, especially for 3D. Therefore, getting familiar to these libraries increases development time but with better graphics, we facilitate user experience.

4.2 Final object design

The figure below, it basically demonstrate all system's class diagrams and how they are related with each other. In the following sections, all classes and their attributes, functions and constructors are explained.

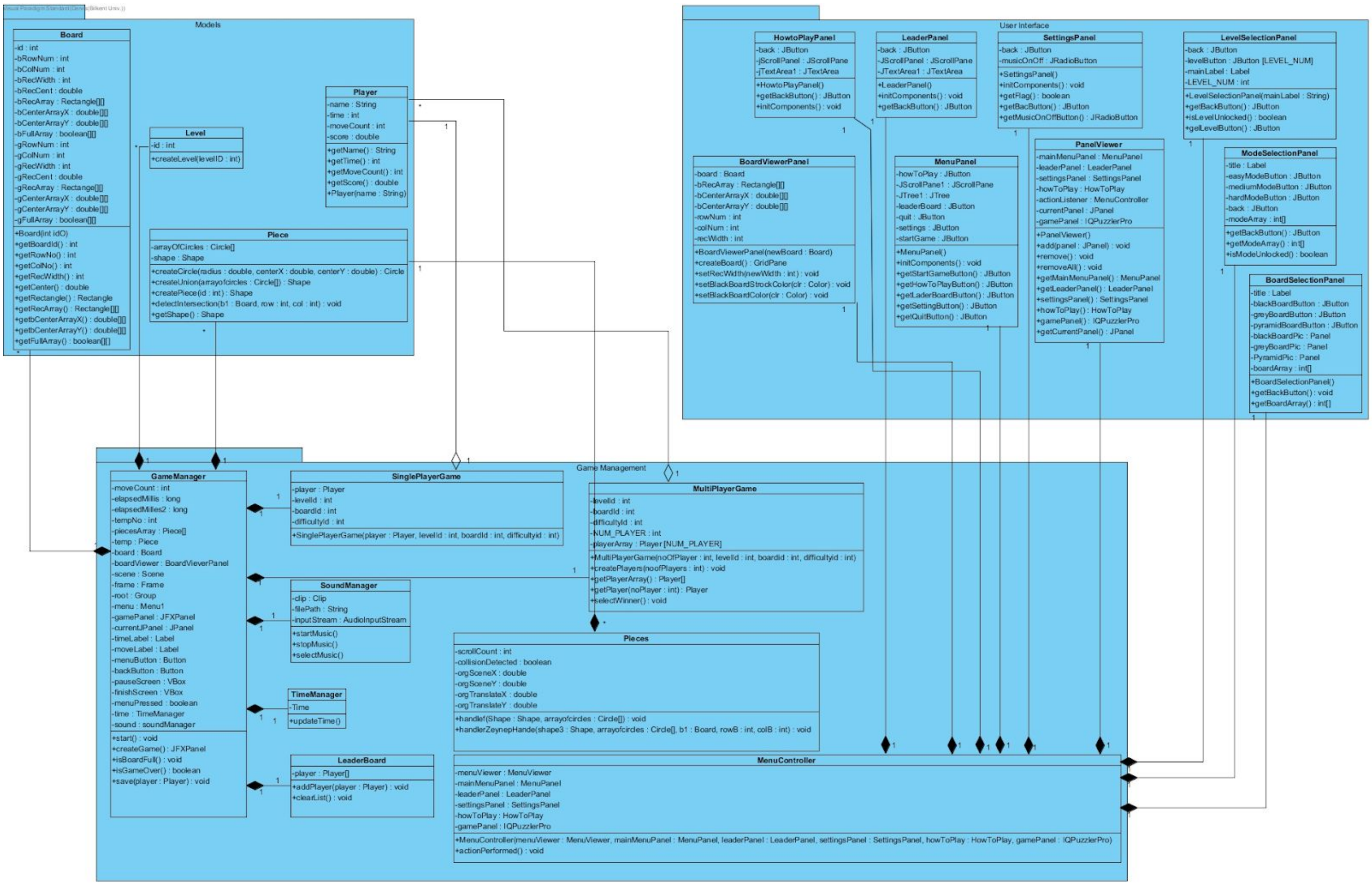

Models
Board
-id : int
-bRowNum : int
-bColNum : int
-bRecWidth : int
-bRecCent : double
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-bFullArray : boolean[][]
-gRowNum : int
-gColNum : int
-gRecWidth : int
-gRecCent : double
-gRecArray : Rectange[][]
-gCenterArrayX : double[][]
-gCenterArrayY : double[][]
-gFullArray : boolean[][]
+Board(int idO)
+getBoardId() : int
+getRowNo() : int
+getColNo() : int
+getRecWidth() : int
+getCenter() : double
+getRectangle() : Rectangle
+getRecArray() : Rectangle[][]
+getbCenterArrayX() : double[][]
+getbCenterArrayY() : double[][]
+getFullArray() : boolean[][]
Level
-id : int
+createLevel(levelID : int)
Player
-name : String
-time : int
-moveCount : int
-score : double
+getName() : String
+getTime() : int
+getMoveCount() : int
+getScore() : double
+Player(name : String)
Piece
-arrayOfCircles : Circle[]
-shape : Shape
+createCircle(radius : double, centerX : double, centerY : double) : Circle
+createUnion(arrayofcircles : Circle[]) : Shape
+createPiece(id : int) : Shape
+detectIntersection(b1 : Board, row : int, col : int) : void
+getShape() : Shape
User Interface
HowtoPlayPanel
-back : JButton
-jScrollPanel : JScrollPane
-jTextArea1 : JTextArea
+HowtoPlayPanel()
+getBackButton() : JButton
+initComponents() : void
LeaderPanel
-back : JButton
-JScrollPanel : JScrollPane
-JTextArea1 : JTextArea
+LeaderPanel()
+initComponents() : void
+getBackButton() : JButton
SettingsPanel
-back : JButton
-musicOnOff : JRadioButton
+SettingsPanel()
+initComponents() : void
+getFlag() : boolean
+getBacButton() : JButton
+getMusicOnOffButton() : JRadioButton
LevelSelectionPanel
-back : JButton
-levelButton : JButton [LEVEL_NUM]
-mainLabel : Label
-LEVEL_NUM : int
+LevelSelectionPanel(mainLabel : String)
+getBackButton() : JButton
+isLevelUnlocked() : boolean
+gelLevelButton() : JButton
BoardViewerPanel
-board : Board
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-rowNum : int
-colNum : int
-recWidth : int
+BoardViewerPanel(newBoard : Board)
+createBoard() : GridPane
+setRecWidth(newWidth : int) : void
+setBlackBoardStrockColor(clr : Color) : void
+setBlackBoardColor(clr : Color) : void
MenuPanel
-howToPlay : JButton
-JScrollPane1 : JScrollPane
-JTree1 : JTree
-leaderBoard : JButton
-quit : JButton
-settings : JButton
-startGame : JButton
+MenuPanel()
+initComponents() : void
+getStartGameButton() : JButton
+getHowToPlayButton() : JButton
+getLaderBoardButton() : JButton
+getSettingButton() : JButton
+getQuitButton() : JButton
PanelViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-actionListener : MenuController
-currentPanel : JPanel
-gamePanel : IQPuzzlerPro
+PanelViewer()
+add(panel : JPanel) : void
+remove() : void
+removeAll() : void
+getMainMenuPanel() : MenuPanel
+getLeaderPanel() : LeaderPanel
+settingsPanel() : SettingsPanel
+howToPlay() : HowToPlay
+gamePanel() : IQPuzzlerPro
+getCurrentPanel() : JPanel
ModeSelectionPanel
-title : Label
-easyModeButton : JButton
-mediumModeButton : JButton
-hardModeButton : JButton
-back : JButton
-modeArray : int[]
+getBackButton() : JButton
+getModeArray() : int[]
+isModeUnlocked() : boolean
BoardSelectionPanel
-title : Label
-blackBoardButton : JButton
-greyBoardButton : JButton
-pyramidBoardButton : JButton
-blackBoardPic : Panel
-greyBoardPic : Panel
-PyramidPic : Panel
-boardArray : int[]
+BoardSelectionPanel()
+getBackButton() : void
+getBoardArray() : int[]
Game Management
GameManager
-moveCount : int
-elapsedMillis : long
-elapsedMilles2 : long
-tempNo : int
-piecesArray : Piece[]
-temp : Piece
-board : Board
-boardViewer : BoardVieverPanel
-scene : Scene
-frame : Frame
-root : Group
-menu : Menu1
-gamePanel : JFXPanel
-currentJPanel : JPanel
-timeLabel : Label
-moveLabel : Label
-menuButton : Button
-backButton : Button
-pauseScreen : VBox
-finishScreen : VBox
-menuPressed : boolean
-time : TimeManager
-sound : soundManager
+start() : void
+createGame() : JFXPanel
+isBoardFull() : void
+isGameOver() : boolean
+save(player : Player) : void
SinglePlayerGame
-player : Player
-levelId : int
-boardId : int
-difficultyId : int
+SinglePlayerGame(player : Player, levelId : int, boardId : int, difficultyid : int)
MultiPlayerGame
-levelId : int
-boardId : int
-difficultyId : int
-NUM_PLAYER : int
-playerArray : Player [NUM_PLAYER]
+MultiPlayerGame(noOfPlayer : int, levelId : int, boardid : int, difficultyid : int)
+createPlayers(noofPlayers : int) : void
+getPlayerArray() : Player[]
+getPlayer(noPlayer : int) : Player
+selectWinner() : void
SoundManager
-clip : Clip
-filePath : String
-inputStream : AudioInputStream
+startMusic()
+stopMusic()
+selectMusic()
TimeManager
-Time
+updateTime()
Pieces
-scrollCount : int
-collisionDetected : boolean
-orgSceneX : double
-orgSceneY : double
-orgTranslateX : double
-orgTranslateY : double
+handlef(Shape : Shape, arrayofcircles : Circle[]) : void
+handlerZeynepHande(shape3 : Shape, arrayofcircles : Circle[], b1 : Board, rowB : int, colB : int) : void
LeaderBoard
-player : Player[]
+addPlayer(player : Player) : void
+clearList() : void
MenuController
-menuViewer : MenuViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-gamePanel : IQPuzzlerPro
+MenuController(menuViewer : MenuViewer, mainMenuPanel : MenuPanel, leaderPanel : LeaderPanel, settingsPanel : SettingsPanel, howToPlay : HowToPlay, gamePanel : IQPuzzlerPro)
+actionPerformed() : void

