Department of Computer Engineering

Bilkent University

CS 319 - Object-Oriented Software Engineering Project

IQ Puzzler

Design Report

Group Name: Violet

Group Members

Ege Akın
Safa Aşkın
Betim Doğan
Alp Ertürk
Elif Gülşah Kaşdoğan

TABLE OF CONTENTS

1. Introduction

1.1 Purpose of the system

IQ Puzzler Pro is a 2-D and 3-D logical puzzle game which is adapted from its board game version. Its design is implemented in such ways to meet the maximum user expectation. IQ Puzzler Pro aims to be portable, user-friendly and a brain-teasing challenging game which supports the development of cognitive skills. The goal of the player is to place geometric pieces of different colors and sizes to board or to create a specific shape with the given pieces.

1.2 Design Goals

Design goals are important steps for users to get the possible maximum satisfaction from the game. There are some non-functional requirements of the system as we mentioned before in our analysis report. These non-functional requirements are detailed below.

1.2.1 Criteria

Maintainability

Maintainability is an important point for attribution of quality of a software. The design of the game is based on object-oriented programming concepts will enable the game to go through the changes easily. The multilayered design of the game provides flexibility that allows the system to be easily modified and meet the future demands.

Usability

Our aim is to create a game that is easy to use for the players. In general, usability is a substantial aspect for our game because it

depends on effectiveness, efficiency of the game and affects satisfaction and comfort of the user. IQ Puzzler Pro has user-friendly interface design to fulfill the user expectation. It will enable users to easily understand the directions of the game and control with easily understandable buttons and control system. In addition to these, the game includes tutorial section, accessible from the main menu that explains and shows the game play in a detailed way.

Performance

Performance is another aspect that we mentioned before due to the fact that, the players will not want to wait for a long time to get response while playing IQ Puzzler Pro. We will design our game with the aim for almost immediate response time. For avoiding decrease in computer's performance, we will not use high-resolution graphics in our game.

Reliability

Our aim is to provide users to a bug-free system, while we designing the game. The system will be prepared for unexpected conditions such as unexpected inputs and while they happens, the game will not crash. For achieving this aim, we will use a testing procedure during the whole development stages. The testing procedure and the boundary conditions will be prepared elaborately, for not missing any kind of unexpected situation which may cause system crashes.

Adaptability

Since we use Java as our programming language for development of our game we will not have to worry about adaptability. Due to the fact that Java has cross-platform

portability, users will be able to play IQ Puzzler Pro in all JRE installed platforms.

1.2.2 Trade-Offs

Memory vs Performance:

We will use object oriented concept and principles while we will designing the IQ Puzzler Pro Game, so that it enables us to have efficient run time in the design. Therefore, we will not concern about memory usage.

Functionality vs Usability:

IQ Puzzler Pro is a game that aims to place the different puzzle pieces into correct places. We do not prefer to use complex interactions and game control system due to provide game to be easily understandable and we add tutorial option to improve usability. Due to the aim of our game is entertaining the users; we will spend more effort to make the system useful rather than developing the functionality more than necessary. The game will have simple instructions and a menu rather than complex and confusing ones.

Efficiency vs Reusability

Since we do not supplement our classes in any other game or system, we will not worry in the aspect of reusability. For this reason, the design of our classes and our game structure do not have to be complex more than needed. Our code and tasks will only be enough for fulfilling the game requirements. Efficiency is a more substantial aspect in terms of limited time.

Functionality vs Robustness

Since the main aim of IQ Puzzler Pro is to place various puzzle pieces in correct order, there can be various invalid actions while placement. Therefore, we will have some kind of precautions for handling with these exceptions and increasing the functionality. The exceptions will be determined carefully and the code will be developed with considering them.

Cost vs Portability

Our game will only be implemented for desktop operating systems that run JRE, since it is implemented by Java. Consequently, the game will not be played in mobile platforms and the platforms that do not satisfy the requirements.

1.3 Definitions

- **Java[1]:** is a fast, secure and reliable programming language and computing platform that allows developers to write code for different systems.

- **JRE (Java Runtime Environment)[2]:** is the combination of JVM (Java Virtual Machine), platform core classes and supported libraries and it is a part of the JDK (Java Development Kit).

- **Cross-Platform[3]:** is a type of computer software that can be implemented on multiple computing platforms. A cross-platform application may run on every platform that satisfies the system requirements of the application.

1.4 Overview

IQ Puzzler Pro is a logical computer game which is adapted from its board version. Our design aim is to create a flexible design to prevent bugs and handling the possible exceptions. Since we have limited time, we tried to make our implementation as easy as possible. For avoiding the complexity and handling the exceptions, we aim to create a design pattern with an elaborate detection.

