Bilkent University

CS319 Term Project

Design Report
Section 3

Team Name: JOKERS Team

Project Name: IQ Puzzler Pro

Project Group Members:

- Burak Erdem Varol
- Ravan Aliyev
- Subhan Ibrahimli
- Ismayil Mammadov
- Mahmud Sami Aydin
- Cihan Erkan

1. Introduction

In this section purpose of the system and our main design goals are explained.

1.1 Purpose of the system

IQ Puzzler is a puzzle game which can be played as 2D and 3D. The purpose of our program is to create the digital version of already existing physical game and to make the game more attractive by adding different modes and creating new level to satisfy the users. In addition to the original version of the game, there are modes such as two player, 3D, and time. This game is planned to be multi-platform software ,which can be played on Microsoft Windows,Mac OS and Linux. It will be user-friendly, new players will be able to learn how to play the game easily. This game will challenge the players and entertain them by improving their inductive reasoning aptitude.The player is expected to put pieces on the board in a logical way, in order to arrive at the correct solution of the puzzle.

1.2 Design Goals

Design is one of the important processes for creating the software. It helps to define the software solution to sets of problems [1]. These problems are mentioned as non functional requirements in our analysis report which will be clarified in our design report. Design goals identify the qualities of our system that needs to be optimized. Designing goals for all aspects of the software for building is the main focus of this paper. Important design goals for our software is given in the following list:

Performance Goals: Response time of the software will be less than 100 ms. Memory needed for this software should be less than 500mb.

Maintenance Goals: The game should be platform independent, it can be played on MacOS, Windows and Linux systems in which JRE 8 is supported. Extensibility is another criteria of maintenance, which should be taken into account while designing our software to make it easier to add new functionality or new classes. So, other developers will be able to use our system to redesign as they desire since project-related documents will be open source.
Another maintenance criteria which should be taken into account is readability, which makes it easy to understand the system from reading the code.

Dependability Goals: Since, our system is very simple and does not contain any personal information we do not have security risks. Simplicity of this software improves its reliability by making specific and observed behaviors same and availability by increasing the percentage of the time that system can be used to accomplish tasks.

End User Goals: Usability is important from a users' point of view, so our system should be easier to learn and to use. 95% of the users should be able to adapt the system in 10 minutes.