Figure 5 : All class diagrams of IQPuzzlerPro

Game Manager Class:

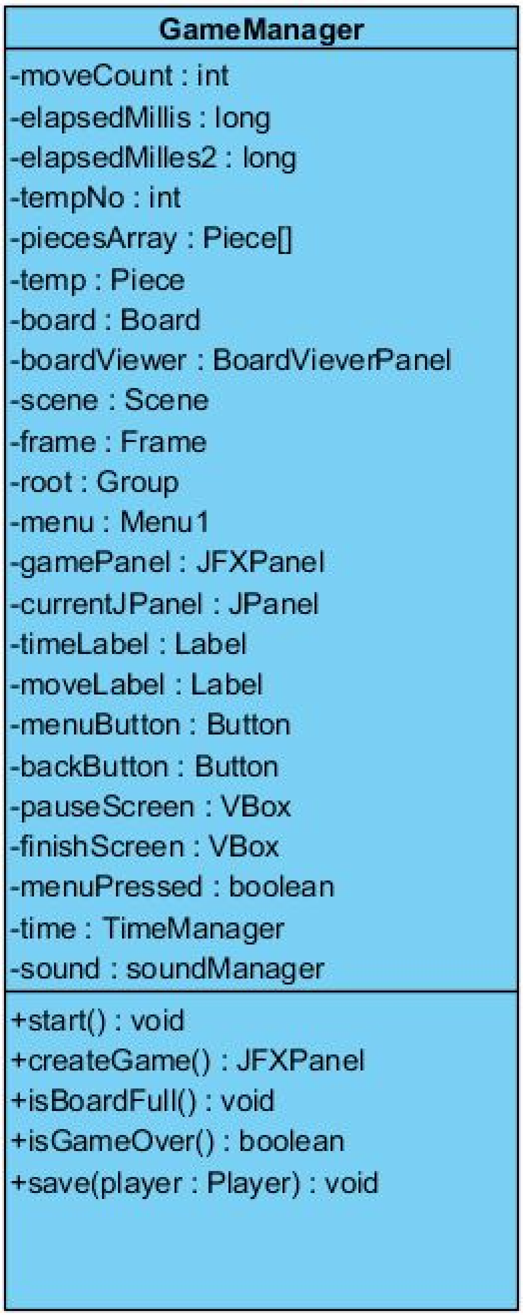


Figure 6 : Game Manager Class

This class is the main class of the game. It takes two parameters in the constructor. By looking these parameters it determines game objects and images. It constructs the level, provide functionalities to the user such as move and place the piece, pause and quit the game. This class controls the main functionalities of the game.

Attributes:

1. **moveCount**: This integer attribute tracks of the count of the users move the pieces. It is increased each time user move and place a piece on the board.
2. **elapsedMillis:** This long attribute is to represent the elapsed time since the beginning of the game until the finish.
3. **elpasedMillis2:** This long attribute represents the elapsed time when the user stop the game.
4. **tempNo:** This integer attribute represents the array index of the selected piece.
5. **piecesArray:** This array keeps the pieces.
6. **boardViewer:** This panel attribute represents the selected board.
7. **scene:** This attribute which belongs to Scene class represents the game scene.
8. **frame:** This frame attribute represents the frame of the game.
9. **root:** This group attribute is a variable of the scene.
10. **menu:** This attribute represents the created menu for the opening screen of the game.
11. **gamePanel:** This attribute' type is JFXPanel represents the panel which include the game.
12. **currentJPanel:** This attribute represents the current panel of the game.
13. **timeLabel:** This label is to show the elapsed time the user.
14. **moveLabel:** This attribute is to show the move count the user.
15. **menuButton:** This button attribute is to show the menu during the game.
16. **backButton:** This button attribute to go back from the game.
17. **pauseScreen:** This screen attribute represents the screen which is shown when the user paused the level.
18. **finishScreen:** This attribute represents the screen which is shown when the user finish the level.
19. **menuPressed:** This boolean attribute represents the menu button is pressed or not.
20. **time:** This attribute represents the time.
21. **sound:** This attribute represents the background sound

22. **board**: This attribute is for one of the game objects. It provides one board object and its functions.

Methods:

1. **start():** This method is for starting the creating the game.
2. **createGame():** This method creates the game scene with board, pieces, labels and buttons.
3. **isBoardFull():** This method is for checking the board of the game is full or not.
4. **isGameOver():** This method was written to check the game is finish or not.
5. **save(Player player):** This method was written to save the players' score with his/her name if it is wanted.