1.5 References

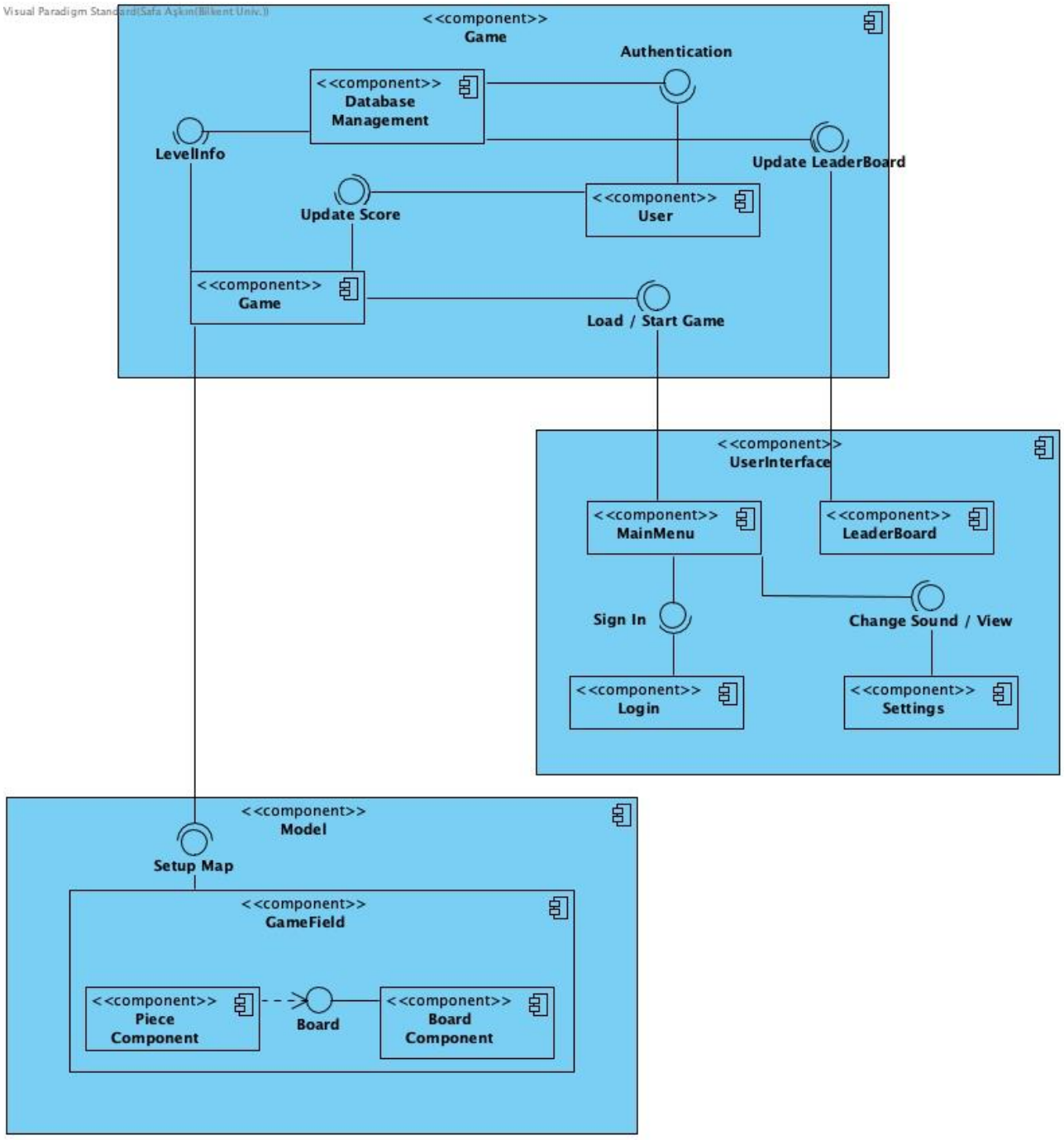
[1] https://java.com/en/download/faq/whatis_java.xml

[2] https://www.techopedia.com/definition/5442/java-runtime-environment-jre

[3]http://www.wikizeroo.net/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQ3Jvc3MtcGxhdGZvcm1fc29mdHdhcmU

2. Software Architecture

2.1 Subsystem Decomposition



We will introduce subsystems in this section. Decomposing the system into subsystems will reduce the coupling which in turn our system will be more concise and organized. Decomposing will allow us to construct a system which is more flexible to extensions and coherency will increase.

We used MVC pattern for decomposition, Game for Model, User Interface component represents view by providing the main user interface and Model component is a controller for system which will manipulate the Model.

We tried to divide systems according to their functionality to provide clarity about job definition of each component. Even if they are separate each subsystem should communicate with others to keep system maintainable.

Controller component Game should communicate with the Model component to set up map for current game, Game Field arranges all items related to game session such as board and pieces. Model component reaches other information related to game by communicating with Game component.

User Interface component is constructed to provide communication between Game component and user. It also updates the database according to current score provided by Game component's elements. Other than those functionalities, User Interface component arranges sound and view changes made by user.

2.2 Hardware / Software Mapping

Our game IQ Puzzler Pro will not need any specific built hardware in order to run properly. IQ Puzzler Pro will require keyboard and mouse for game play. Our game will work properly in different operating systems where required JavaFX libraries and Java Development kits are available. For the storage of the game as levels, board and pieces we will store them as text files also our game will store user profiles and their scores in a database.

2.3 Persistent Data Management

IQ Puzzler Pro will store user profiles and their high scores in a database. Our game will use the users hard drive to store game objects like board, level count or pieces. According to this we are going to store

game data in txt files where some of these files will not be modified by user. However, our game will have user preferences settings, so these data will be able to modified during game by user.

2.4 Access Control and Security

There will be a user authentication, so users will enter the game with username and password.

2.5 Boundary Conditions

2.5.1 Initialization

IQ Puzzler Pro does not require an install because it does not have .exe extension file. Our game will start with an executable .jar file. The game can be transferred by copying .jar file to other computers.