2. High-level software architecture

2.1 Subsystem Decomposition

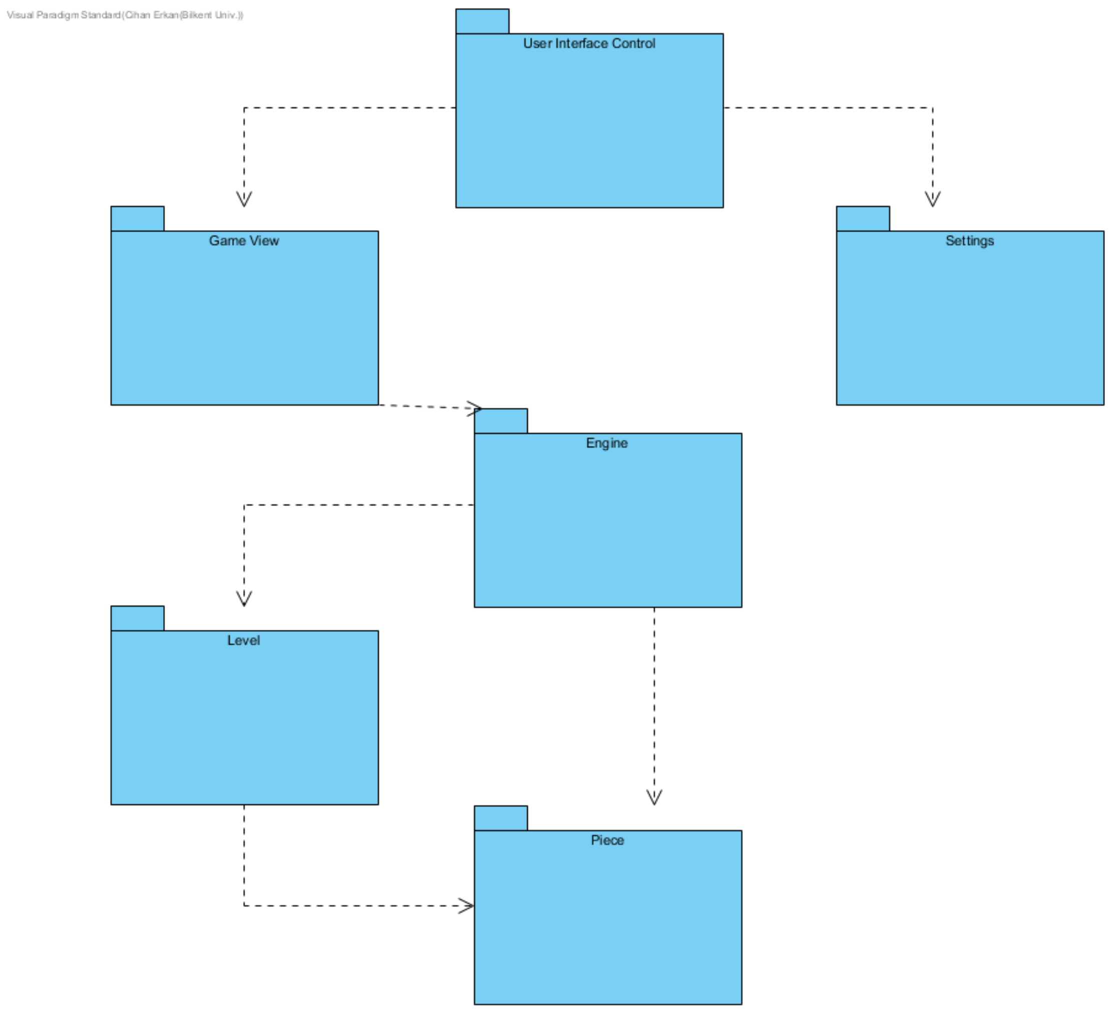


Figure 1 : Subsystem Decomposition Diagram

Subsystem of IQ Puzzler Pro will be composed of 6 subsystems and those subsystems will be distributed to 5 layers. While designing subsystems we tried to follow MVC design pattern and we aimed to achieve high coherence and low coupling.

Highest two layers of design are user interfaces of IQ Puzzler Pro. At highest level we have User Interface Controller subsystem. This subsystem controls simple user interfaces such as “Credits” and “How To Play” as well as more complex user interfaces, such as “Settings” and “Game View”. Instead of merging Settings and Game View subsystems with User Interface Controller, we designed them as separate subsystems in a lower layer. We separated Settings because, Settings is composed of many classes that are highly depend on each other, and merging those two subsystems would result in a low level of coherence. We also separated Game View from User Interface Controller subsystem for two reasons. First, even though both GameView and UserInterfaceController classes designed as user interfaces, they do not depend on each other and displaying game screen and menus are significantly different tasks. The second reason is, both GameView and UserInterfaceController are quite complex classes and thus, they require high amount of resources to implement. By designing them as separate subsystems we aimed to keep resources required for each subsystem at similar level.

In the middle layer we have Engine subsystem and this subsystem corresponds to Controller part of MVC design pattern. This subsystem is responsible for merging core components,such as levels and game pieces, of the game and allowing user to interact with the game by using keyboard. Engine subsystem has the highest coupling level among other subsystems. In order to reduce coupling we considered the option to merge Engine and Piece subsystems but we dismissed that option because Piece and Engine subsystems performs significantly different tasks. Therefore, we decided to keep Engine as a separate subsystem in the cost of increasing coupling.

In the lowest two layers of the design we have Piece and Level subsystems. Those subsystems corresponds to Model part of MVC and they are the foundations of our game. Because of that, they are at the lowest level of our design. Piece subsystem is responsible for creation and storage of game pieces. Game pieces are the core components of the game and properties of those pieces are defined in this subsystem. Level subsystem is responsible for defining properties of a certain level of the game, and it will provide specified game level to Engine subsystem. Moreover, Level subsystem is also responsible for creation of customly designed levels. We placed Level subsystem in a higher level compared to Pieces subsystem because Level subsystem depend on piece definitions provided by Piece subsystem. We did not merge those subsystems because only PolyminoList and Level classes are related, and merging those subsystem would result in a lower level of coherence.

2.2 Hardware/Software Mapping

In this project we did not use any external components and there is no distinction between user machines and servers. Therefore we did not do any hardware/software mapping.

2.3 Persistent Data Management

IQ Puzzler Pro requires persistent data storage in 4 different cases: Saving player's progress -which levels are solved-, saving custom and predefined levels, saving settings, saving achievements and highscores. In game data will not be saved because a typical IQ Puzzler Pro game does not last more that 5-6 minutes, therefore, it is not worth to save progress of a single game.

IQ Puzzler Pro does not require complex data to be saved. Players will have 3 distinct progresses that corresponds to our 3 game modes, and those progresses can be saved as 3 integers. Saving levels will require definitions and coordinates of pieces in three dimensions and the name of the level. Settings, highscores and achievements can be saved as a combination of strings and integers. Since every IQ Puzzler Pro game will be specific to its user, the game will not require concurrent access and multiple platforms and applications will not be accessing data. Moreover, the data we will be storing will have a very low density of information. Considering everything discussed above, we believe that using a database to store such data will be an overkill and it is not necessary for our case.Furthermore, using flat files will increase the performance of our application since it does not require any access process to a database. However, using .txt files as our persistent data management strategy, will allow users to access and modify the data. Modification of the data might corrupt the data stored, and it might cause whole application to lose its functionality. Nevertheless, IQ Puzzler Pro is a simple game, and we do not want to force our users to obtain Internet connection or a database application for such a simple game, therefore, we are going to use flat files, even though they introduce some risks to stored data.