Board Class:

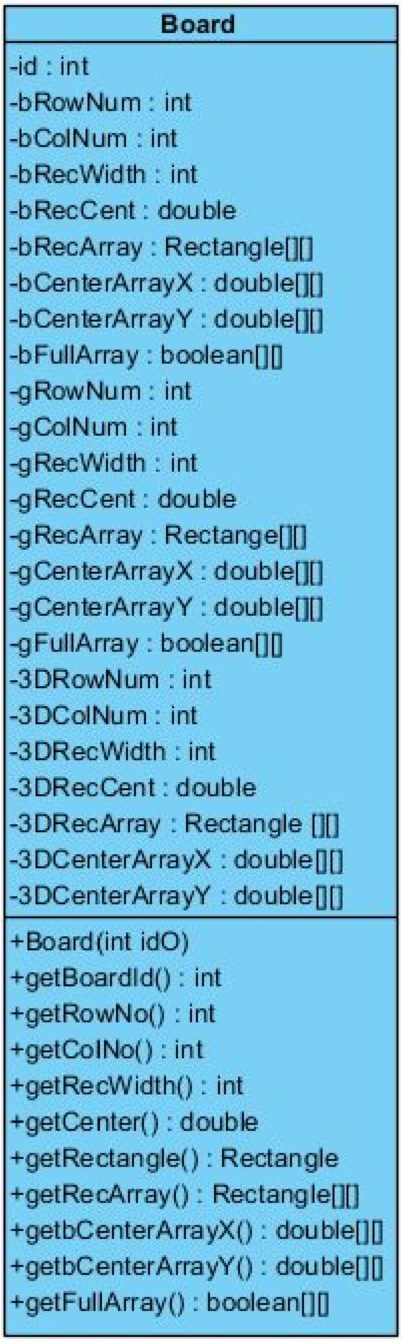

Board
-id : int
-bRowNum : int
-bColNum : int
-bRecWidth : int
-bRecCent : double
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-bFullArray : boolean[][]
-gRowNum : int
-gColNum : int
-gRecWidth : int
-gRecCent : double
-gRecArray : Rectange[][]
-gCenterArrayX : double[][]
-gCenterArrayY : double[][]
-gFullArray : boolean[][]
-3DRowNum : int
-3DColNum : int
-3DRecWidth : int
-3DRecCent : double
-3DRecArray : Rectangle [][]
-3DCenterArrayX : double[][]
-3DCenterArrayY : double[][]
+Board(int idO)
+getBoardId() : int
+getRowNo() : int
+getColNo() : int
+getRecWidth() : int
+getCenter() : double
+getRectangle() : Rectangle
+getRecArray() : Rectangle[][]
+getbCenterArrayX() : double[][]
+getbCenterArrayY() : double[][]
+getFullArray() : boolean[][]

Figure 7 : Board Class Diagram

This class contains board objects of the game.

Attributes:

1. **Id**: This integer attribute is an integer representation of the board type.
2. **bRowNum**: Row number of black board.
3. **bColNum**: Column number of black board.
4. **bRecWidth**: Size of every rectangle which is a part of the black board.

5. **bRecCent**: Coordinate of center of the rectangle.
6. **bRecArray**: A two dimensional array which includes the rectangles of the black board.
7. **bCenterArrayX**: This two dimensional array keep the x coordinates of centers of the rectangles.
8. **bCenterArrayY**: This two dimensional array keep the y coordinates of centers of the rectangles.
9. **bFullArray**: This two dimensional array keeps the checking result about the black board is full or not as boolean.
10. **gRowNum**: Row number of grey board
11. **gColNum**: Column number of grey board
12. **gRecWidth**: Size of every rectangle which is a part of the grey board.
13. **gRecCent**: Coordinate of center of the rectangle.
14. **gRecArray**: A two dimensional array which includes the rectangles of the grey board.
15. **gCenterArrayX**: This two dimensional array keep the x coordinates of centers of the rectangles.
16. **gCenterArrayY**: This two dimensional array keep the y coordinates of centers of the rectangles.
17. **gFullArray**: This two dimensional array keeps the checking result about the grey board is full or not as boolean.
18. **3DRowNum**: Row number of 3D board.
19. **3DColNum**: Column number of 3D board.
20. **3DRecWidth**: Size of every rectangle which is a part of the 3D board.
21. **3DRecCent**: Coordinate of center of the rectangle.
22. **3DRecArray**: A two dimensional array which includes the rectangles of the 3D board.
23. **3DCenterArrayX**: This two dimensional array keep the x coordinates of centers of the rectangles.
24. **3DCenterArrayY**: This two dimensional array keep the y coordinates of centers of the rectangles.

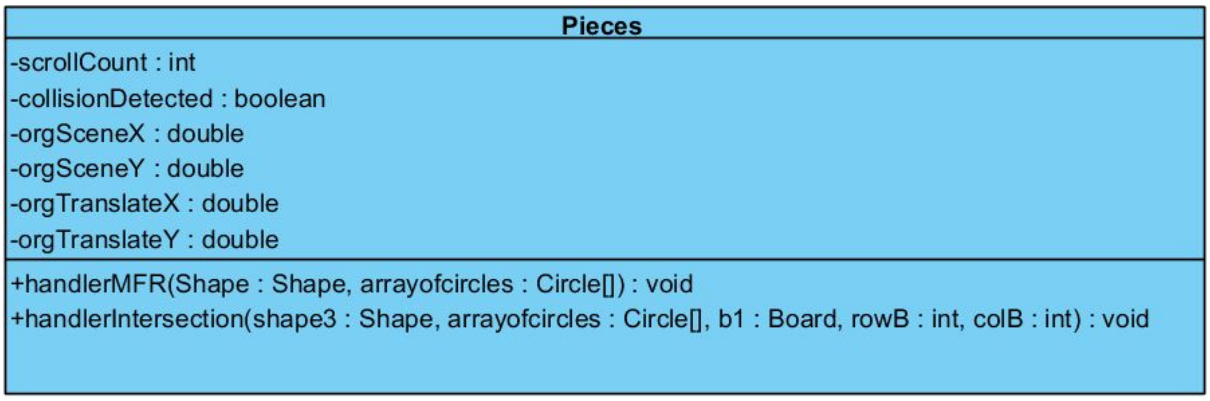
Constructor:

1. **Board(int id):** This method takes id of the board as parameter and decides to use which board according to the id value. After choosing the board type it creates the board for the game scene.

Methods:

1. **getBoardId():** This method is to return the id of the selected board.
2. **getRowNo():** This method returns the row number of the selected board.
3. **getColNo():** This method returns the column number of the selected board.
4. **getRecWidth()**: This returns the width of the rectangles of the selected board.
5. **getCenter():** This method returns the coordinate of the center of the rectangles of the selected board.
6. **getRectangle(int row, int col):** This method takes the row and column numbers as parameters then returns the rectangle with these numbers of rectangle array.
7. **getRecArray():** This method returns the array which keeps the rectangles of the selected board.
8. **getCenterArrayX():** This method was written to return the x coordinate of the rectangle of the selected board.
9. **getCenterArrayY():** This method is to return the y coordinate of the rectangle of the selected board.
10. **getFullArray():** This method return boolean value about the board is full or not.

Pieces Class:



```
Pieces
-scrollCount : int
-collisionDetected : boolean
-orgSceneX : double
-orgSceneY : double
-orgTranslateX : double
-orgTranslateY : double
+handlerMFR(Shape : Shape, arrayofcircles : Circle[]) : void
+handlerIntersection(shape3 : Shape, arrayofcircles : Circle[], b1 : Board, rowB : int, colB : int) : void
```

Figure 8 : Pieces Class Diagram

This class contains piece objects of the game. There are 12 pieces on the game.

Attributes:

1. **scrollCount**: This attribute is an integer to calculate number of scrolling.
2. **collisionDetected:** This attribute is a boolean to check the collision between the pieces.
3. **orgSceneX, orgSceneY, orgTranslateX, orgTranslateY:** These attributes are double and were written to calculate the coordinates of pieces when they are dragging or dropping.