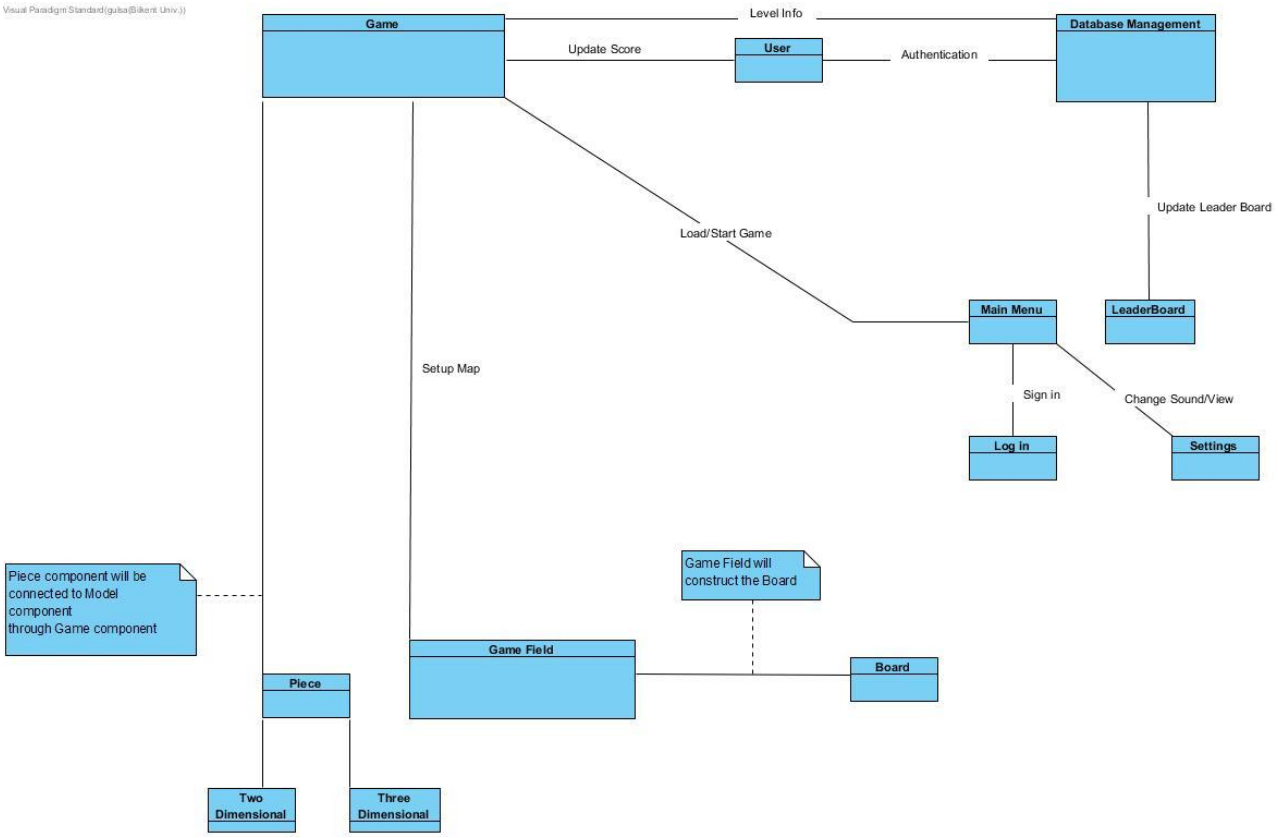
2.5.2 Termination

Player can terminate the game by clicking “Exit” button or by closing .jar file. If player clicks the exit button, there will be a pop up window which asks to user to save the current game.

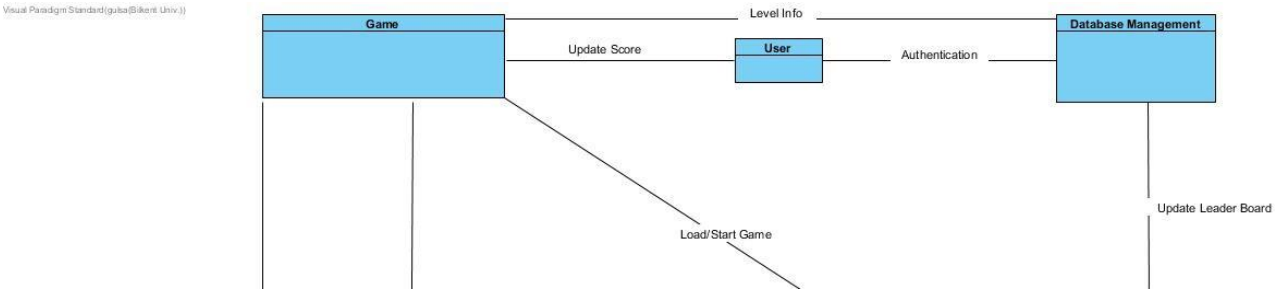
2.5.3 Error

In a case of an error while playing the game there will be a pop up screen which displays a message about loading to user. In such a case the game should be restarted or data files should be changed.

3. Subsystem Services



3.1 Game Subsystem



Game System consists of five classes and responsible for connection between Game Field components, User Interface components and Database System which will record the user information and leaderboard data.

3.1.1 Game Class:

Game Class arranges operations like starting or finishing a game, changing settings by accessing User Interface subsystem components, sets high score and loads a game previously played by accessing Database.