2.4 Access Control & Security

The clarifications of subsystems of IQ Puzzler were written at section 2.1 as how and why the subsystems are separated in a certain manner. It is not mandatory for the users of the game to have an internet access in their devices since IQ Puzzler will not require any internet or network connection. Thus, if the execution of the game is accomplished to play, then anybody can freely play the game in their local devices. Since there is no need to have or the usage of internet access in the game, it will be risk-free. In the multiplayer mode it is also played locally in the same device. During system design, we model access by determining which exact information could be shared among users or restricted for others. For example, “Credits” and “How to play” subsystems are globally accessed to everyone and cannot be modified. Nevertheless, “Settings”, “Piece” and “Level” subsystems can be different dependent on the users. Therefore, the users will have an access to control the specific manners such as changing keyboard settings to play, or defining properties of a certain level according to the users.But at the end it will be checked by the “Engine” subsystems in order to determine if it is proper to execute users’ desires.

2.5 Boundary Conditions

In order to examine the boundary conditions of the system, we probe to decide how the system is executed , started, initialized and shutdown. Moreover, there could be some game failures that can be in consideration to handle when the issues appears such as disconnection in the middle of the game, data corruption and network problem. We need to determine potential software errors and how to handle it.

IQ Puzzler does not require any installation in personal computers. Therefore, it is portable among different devices. The game will be executed from .jar file and since the users will not need to use internet connection and the game is played through local device, there will be no network or connection problem in the game. There can also be cases in which the game is terminated either by the user or other issues can occur. In that case, some data can be destroyed such as the score of the previously played game, or the particular game settings which was set by the users. Therefore there can be a potential data loss during the experience of the game. Lastly, if the game is not started properly, some functions may not work as normal such as sound in the game, particular settings identified by the user and so on.