Methods:

1. **handlerMFR(Shape shape, Circle[ ] arrayOfCircles):** This method was written to drag and drop a piece and rotate or flip the selected piece.
2. **handlerIntersection(Shape shape3, Circle[] arrayOfCircles, Board b1, int rowB, int colB):** This method was written to check the collision between a piece and a part of the board. Also this method is to make fit the piece at the middle of the desired rectangle.

Piece Class:

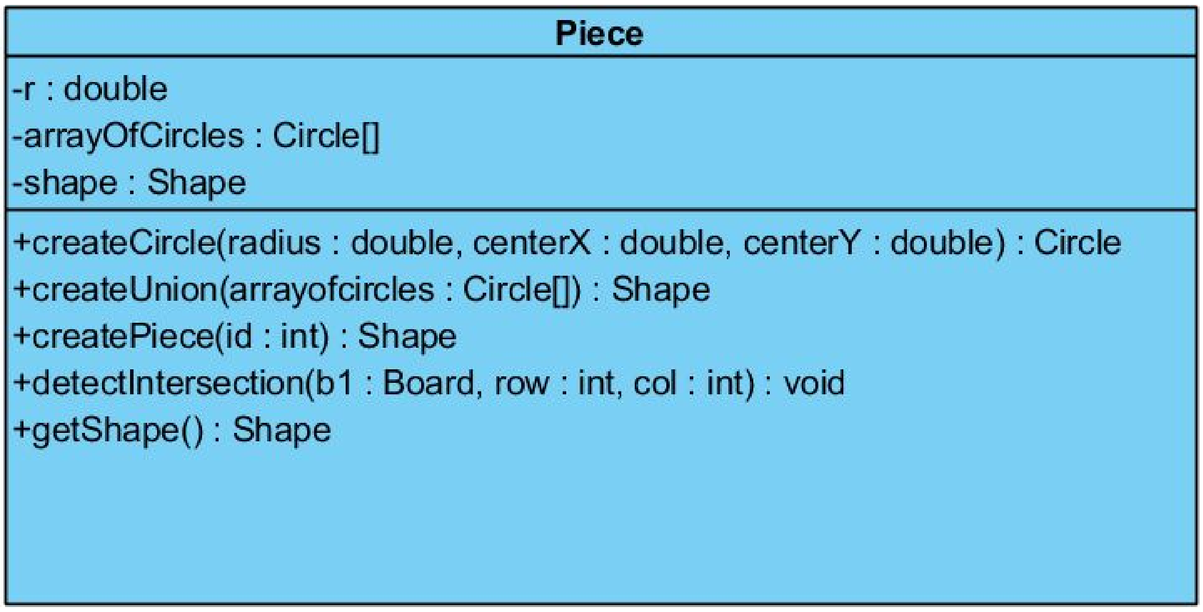

Piece
-r : double
-arrayOfCircles : Circle[]
-shape : Shape
+createCircle(radius : double, centerX : double, centerY : double) : Circle
+createUnion(arrayofcircles : Circle[]) : Shape
+createPiece(id : int) : Shape
+detectIntersection(b1 : Board, row : int, col : int) : void
+getShape() : Shape

Figure 9 : Piece Class Diagram

This class contains piece object of the game.

Attributes:

1. **r:** This attribute is a double and hold radius of the circles of the piece.
2. **arrayOfCircles**: This attribute is an circle array to hold circles of the piece.
3. **shape**: This attribute is a shape and represent final shape of piece.

Methods:

1. **createCircle(double radius, double centerX, double centerY):** This method was written to create circle with given radius and coordinates.
2. **createUnion(Circle[ ] arrayofcircles):** This method was written to create union of circles to create shape object.
3. **createPiece(int id):** This method create piece object with given id. Id can range between 1 to 12.
4. **detectIntersection(Board b1, int row, int col):** This method is detect any intersection between board and piece.
5. **getShape():** This method returns the shape attribute.

Levels Class

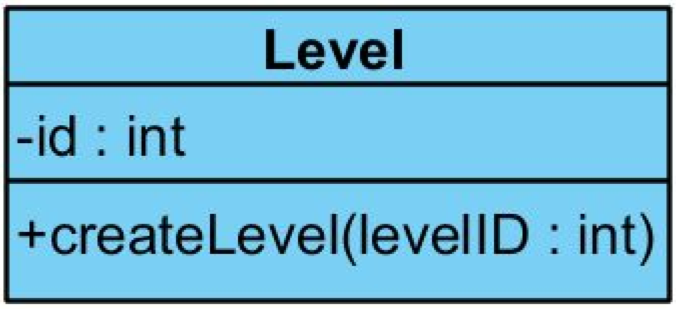


Figure 10 : Level Class Diagram

This class is to create the selected level.

Attributes:

1. **Id**: This attribute is an integer representation of the piece type.

Methods:

1. **createLevel** (int levelId): Create level according to its Id.

Leaderboard Class

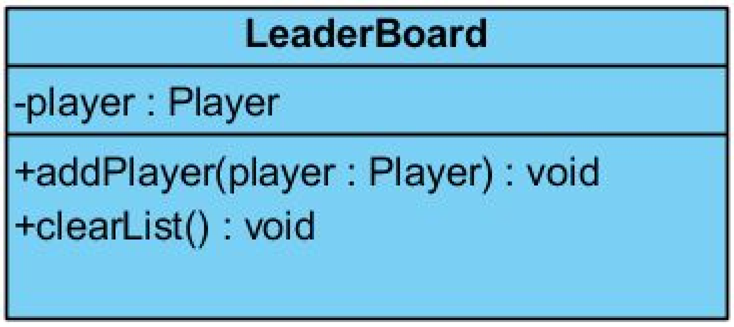


Figure 11 : LeaderBoard Class Diagram

This class is to keep and demonstrate the scores of the players. At the end of the each level the player will be asked if he wants to save his score. If he want it, this class takes his name and score and adds to the score list's appropriate place considering other scores.

Attributes:

1. **player**: This attribute is an array which keeps the scores and the names as objects of the players in orders.

Methods:

1. **addPlayer(Player player):** It adds player object to the list if user desire to record his score. It decides the place of the new player by comparing his score with the others.
2. **clearList():** It removes player objects from the list to clear leaderboard.

Sound Manager Class

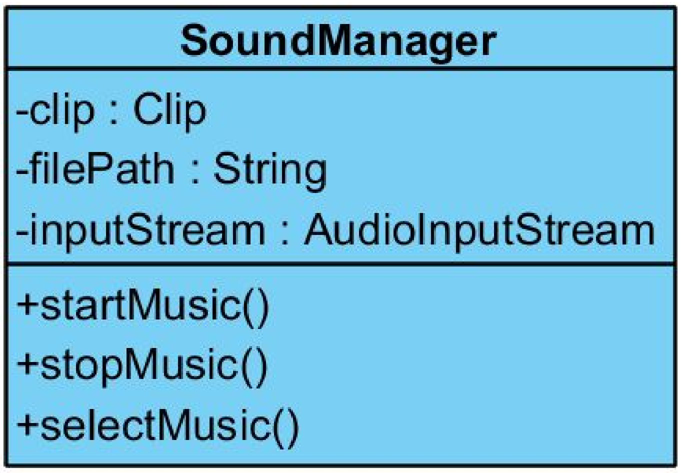


Figure 12 : SoundManager Class Diagram

This class is written for the sound settings of the game. The player can choose the soundtrack which plays at the background of the game by this class. Also the player can start or stop the sound.

Attributes:

1. **clip:** This attributes’ type is clip and it represents the sound inthe background of the game.
2. **filePath:** This attribute is for determining which file path to find sound file.
3. **inputStream:** This attribute is related to sound of the game.