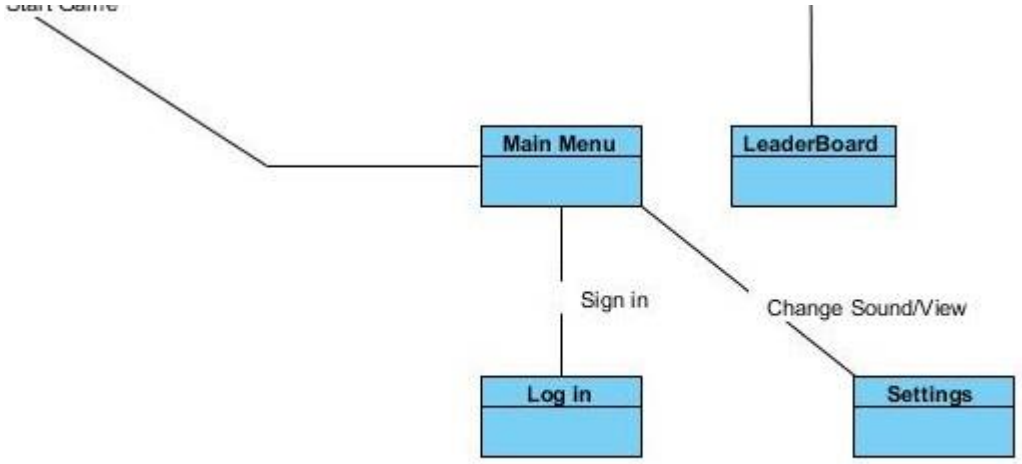
3.1.2 User Class:

User class represents a user with a username and a valid password and arrays representing passed levels. For each game arrays and score will be updated as needed by provided methods. User's data will be stored in database system.

3.1.3 Level Class:

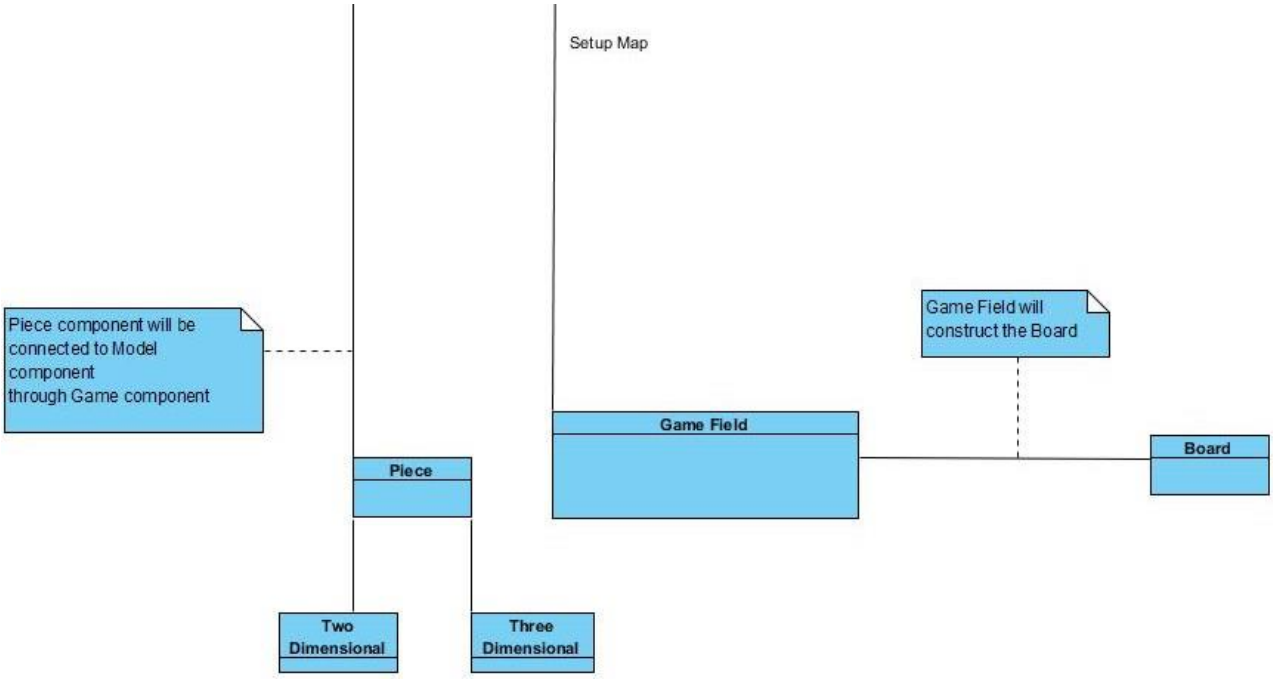
Level class is created to represent typical behavior of a level in our game. Each level has a highest score which will be kept as an attribute of each level object to easily compare if the highest score is reached or not by the current player without retrieving data from database for each game. Level can be either locked or unlocked; each level has an assigned time and array of used and unused pieces to determine needed pieces which will be set by Game Manager without extra operations. Level's highest score will be updated after each game, game will be unlocked if the prerequisites are satisfied and determining used and unused pieces will be done within level object.

3.2 User Interface Subsystem



User Interface Subsystem consists of one class.

3.3 Model Subsystem



3.3.1 Piece Class:

Piece class represents individual pieces which will be inserted to board. Piece has id, position, rotation and size attributes, images can be attached to a piece. Determining if a piece is used or not will be done by a boolean attribute isUsed inside a piece object for simplicity. Methods allow us to change rotation of a piece and image attached.

3.3.2 GameField Class:

Game Field object has size attributes, a board object, a piece object, elapsed time for game set, current score of game and number of moves made so far. Game Field is the class which sets the game essential. Class organizes and updated the pieces board, time, rotations regularly. Basically each object manages its own data and Game Manager manages objects and all game will be set properly. Score will be calculated in this class and will be recorded to database through Game subsystem.