3. Subsystem Services

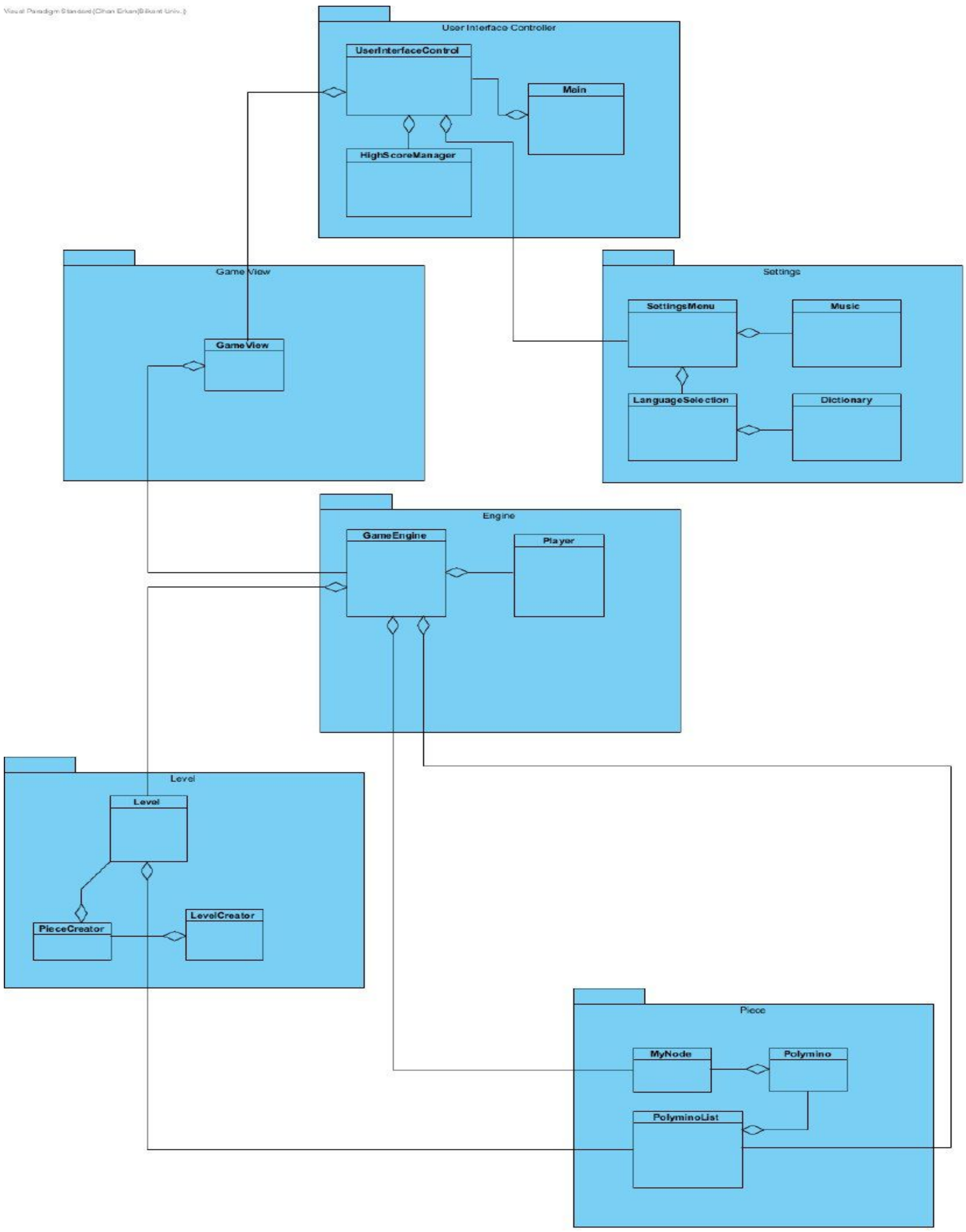


Figure 2 : Detailed Subsystem Diagram

3.1. UserInterfaceController Subsystem

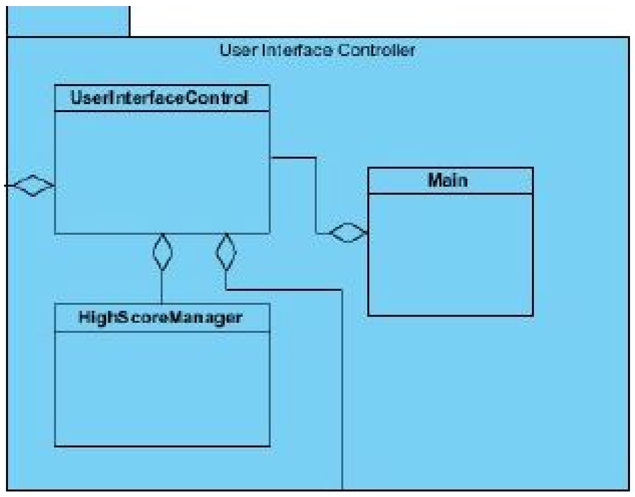


Figure 3 : UserInterfaceController Subsystem Diagram

User Interface Controller is responsible for creating connection between user and main functions of the system. In this project, User Interface System contains three classes:

1. UserInterfaceControl Class
2. Main Class
3. HighScoreManager Class

3.1.1 UserInterfaceControl Class

UserInterfaceControl class contains all screen in the game except gameview and levelCreator screens.In this class we use JavaFX library.

3.1.2 Main Class

Main Class contains Main Menu components which provide interaction user and the program. User can decide which game mode will play and quit game.

3.1.3 HighScoreManager Class

HighScoreManager class holds all high scores which achieved by users. This class will write and read high scores to/from txt file.

3.2 Game View Subsystem

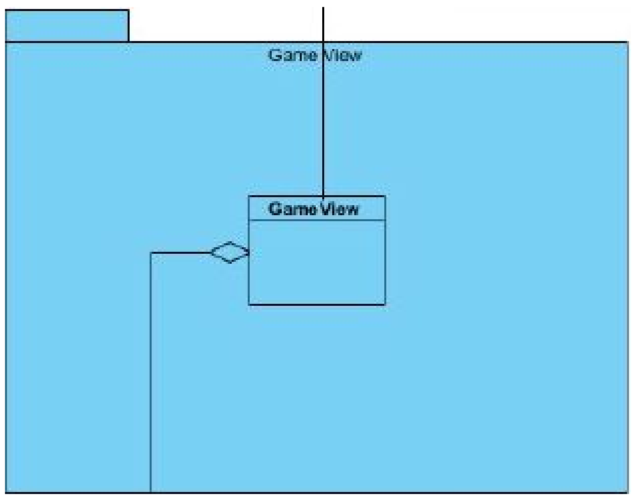


Figure 4 : GameView Subsystem Diagram

Game View is responsible for playing ground of the game. It provides interaction between user and game playing screen. It has one class: GameView Class.

3.2.1 GameView Class

GameView class provides visual representation of data produced by GameEngine class.

3.3 Settings Subsystem

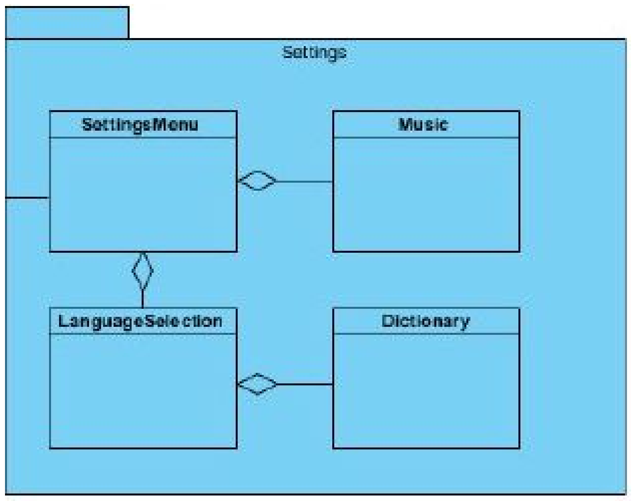


Figure 5 : Settings Subsystem Diagram

Settings package provides user to adjust game according to his/her enjoyment. It has 4 classes:

1. SettingsMenu Class
2. Music Class
3. LanguageSelection Class
4. Dictionary Class

3.3.1 SettingsMenu Class

SettingsMenu class provides settings components which contains adjusting/choosing audio/sound and language selection. In this class we used JavaFX library.