Methods:

1. **startMusic():** This method is for starting the sound if the user want to. It doesn’t return anything.
2. **stopMusic():** This method is for stop the music in the background if the user want to. It doesn’t return anything.

3. **selectMusic():** The soundtrack can be changed by the user with this method. It doesn't return anything.

Time Manager Class

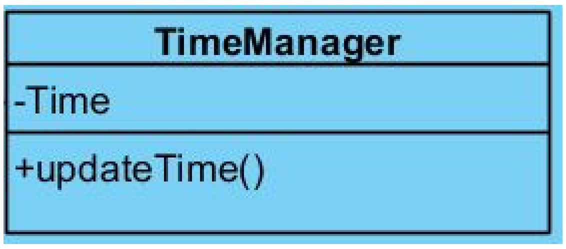


Figure 13 : TimeManager Class Diagram

This class is for control the time in the game.

Attribute:

1. **Time:** This attribute holds the time.

Methods:

1. **updateTime():** This method update time.

Menu Controller Class

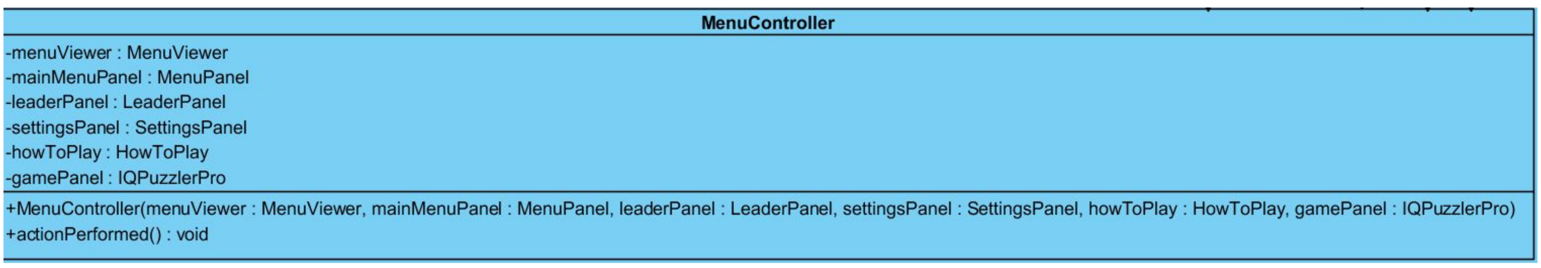

MenuController
-menuViewer : MenuViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-gamePanel : IQPuzzlerPro
+MenuController(menuViewer : MenuViewer, mainMenuPanel : MenuPanel, leaderPanel : LeaderPanel, settingsPanel : SettingsPanel, howToPlay : HowToPlay, gamePanel : IQPuzzlerPro)
+actionPerformed() : void

Figure 14 : MenuController Class Diagram

This class is controller class of menu panels.

Attributes:

1. **MenuViewer:** This attribute is the instance of menu viewer class.
2. **mainMenuPanel:** This attribute is the instance of main menu panel.
3. **leaderPanel:** This attribute is the instance of leader panel.
4. **settingsPanel:** This attribute is the instance of settings panel.
5. **howToPlay:** This attribute is the instance of how to play panel.
6. **gamePanel:** This attribute is the instance of game panel.

Methods:

1. **actionPerformed():** This method is button clicked action for all buttons in the menu panels. This method is determine what to do according to which button is clicked.

Multiplayer Game Class

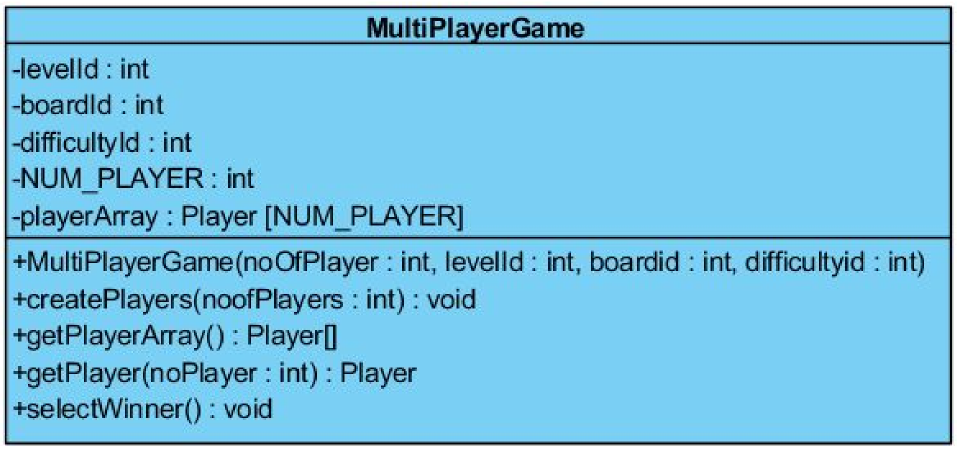

MultiPlayerGame
-levelId : int
-boardId : int
-difficultyId : int
-NUM_PLAYER : int
-playerArray : Player [NUM_PLAYER]
+MultiPlayerGame(noOfPlayer : int, levelId : int, boardid : int, difficultyid : int)
+createPlayers(noofPlayers : int) : void
+getPlayerArray() : Player[]
+getPlayer(noPlayer : int) : Player
+selectWinner() : void

Figure 15 : MultiPlayer Class Diagram

In our version of IQPuzzlerPro , we have multiplayer selection. If the user wants to, he or she can play the game with his friends. To provide this property 'MultiPlayerGame' class is written. With this class, two people can play the same level together in order. At the end of the game , scores of the players will be compared and determined the winner.

Constructor:

1. **MultiPlayerGame(int noOfPlayer,int levelId,int boardid,int difficultyid) :** This constructor creates to multi player game mode .

Method:

1. **selectWinner**(): This method is to determine who the winner is. It takes the array player and compares the scores of the players and chooses the best score. Then determine the player with the chosen score as the winner. It returns the player who win.
2. **getPlayerArray():** This method is to return player array.This returns Player.
3. **getPlayer(int noPlayer):**This method is to return player according to player no.

4. **createPlayers(int noOfPlayer):**This method is to create players in player array according to player no.

Attributes:

1. **playerArray:** This attribute is an array of Players which used Player objects.It includes name and score of each player.
2. **NUM_PLAYER :**This attribute is int for number of player.
3. **levelId:** This attribute is an integer level counter of player. Player could continue from the last checkpoint.
4. **boardId:** This attribute is an integer board holder, before the game begins player chose his board and it is taken as integer for black and gray ones.
5. **difficultyId:** This attribute is an integer difficulty data, before game begins player chose his difficulty preference and its is taken as integer for system. There are several types of difficulty and different game options can be served.

SinglePlayer Game Class

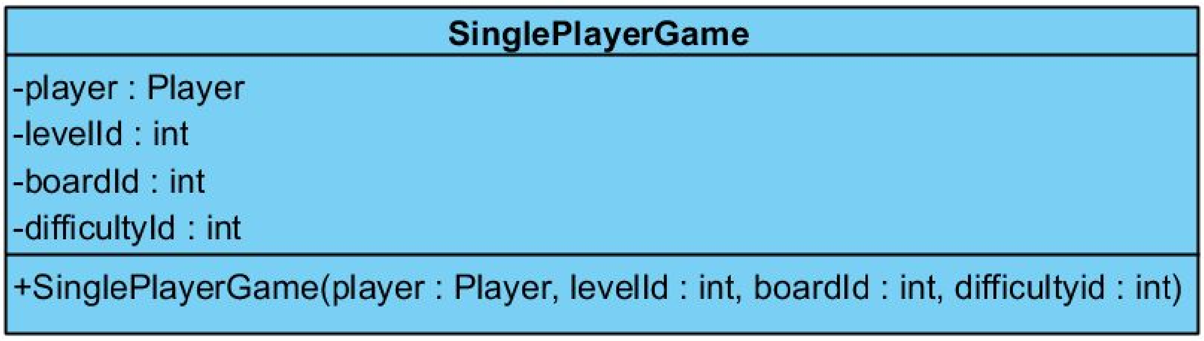

SinglePlayerGame
-player : Player
-levelId : int
-boardId : int
-difficultyId : int
+SinglePlayerGame(player : Player, levelId : int, boardId : int, difficultyid : int)

Figure 16 : SinglePlayer Class Diagram

This class is to keep the player's keep the properties by using Player class.