3.3.3 Board Class:

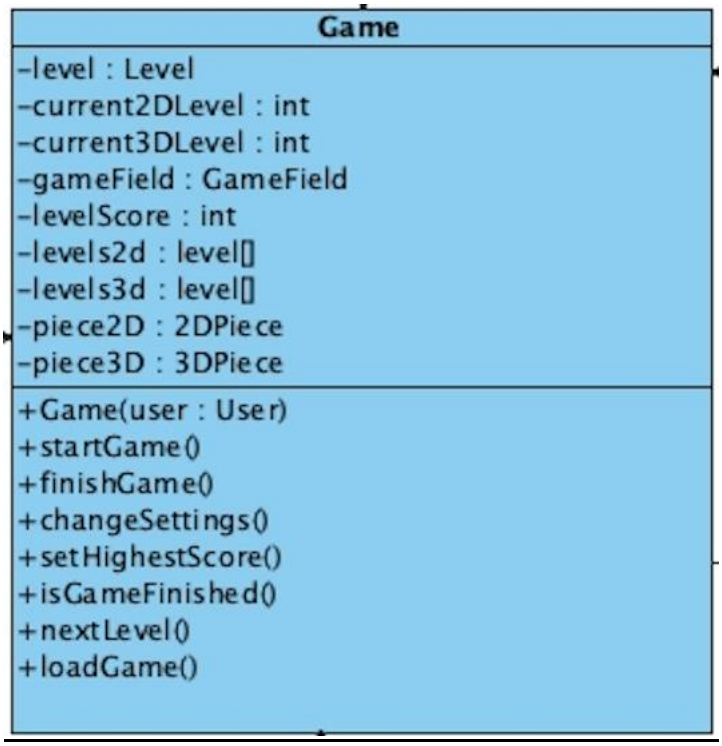
Board is object representing board which player will insert pieces; it is actually the main play field. It has size attributes and functions to check existence of available positions and whether the board is full or not.

3.3.4 InputManager Class:

Input manager takes user input and this information will be used in GameField for game dynamics, it can be seen a class related to default Java listeners.

Class Interfaces

Game Class



Attributes:

- **private Level level:** This attribute is an initialization of Level object, which initialize a level's attributes.

- **private int highestScore:** This attribute stores the user's highest score which is initially 0.

- **private int current2DLevel:** This attribute stores index of current 2D level.

- **private int current3DLevel:** This attribute stores the index of current 3D level.

- **private GameField gameField:** This attribute is an instantiation of GameField class which stores physical objects of the game.

- **private int levelScore:** This attribute stores the level's score which is initially 0 at the beginning of the level.

- **private Level [] levels2D:** This attribute stores all previously generated maps of 2D levels which are generated by Game, in an array.

- **private Level [] levels3D:** This attribute stores all previously generated maps of 3D levels which are generated by Game, in an array.

- **private Piece2D 2DPiece:** This attribute initializes one of twelve distinct pieces.

- **private piece3D 3DPiece:** This attribute initialize the Piece3D class.

Constructors:

- **public Game():** Default constructor for Game class, which simply initialize the game.

Methods:

- **public void startGame:** This method simply starts the game.

- **public void finishGame():** This method simply finishes the game.

- **public void changeSettings():** This method provides user to change settings of the game which is open sound or not.

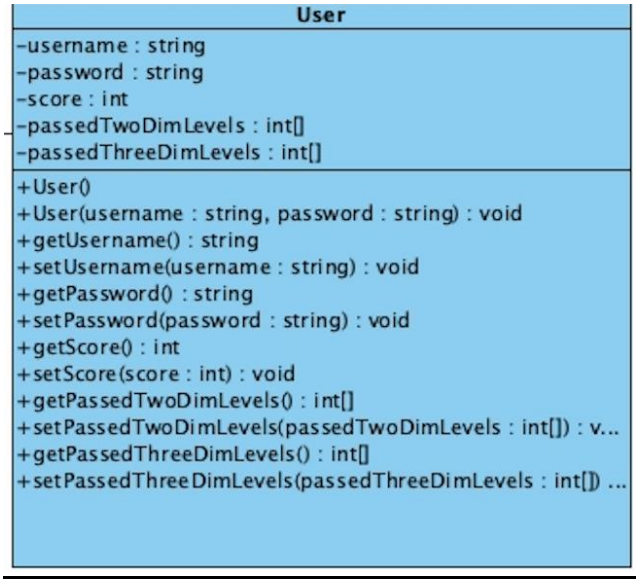
- **public void setHighestScore(int highestScore):** If user passes the previous highest score, this method updates his/her highest score.

- **public boolean isGameFinished():** This method simply checks if the game is over or not and returns 1 if game is over 0, otherwise.

- **public void nextLevel():** This method determines the next level and passes next level.

- **public void loadGame():** This method provides user the load a previous game.

User Class



Attributes:

- **private string username:** This attribute stores the name of user.

- **private string password:** This attribute stores the password of user.

- **private int score:** This attribute stores the score of user.

- **private int [] passedTwoDimLevels:** This attribute stores the passed 2Dlevels which user passed before.

- **private int [] passedThreeDimLevels:** This attribute stores the passed 3D levels which user passed before.

Constructors:

- **public User():** Default constructor for user class which initialize the class.