3.3.2 Music Class

Music Class serves user to choose his/her favorite music and play the game under the music sound. Apart from that, it also gives to adjust volume for the game.

3.3.3 LanguageSelection Class

LanguageSelection class provides adjusting language to desired language.

3.3.4 Dictionary Class

Dictionary class provides languages to LanguageSelection class.

3.4 Engine Subsystem

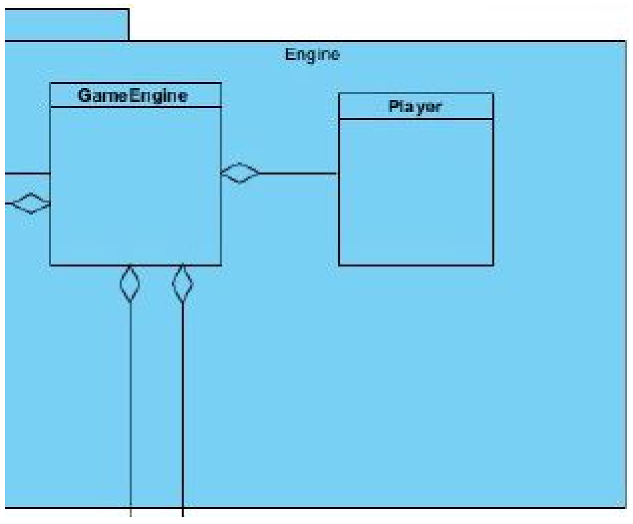


Figure 6 : Engine Subsystem Diagram

Engine package produce game dynamics for the playing game. It contains two classes:

1. GameEngine Class
2. Player Class

3.4.1 GameEngine Class

GameEngine class provides game logics which define how to play the game.

3.4.2 Player Class

Player class provides player name which will be used create a new players and their data. In addition to that, this will be used in multi-player part.

3.5 Level Subsystem

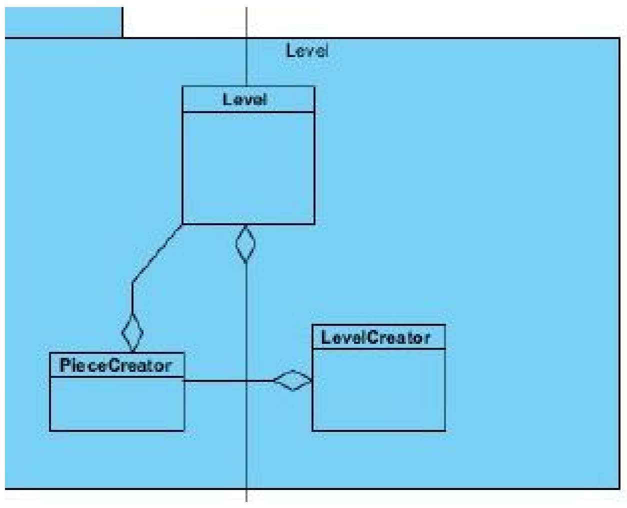


Figure 7 : Level Subsystem Diagram

Level Package provides user to interact with levels (i.e. to create levels, pieces and choose level). It has three classes:

1. Level Class
2. LevelCreator Class
3. PieceCreator Class

3.5.1 Level Class

Level class holds all levels in the game. According to users achievements it makes unavailable unachieved levels.

3.5.2 LevelCreator class

User can use his/her creativity to create new levels with the help of LevelCreator class. It provides tools to user for creating levels and also check whether created level is appropriate for the game or not (i.e. whether has solution or not).

3.5.3 PieceCreator Class

PieceCreator Class provides tools to user to create new pieces for the game according to their tastes.

3.6 Piece Subsystem

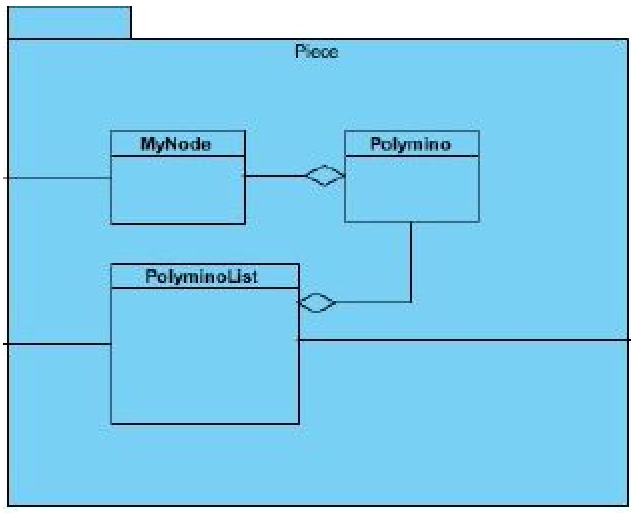


Figure 8 : Piece Subsystem Diagram

Piece package provides pieces for the playground. It has three classes:

1. MyNode Class
2. Polymino Class
3. PolyminoList Class

3.6.1 MyNode Class

MyNode class responsible for one node of the piece and checks neighborhood of this node.