Attributes:

1. **player**: This attribute is an object of Players. It includes the name, time, move count, and the score of the player.
2. **levelId:** This attribute is an integer level counter of player. Player could continue from the last checkpoint.
3. **boardId:** This attribute is an integer board holder, before the game begins player chose his board and it is taken as integer for black and gray ones.

4. **difficultyId:** This attribute is an integer difficulty data, before game begins player chose his difficulty preference and its is taken as integer for system. There are several types of difficulty and different game options can be served.

Methods:

1. **saveScore**(): This method is that If the player wants to, it takes the score of the player from the Players class and return it to send it to the leaderboard and add the player's score with his name to the list.

Player Class:

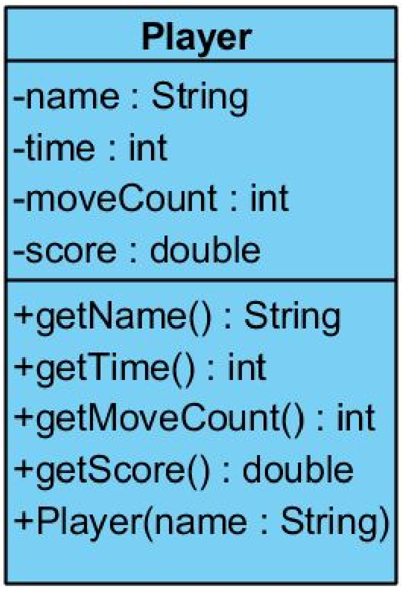


Figure 17 : Player Class Diagram

This class is used to keep the name, score, time and move count of players.

Attributes:

1. **name**: This attribute is a string for the name of the player.
2. **time:** This int attribute is elapsed time for finishing the game.
3. **moveCount:** This int attribute represents the number of move to finish the game.
4. **score**: This attribute is an integer for score.

Methods:

1. **getName**(): This method asks the name of the player and it returns the string name.
2. **getTime():** This method returns the time as int.

3. **getMoveCount():** This method returns the number of moves as int.
4. **getScore**(): This method is used to get score from the game manager class and returns it as integer.
5. **Player(String name):** It creates a player with the name taking as parameter.

LeaderPanel:

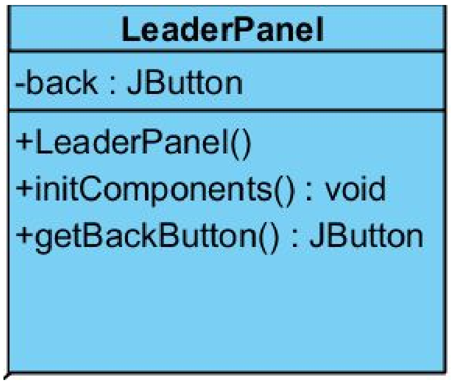


Figure 18 : Leader Panel Class Diagram

This panel is created to demonstrate Leader Board.

Constructor:

1. **LeaderPanel :** This is constructor for this class.

Attributes :

1. **back :** This attribute is a button to return back

Methods :

1. **getBackButton():** This function returns back button.

SettingsPanel:

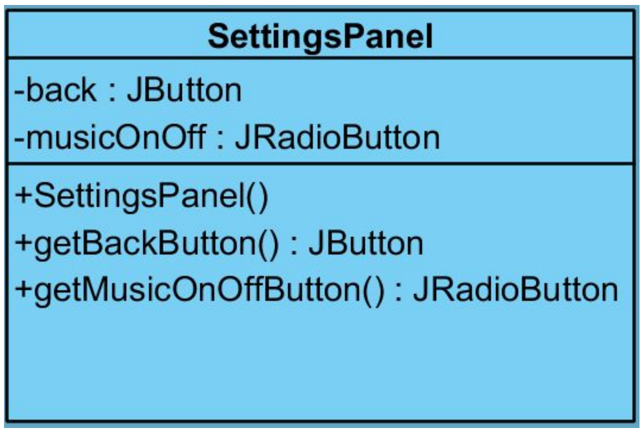

SettingsPanel
-back : JButton
-musicOnOff : JRadioButton
+SettingsPanel()
+getBackButton() : JButton
+getMusicOnOffButton() : JRadioButton

Figure 19 : Settings Panel Class Diagram

This panel is for user choose setting of the game.

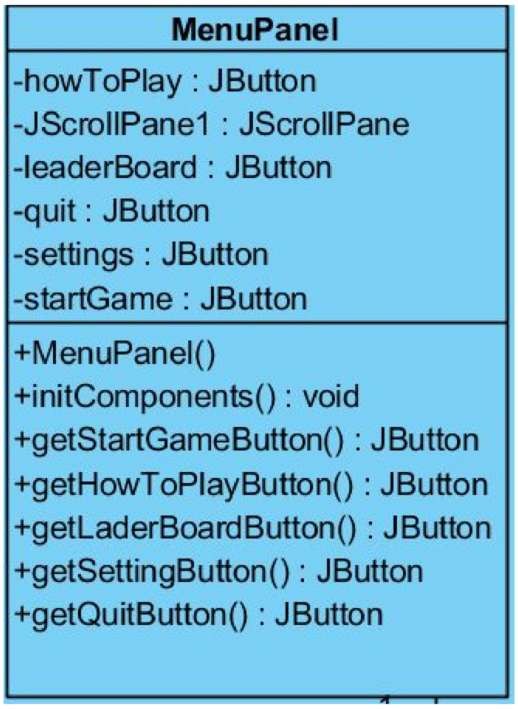
Attributes:

1. **back:** This attribute is a button for return back to the game.
2. **musicOnOff**: This attribute is a radio button to choose whether music is wanted to be on or off.

Method:

1. **getBackButton():** This function returns back button.
2. **getMusicOnOffButton():** This function returns musicOnOff button.

MenuPanel:



```
MenuPanel
-howToPlay : JButton
-JScrollPane1 : JScrollPane
-leaderBoard : JButton
-quit : JButton
-settings : JButton
-startGame : JButton
+MenuPanel()
+initComponents() : void
+getStartGameButton() : JButton
+getHowToPlayButton() : JButton
+getLaderBoardButton() : JButton
+getSettingButton() : JButton
+getQuitButton() : JButton
```

Figure 20 : Menu Panel Class Diagram

This panel is for generating the first panel of the game, Menu.

Constructor : Creates the necessary components for panel.

Attributes :

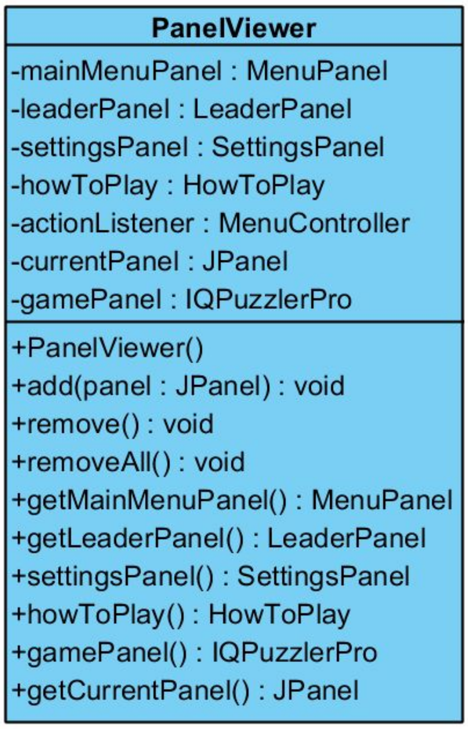
1. **howToPlay** :This attribute is a JButton for entering the “how to play” panel
2. **jScrollPane** : This attribute is a panel.
3. **leaderBoard** :This attribute is for going to “LeaderBoard” panel.
4. **quit** : This attribute is to quit the IQ Puzzler Pro game.
5. **setting** :This attribute is to go to “Setting” panel.