- **public User(string username, string password):** This constructor initialize the user class with given username and password.

Methods:

- **public string getUserName():** This method simply returns the username.

- **public void setUserName(string username):** This method simply sets the username with given parameter.

- **public string getPassword():** This method simply returns the password of the user.

- **public void setPassword(string password):** This method simply sets the password of user with given parameter.

- **public int getScore():** This method returns the score of user.

- **public void setScore(int score):** This method sets the score of user with given parameter.

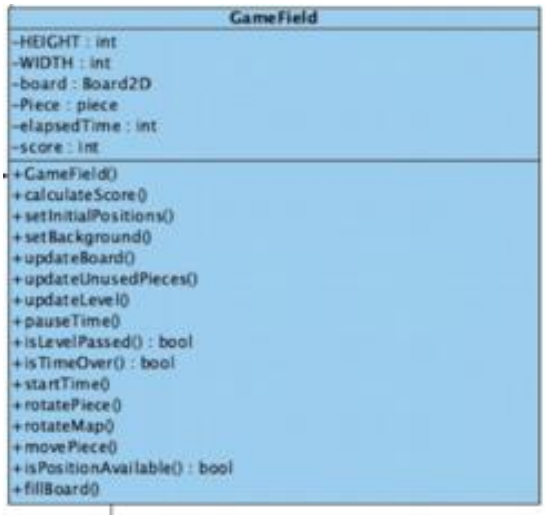
- **public int [] getPassedTwoDimLevels():** This method returns the user's passed 2D levels in an array in the 2D game mode.

- **public void setPassedTwoDimLevels(int [] passedTwoDimLevels):** This method sets the passedTwoDimLevels to update passed levels of user in 2D game mode.

- **public int [] getPassedThreeDimLevels():** This method returns the user's passed 3D levels in an array in the 3D game mode.

- **public void setPassedThreeDimLevels(int [] passedThreeDimLevels):** This method sets the passedTwoDimLevels to update passed levels of user in 3D game mode.

GameField Class



Attributes:

- **private const int HEIGHT:** This attribute stores the height of game window.
- **private const int WIDTH:** This attribute stores the width of game window.
- **private Board2D 2Dboard:** This attribute initializes the Board2D object which is for initializing the physical map of the current level in the 2D game mode.
- **private Board3D 3DBoard:** This attribute initializes the Board2D object which is for initializing the physical map of the current level in the 3D game mode.
- **private Piece piece:** This attribute is for initialization of Piece class.
- **private int elapsedTime:** This attribute is for keep track of elapsed time in the game which is 0 at the beginning of a level.
- **private int score:** This attribute is for keep track of score of current level**.**

Constructors:

- **public GameField()**: Default constructor which initializes the physical map of Game.

Methods:

- **public void calculateScore()**: This method calculates and updates the score when the level is passed by using elapsed time and number of movements.

- **public void setInitialPositions()**: This method sets the components initial positions in the window.

- **public void setBackground():** This method simply sets the background of the game by usi

- **public void updateBoard()**: When user moves a piece to board, this method updates the view of board

- **public void updateUnusedPieces():** When user pick a piece in the group of unused pieces, and move to the board, this method updates the unused puzzles. For example when user put a piece on the board this method should be able to erase put piece from the group of unused pieces.

- **public void updateLevel():** If user passes the current game, this method updates the level.

- **public void pauseTime():** This method simply pause the elapsed time.

- **public boolean isLevelPassed():** This method determines whether the game is passed or not.

- **public boolean isTimeOver:** This method determines whether elapsed time is passed the boundary time or not.

- **public void startTime():** This method simply starts to count the elapsed time in the current level.

- **public void rotatePiece():** This method simply rotates the picked piece.

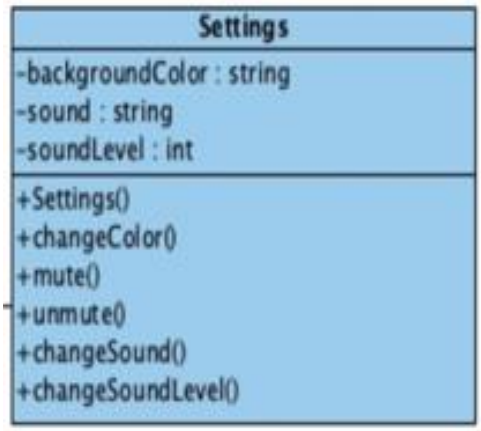
- **public void rotateMap()**: This method simply rotates the map. This method is for 3D game mode because in 2D game mode there is no need to rotate map.

- **public void movePiece():** This method simply moves the piece in the boundary of map.

- **public boolean isPositionAvailable():** When user wants to put the picked puzzle to the board, this method checks whether wanted area in the board is available or not.

- **public void fillBoard():**

Settings Class



Attributes:

- **private String backgroundColor:** This attribute stores the color of bacground in a string
- **private String sound:** This attribute stores the sound of the game
- **private int soundLevel:** This attribute keeps track of level of sound in the game

Constructors:

- **public Settings():** Default constructor which initialize the Settings class.

Methods:

- **public void changeColor():** This method changes the skin of board.
- **public void mute():** This method mutes the sound in the game.
- **public void unmute():** This method unmutes the sound if the sound is muted before.
- **public void changeSound():** This method changes the sound of game with another sound. ??
- **public void changeSoundLevel():** This method increase or decrease the level of sound.