3.6.2 Polymino Class

Polymino Class provides one figure from polymino list. It holds color of the piece, last action acted on the piece and so on.

3.6.3 PolyminoList Class

PolyminoList class holds all the polyminos in the game. In addition, new created pieces by the user will be holded in this class.

4. Low-Level Design

4.1 Object Design Trade-Offs

It is unrealistic to perform some design goals at the same time. So, design goals that need to be prioritized over another are described below:

Functionality vs Usability

In the game, we decide that functionality is more important. We designed the game to have 3 dimensional control which provides more realistic game instead of 2d game. Also we will use discrete game modal like classical tetris. Because putting pieces in three dimensional space without guide is hard in real life. Hence using discrete modal is avoiding illegal moves in the game. However, because of 3D discrete game cannot be handled by mouse control, we use only keyboard which delays users to reach point that comfortable with game control.

Efficiency v. Portability

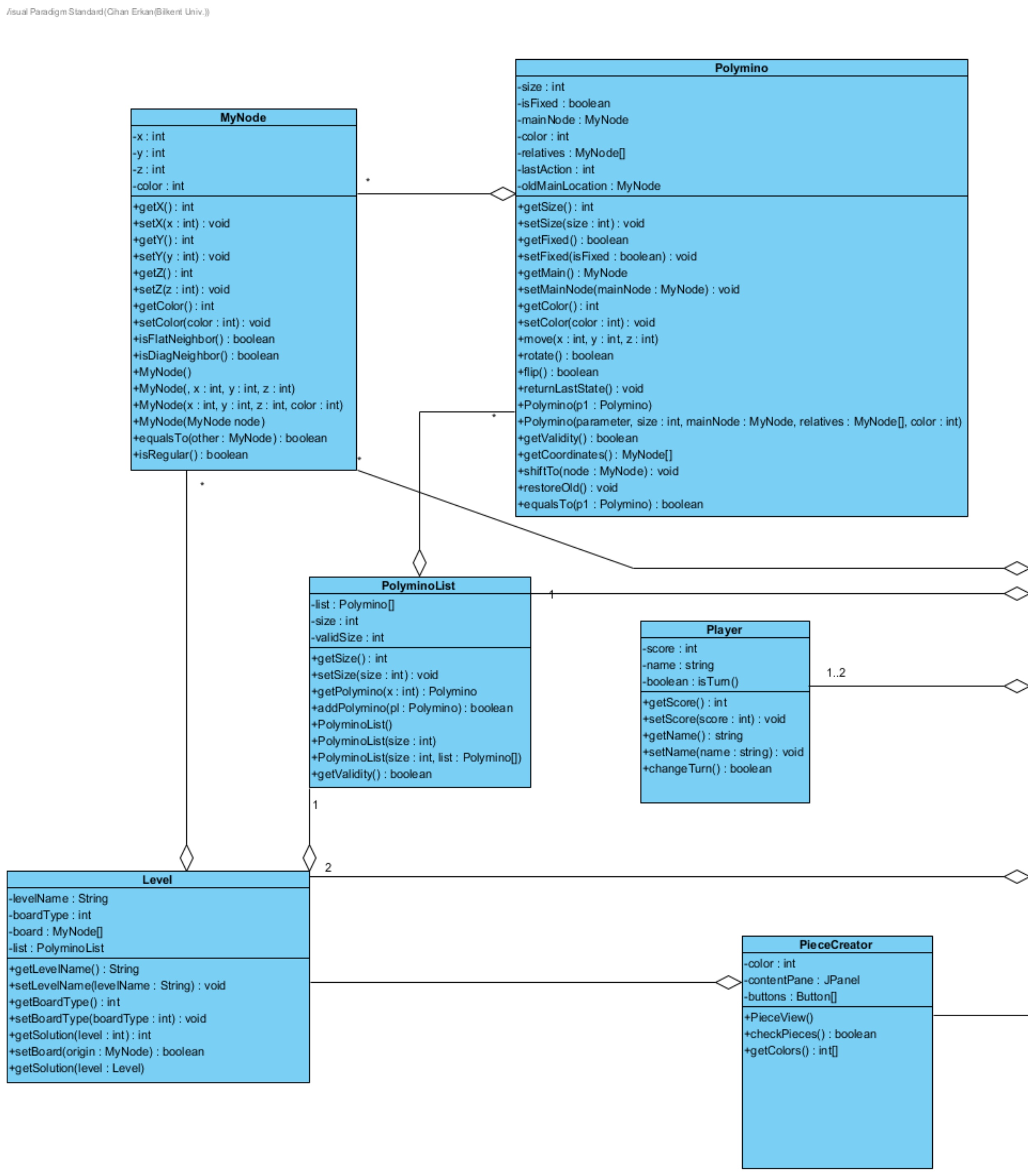
Because we will write our software in Java and this software will support all operating systems which support JRE 8. We cannot use operating system dependent codes to make software more efficient. We want to access more people. Hence we will prefer portability to efficiency.