6. **startGame** : This attribute is to go for the next step of starting game, “ModeSelection” panel.

Methods :

1. **getStartGameButton()** :This method returns the startGame button.
2. **getHowToPlayButton()** : This method returns the howToPlay button.
3. **getLeaderBoardButton()** : This method returns the leaderBoard button.
4. **getSettingButton()** : This method returns the settings button.
5. **getQuitButton()** : This method returns the quit button.

PanelViewer:



```
PanelViewer
-mainMenuPanel : MenuPanel
-leaderPanel : LeaderPanel
-settingsPanel : SettingsPanel
-howToPlay : HowToPlay
-actionListener : MenuController
-currentPanel : JPanel
-gamePanel : IQPuzzlerPro
+PanelViewer()
+add(panel : JPanel) : void
+remove() : void
+removeAll() : void
+getMainMenuPanel() : MenuPanel
+getLeaderPanel() : LeaderPanel
+settingsPanel() : SettingsPanel
+howToPlay() : HowToPlay
+gamePanel() : IQPuzzlerPro
+getCurrentPanel() : JPanel
```

Figure 21 : Panel Viewer Class Diagram

This class is contains all menu related panels and help menu controller class to show menu related panels.

Attributes:

1. **mainMenuPanel:** This attribute is the instance of main menu panel class.
2. **leaderPanel:**This attribute is the instance of leader panel class.
3. **settingsPanel:** This attribute is the instance of settings panel class.

4. **howToPlay:** This attribute is the instance of how to play panel class.
5. **actionListener:** This attribute is the instance of menu contoller class.
6. **currentPanel:** This attribute is the represent current panel in the frame.
7. **gamePanel:** This attribute is for the general game panel.

Constructor : Creates the necessary components for panel.

Methods:

1. **add(JPanel panel):** This method add a panel to frame.
2. **remove():** This method remove a panel from frame.
3. **removeAll():** This method remove all panels from frame.
4. **getMainMenuPanel():** This method returns main menu panel.
5. **getLeaderPanel():** This method returns leader panel.
6. **settingsPanel():** This method returns settings panel.
7. **howToPlay():** This method returns howToPlay panel.
8. **gamePanel():** This method returns game panel.
9. **getCurrentPanel():** This method returns current panel.

BoardSelectionPanel:

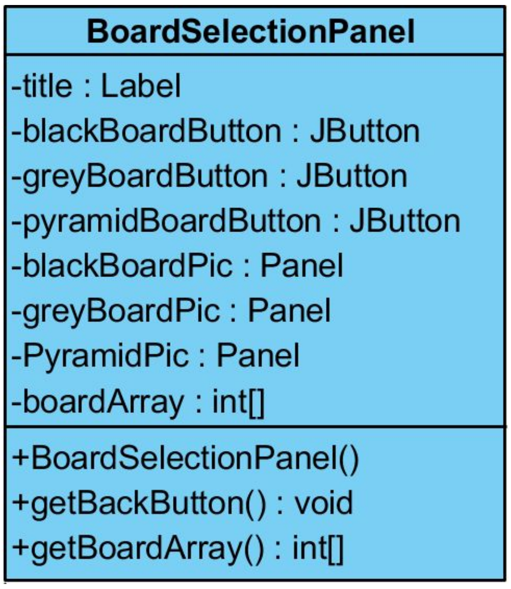


Figure 22 : Board Selection Panel Class Diagram

This panel is for selecting board type.

Attributes:

1. **title:** This attribute is the title of the label.

2. **blackBoardButton:** This attribute offers a selection as a button for black board.
3. **greyBoardButton:** This attribute offers a selection as a button for grey board.
4. **pyramidBoardButton:** This attribute offers a selection as a button for 3D game board.
5. **blackBoardPic:** This attribute is the image of black board.
6. **greyBoardPic:** This attribute is the image of grey board.
7. **PyramidPic:** This attribute is the image of 3D game board.
8. **boardArray:** This attribute is the array that holds board ids.

Constructor : Creates the necessary components for panel.

Methods:

1. **getBackButton():** This function returns the button object.
2. **getBoardArray():** This function returns the board array.

BoardViewPanel:

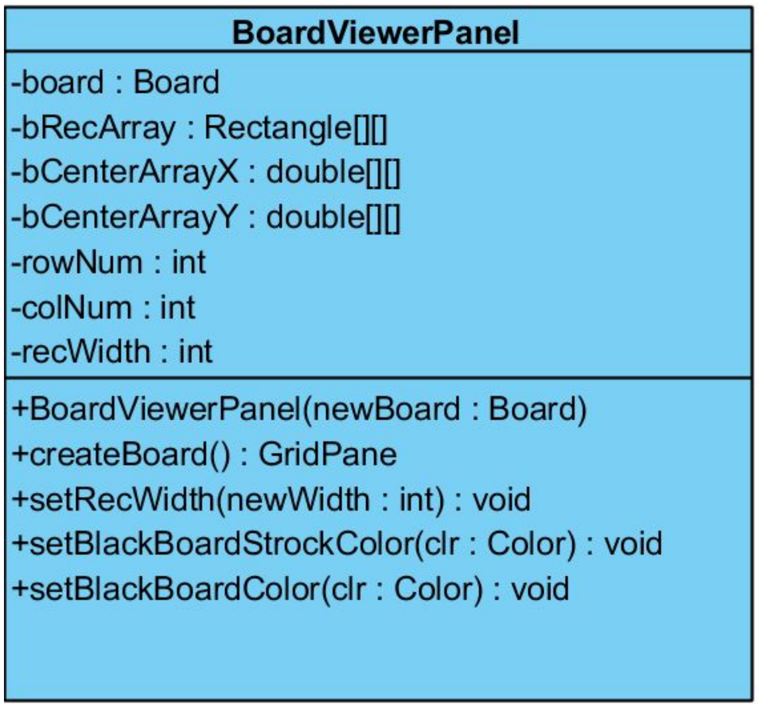

BoardViewerPanel
-board : Board
-bRecArray : Rectangle[][]
-bCenterArrayX : double[][]
-bCenterArrayY : double[][]
-rowNum : int
-colNum : int
-recWidth : int
+BoardViewerPanel(newBoard : Board)
+createBoard() : GridPane
+setRecWidth(newWidth : int) : void
+setBlackBoardStrockColor(clr : Color) : void
+setBlackBoardColor(clr : Color) : void

Figure 23 : Board Viewer Panel Class Diagram

This panel is view of the board object.

Attributes:

1. **board**: This attribute is instance of board object.

2. **bRecArray**: This attribute is 2D Rectangle array of rectangles. This array represents board as rectangles.
3. **bCenterArrayX**: This attribute is 2D double array. This array holds x coordinates of centers of rectangles in the board.
4. **bCenterArrayY**: This attribute is 2D double array. This array holds x coordinates of centers of rectangles in the board.
5. **rowNum**: This attribute holds row num of board.
6. **colNum**: This attribute holds col num of board.
7. **recWidth**: This attribute holds width of rectangle object.