Level Class

Attributes:

- **private int highestScore:** This attribute stores the highest score that is be made in a specific level. It's zero as default and Game updates it when user passes the level.

- **private int time:** As the user passes levels become more difficult. Therefore, every level has its unique time limit and this attribute keeps the time limit of a level.

- **private int levelNo**: This attribute stores the number of level.

- **private bool isLocked**: This attribute stores whether the current level is passed before or not. If it is passed before, or it is the last played level that is not finished yet, isLocked is false and true otherwise

- **private Piece [] unusedPieces**: This attribute stores the pieces that is not used, in other words the pieces that is not on the board.

- **private Piece [] usedPieces:** This attribute stores the usedPieces in the board. For example if player moves a piece and puts the piece to the board, this piece becomes a usedPiece. So usedPieces stores the pieces that is on the board.

Constructors:

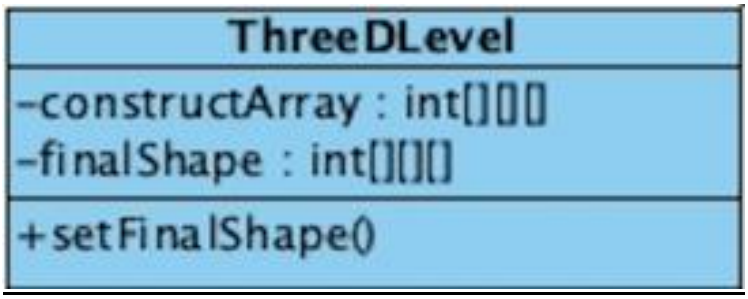
- **public Level():** Default constructor that initializes the Level object.

Methods:

- **public void updateScore()**: This method updates the highest score.

- **public void unlockLevel()**: If user passes a level, this method unlocks the level so that user can access this unlockedLevel. ??

ThreeDLevel Class



Attributes:

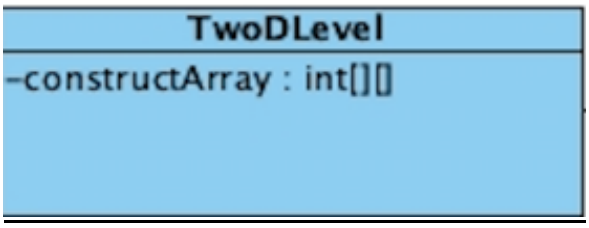
- **private int[][][] constructArray:** This array keeps the used pieces and occupied positions for each level specificly.

- **private int[][][] finalShape:** This attribute keeps the 3D matrix of final shape to decide whether the game finished or not.

Methods:

- **void setFinalShape():** Sets the finalShape array.

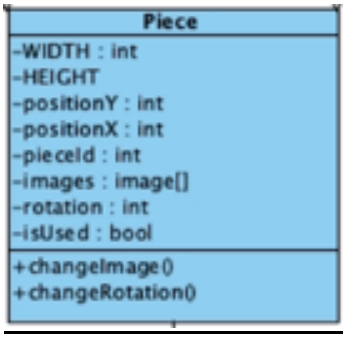
TwoDLeve Class



Attributes:

- **private int[][] constructArray:** This array keeps the used pieces and occupied positions for each level specificly.

Piece Class



Attributes:

- **private int WIDTH:** This attributes keeps the width of piece matrix as constant.

- **private int HEIGHT:** This attributes keeps the height of piece matrix as constant.

- **private int rotation:** This attribute stores the current rotation of the piece as integer to decide which image of the piece will be used.

- **private Image [] images**: This attribute keeps the images of the piece in from different rotations.

- **private int positionX**: This attribute holds the X coordinate of piece object to place it into proper position when game starts.

- **private int positionT**: This attribute holds the Y coordinate of piece object to place it into proper position when game starts.

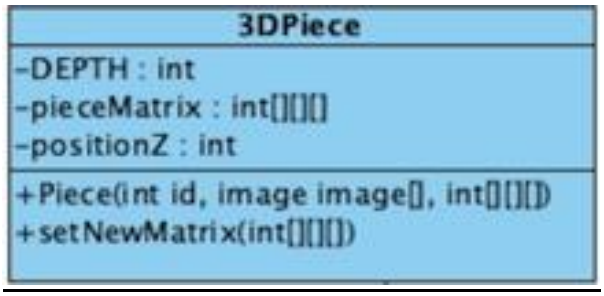
- **private int pieceID: T**his attribute holds the unique piece id to distunguish pieces.
- **private bool isUsed:** This attributes stores the information of piece about its current usage situation.

Methods:

- **void changeRotation():** This methods sets the new matrix and width-height values after its rotation.

- **void changeImage():** This methods sets the new image of piece after rotation.

3DPiece Class



Attributes:

- **private int DEPTH:** This attribute keeps the depth of piece matrix as constant.

- **private int[][][] pieceMatrix:** Keeps the size and boarders of piece in a 3D array, to properly place it on game board.

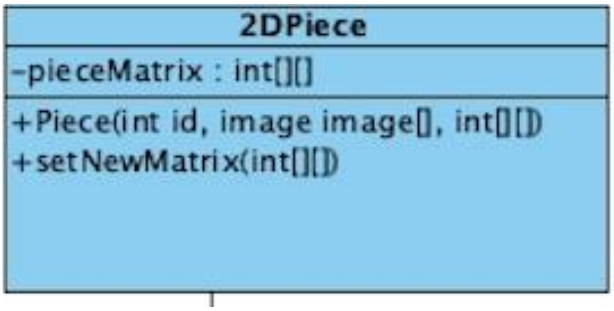
- **private int positionZ:** In addition to its 2D version 3D pieces also keeps the Z coordinate.