Security vs Usability

We have no security concern because the game is very simple. We don’t use database, only need is file system for the levels. We don’t use user/password system in the game because it disrupts the player. Usability is main concern we don’t limit the player because of security issues.

4.2 Final Object Design

This is our final class diagram. We seperated it in two parts in order to keep it visible.



Visual Paradigm Standard(Cihan Erkan(Bilkent Univ.))
MyNode
-x : int
-y : int
-z : int
-color : int
+getX() : int
+setX(x : int) : void
+getY() : int
+setY(y : int) : void
+getZ() : int
+setZ(z : int) : void
+getColor() : int
+setColor(color : int) : void
+isFlatNeighbor() : boolean
+isDiagNeighbor() : boolean
+MyNode()
+MyNode(, x : int, y : int, z : int)
+MyNode(x : int, y : int, z : int, color : int)
+MyNode(MyNode node)
+equalsTo(other : MyNode) : boolean
+isRegular() : boolean
Polymino
-size : int
-isFixed : boolean
-mainNode : MyNode
-color : int
-relatives : MyNode[]
-lastAction : int
-oldMainLocation : MyNode
+getSize() : int
+setSize(size : int) : void
+getFixed() : boolean
+setFixed(isFixed : boolean) : void
+getMain() : MyNode
+setMainNode(mainNode : MyNode) : void
+getColor() : int
+setColor(color : int) : void
+move(x : int, y : int, z : int)
+rotate() : boolean
+flip() : boolean
+returnLastState() : void
+Polymino(p1 : Polymino)
+Polymino(parameter, size : int, mainNode : MyNode, relatives : MyNode[], color : int)
+getValidity() : boolean
+getCoordinates() : MyNode[]
+shiftTo(node : MyNode) : void
+restoreOld() : void
+equalsTo(p1 : Polymino) : boolean
PolyminoList
-list : Polymino[]
-size : int
-validSize : int
+getSize() : int
+setSize(size : int) : void
+getPolymino(x : int) : Polymino
+addPolymino(pl : Polymino) : boolean
+PolyminoList()
+PolyminoList(size : int)
+PolyminoList(size : int, list : Polymino[])
+getValidity() : boolean
Player
-score : int
-name : string
-boolean : isTurn()
+getScore() : int
+setScore(score : int) : void
+getName() : string
+setName(name : string) : void
+changeTurn() : boolean
Level
-levelName : String
-boardType : int
-board : MyNode[]
-list : PolyminoList
+getLevelName() : String
+setLevelName(levelName : String) : void
+getBoardType() : int
+setBoardType(boardType : int) : void
+getSolution(level : int) : int
+setBoard(origin : MyNode) : boolean
+getSolution(level : Level)
PieceCreator
-color : int
-contentPane : JPanel
-buttons : Button[]
+PieceView()
+checkPieces() : boolean
+getColors() : int[]
*
*
*
*
1
1..2
1
2