Constructor : Creates the necessary components for panel.

Methods:

1. **BoardViewerPanel(Board newBoard):** This is the constructor of panel. It takes board object to its constructor.
2. **createBoard():** This method create view of board object according to its id.
3. **setRecWidth(int newWidth):** This method determines the width of the rectangle objects in the board.
4. **setBlackBoardStrockColor(Color clr):** This method determines the color of rectangle objects strock property.
5. **setBlackBoardColor(Color clr):** This method determines the color of the board object.

How To Play Panel

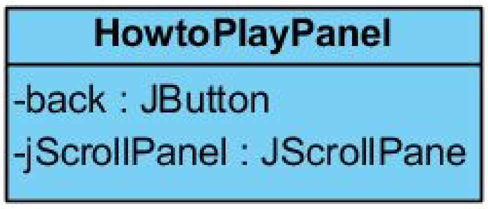


Figure 24: How To Play Panel Class Diagram

This class is created for How To Play panel which describes user to how to play.

Attributes :

1. **back :** this button is created for back which is JButton
2. **jScrollPanel :** this is panel for game view.

Mode Selection Panel

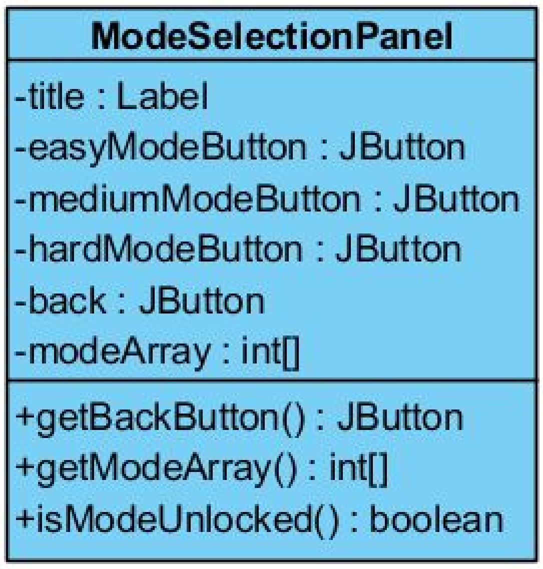

ModeSelectionPanel
-title : Label
-easyModeButton : JButton
-mediumModeButton : JButton
-hardModeButton : JButton
-back : JButton
-modeArray : int[]
+getBackButton() : JButton
+getModeArray() : int[]
+isModeUnlocked() : boolean

Figure 25 : Mode Selection Panel Class Diagram

Attributes:

1. **easyModeButton**:This button helps to user to choose easy level game.
2. **hardModeButton**: This button helps to user to choose hard level game.
3. **mediumModeButton**:This button helps to user to choose medium level game.
4. **title:** This label attribute is for the title.
5. **back:** This button attribute is to get back.
6. **modeArray:** This integer array keeps the mode types.

Constructor : Creates the necessary components for panel.

Methods :

1. **getBackButton():** This method returns the back button.
2. **getModeArray():** This returns the selected mode type from the array.
3. **isModeUnlocked():** This method checks and returns the mode is unlocked or locked.

Level Selection Panel:

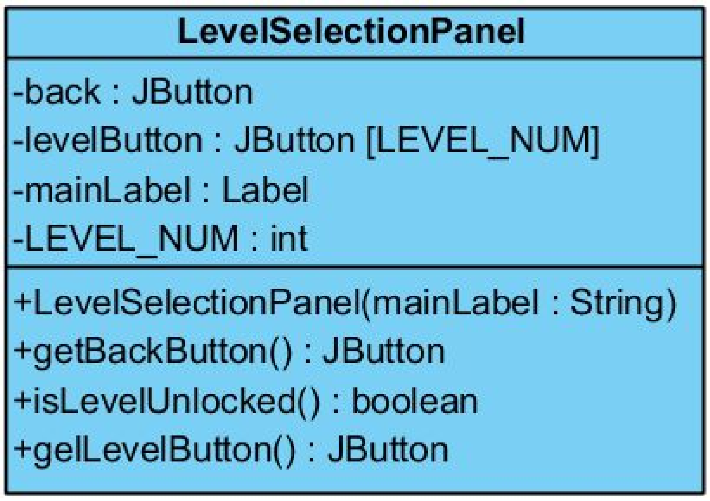

LevelSelectionPanel
-back : JButton
-levelButton : JButton [LEVEL_NUM]
-mainLabel : Label
-LEVEL_NUM : int
+LevelSelectionPanel(mainLabel : String)
+getBackButton() : JButton
+isLevelUnlocked() : boolean
+gelLevelButton() : JButton

Figure 26: Level Selection Panel Class Diagram

Constructor:

1. **LevelSelectionPanel(String mainLabel):** This constructor display new level selection screen.

Methods:

1. **getBackButton():** This method returns the back button.
2. **getLevelButton():** This method returns level button.
3. **isLevelUnlocked():** This method returns boolean if new level is unlocked or not. Whenever levels are passed new levels are unlocked.

Attributes:

1. **LEVEL_NUM:** This label holds level num integers.
2. **back:** This button attribute is to get back.
3. **levelButton:** This button attribute represents the buttons of levels.
4. **mainLabel:** This attribute represents the label in the screen.

4.3 Packages

4.3.1 JAVA.util

Util package can be thought as a data structures container, so we need ArrayList to hold multiple variables inside, such as board types and members inside. Thus, we could use these lists to integrate our game into language environment.

4.3.2 java.fx.scene.layout

This package could give us User Interface layouts to integrate our Graphical User Interface objects. JAVAFX is thought as beneficial support for game’s 3D object mode among group members.

4.3.3 java.fx.scene.paint

This package could beneficial for visualization our game in JAVA programming language.

4.3.4 java.fx.application

It provides the application life cycle for our game project.

4.3.5 javafx.embed.swing

It provides set of classes to use JAVAFX inside the Swing applications for Graphical User Interface support for visualization.

4.3.6 java.fx.geometry

It provides the sets for 2D objects of 2D game mode. These are usable for defining and performing operations among objects related to two-dimensional geometry spaces.

4.3.7 javafx.util

It provides various utilities and helper classes by containing inside.

4.4 Design Pattern

To make our game implementation more readable and writable, we use façade design pattern and MVC design pattern in our game.

4.4.1 Façade Design Pattern

In our game, we will use data from JSON File in several classes such as player class, level class, etc. All these classes must interact with data. So we decided to use façade design pattern, and create dataManagement class that manage communication with these classes and database. By this way, when we change something in the JSON File, we don’t need to change all these classes, we just need to change dataManagement class. In this way, we aim to reduce coupling between classes.

4.4.2 MVC

As we mention above, we will use MVC design pattern since it fits our game. In our implementation we constantly change features about interface of our game, so separating interface and model is a good idea in our game. Also, it increases readability and writability of our code. In addition, in the future, we are planning to add shop future in our game where users can buy color of the board or background image. So MVC also helps this kind of situation since we don’t need to change all classes.

5. Improvement summary

In the first iteration, we don't know the general structure of our game. We only know some of the classes. So in this iteration, we add new classes in our diagram, create subsystem decomposition among these classes, use and show some design patterns. Also we reorganize our tradeoffs according to new implementations.

In the first iteration, we decide to use MS database system to hold and manage our data. However, after first iteration, we decide to use JSON File system to hold and manage our data since it is more convenient to use JSON in our project. According to changes in game management, we add new classes and attributes. Also, we add classes about 3D implementations in the second iteration.

6 Glossary & References

[1] (2013, December 09). Retrieved November 11, 2018, from https://docs.oracle.com/javafx/2/api/overview-summary.html

[2] Retrieved December 12, 2018 from
https://en.wikipedia.org/wiki/JavaFX