Methods:

- **Piece(): It initializes the piece with given images and values.**

- **void setNewMatrix(): This method sets the 3D new matrix of piece when its rotation change.**

2DPiece Class



Attributes:

- **private int[][] pieceMatrix:** Keeps the size and boarders of piece in a 2D array, to properly place it on game board.

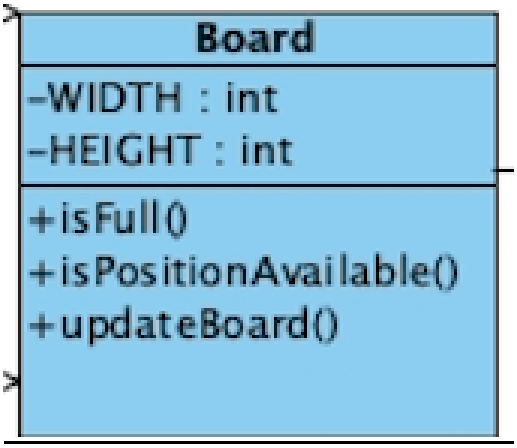
Constructors:

- **Piece():** It initializes the piece with given images and values.

Methods:

- **void setNewMatrix():** This method sets the new 2D matrix of piece when its rotation change.

Board Class



Attributes:

- private **int WIDTH:** This attributes keeps the width of Board matrix as constant.

- private **int HEIGHT:** This attributes keeps the height of Board matrix as constant.

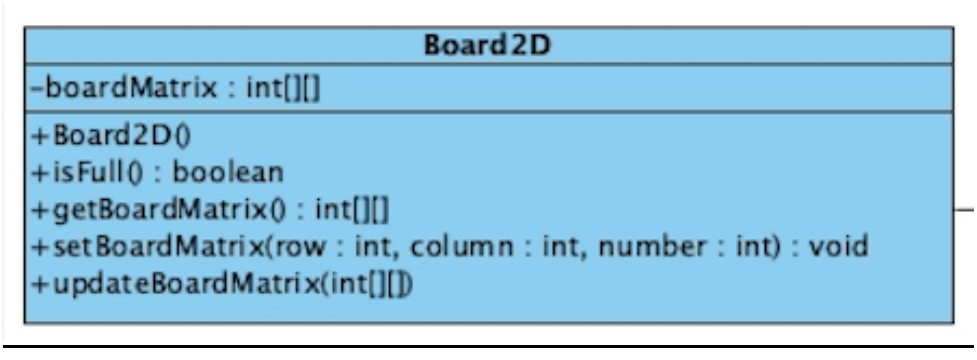
Methods:

- **boolean isFull():** This methods checks whether the board is full or not by using board matrix and returns boolean.

- **boolean isPositionAvailable():** Checks whether the position where user trying to place the piece is available or not.

- **void updateBoard():** Updates new board matrix by using piece objects matrix after the placement.

Board2D Class



Attributes:

- **private int[][] boardMatrix:** This attribute keeps the board matrix in 2D array.

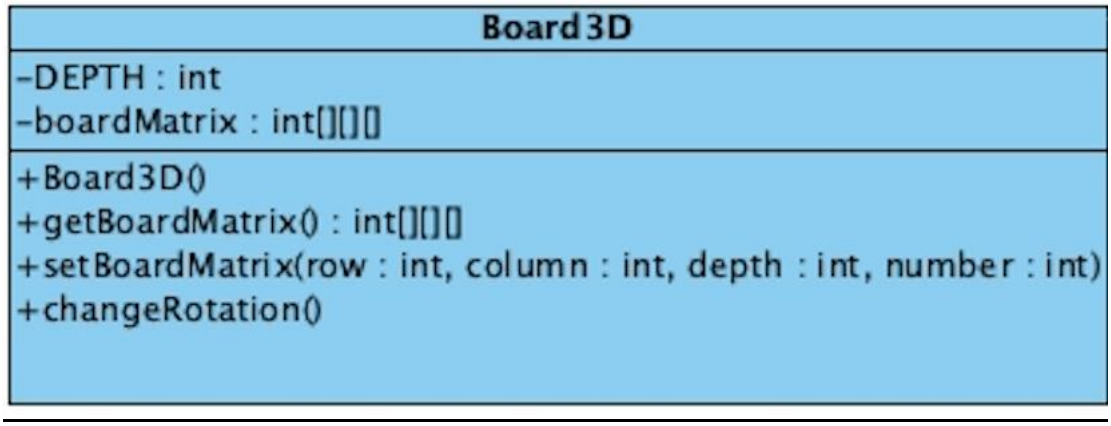
Methods:

- **public void updateBoardMatrix():** This methods updates the board matrix when a new piece placed into board.

- **public int[][] getBoardMatrix():** This method returns the board matrix.

- **public void setBoardMatrix():** This method sets the board matrix with initial conditions.

Board3D Class



Attributes:

- **private int[][][] boardMatrix:** This attribute keeps the board matrix in 3D array.

- **private int DEPTH** : Other than 2D board, this attribute keeps the depth of the 3D board matrix.

Methods:

- **public int[][][] getBoardMatrix():** This method returns the 3D board matrix.

- **public void setBoardMatrix():** This method sets the board matrix with initial conditions.

- **public void changeRotation() :** This method changes the 3D boards rotation with given commands.