Figure 9a : Final Class Diagram

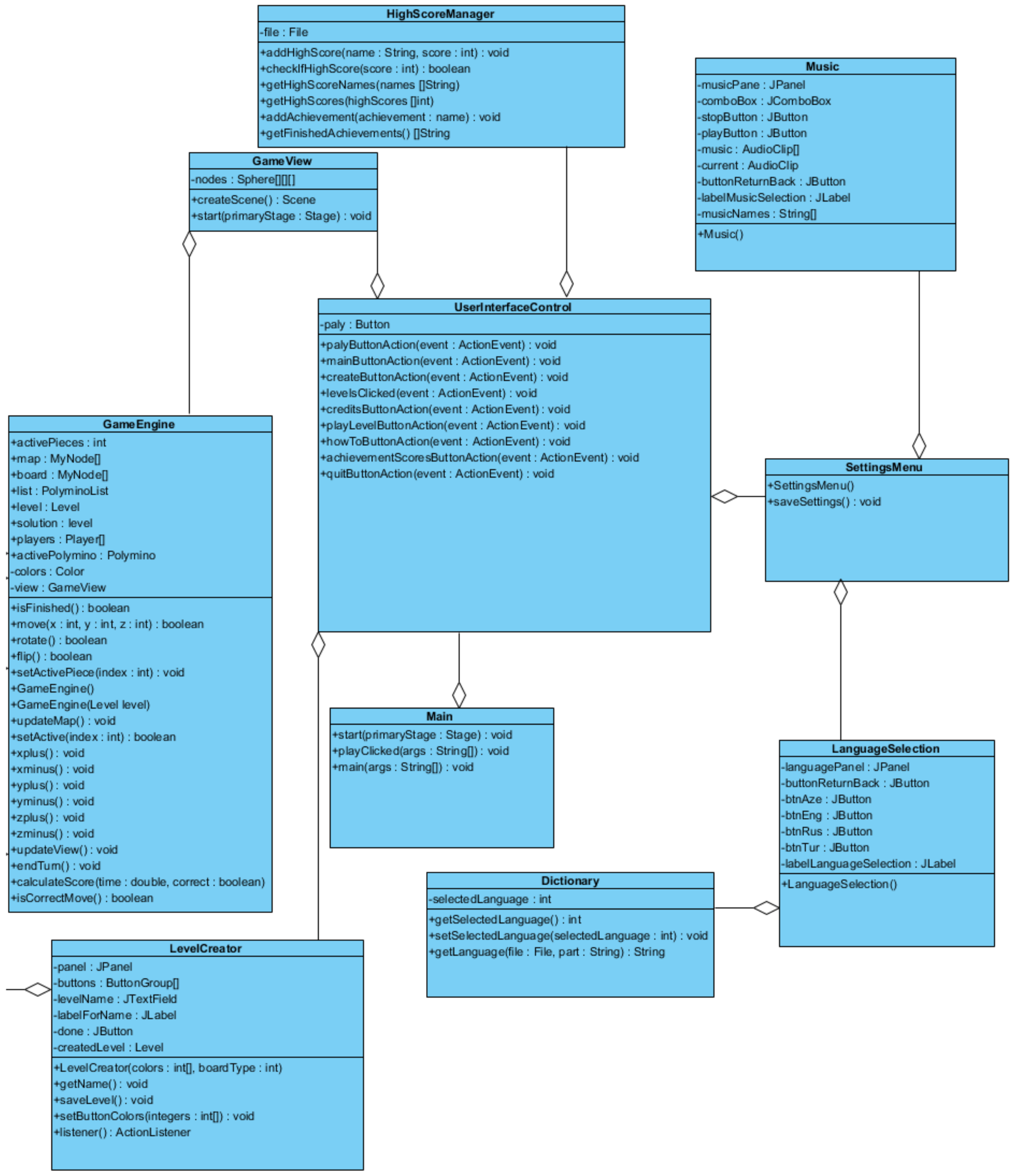

HighScoreManager
-file : File
+addHighScore(name : String, score : int) : void
+checkIfHighScore(score : int) : boolean
+getHighScoreNames(names []String)
+getHighScores(highScores []int)
+addAchievement(achievement : name) : void
+getFinishedAchievements() []String
Music
-musicPane : JPanel
-comboBox : JComboBox
-stopButton : JButton
-playButton : JButton
-music : AudioClip[]
-current : AudioClip
-buttonReturnBack : JButton
-labelMusicSelection : JLabel
-musicNames : String[]
+Music()
GameView
-nodes : Sphere[][][]
+createScene() : Scene
+start(primaryStage : Stage) : void
UserInterfaceControl
-paly : Button
+palyButtonAction(event : ActionEvent) : void
+mainButtonAction(event : ActionEvent) : void
+createButtonAction(event : ActionEvent) : void
+levelsClicked(event : ActionEvent) : void
+creditsButtonAction(event : ActionEvent) : void
+playLevelButtonAction(event : ActionEvent) : void
+howToButtonAction(event : ActionEvent) : void
+achievementScoresButtonAction(event : ActionEvent) : void
+quitButtonAction(event : ActionEvent) : void
GameEngine
+activePieces : int
+map : MyNode[]
+board : MyNode[]
+list : PolyminoList
+level : Level
+solution : level
+players : Player[]
+activePolymino : Polymino
-colors : Color
-view : GameView
+isFinished() : boolean
+move(x : int, y : int, z : int) : boolean
+rotate() : boolean
+flip() : boolean
+setActivePiece(index : int) : void
+GameEngine()
+GameEngine(Level level)
+updateMap() : void
+setActive(index : int) : boolean
+xplus() : void
+xminus() : void
+yplus() : void
+yminus() : void
+zplus() : void
+zminus() : void
+updateView() : void
+endTurn() : void
+calculateScore(time : double, correct : boolean)
+isCorrectMove() : boolean
SettingsMenu
+SettingsMenu()
+saveSettings() : void
Main
+start(primaryStage : Stage) : void
+playClicked(args : String[]) : void
+main(args : String[]) : void
LanguageSelection
-languagePanel : JPanel
-buttonReturnBack : JButton
-btnAze : JButton
-btnEng : JButton
-btnRus : JButton
-btnTur : JButton
-labelLanguageSelection : JLabel
+LanguageSelection()
Dictionary
-selectedLanguage : int
+getSelectedLanguage() : int
+setSelectedLanguage(selectedLanguage : int) : void
+getLanguage(file : File, part : String) : String
LevelCreator
-panel : JPanel
-buttons : ButtonGroup[]
-levelName : JTextField
-labelForName : JLabel
-done : JButton
-createdLevel : Level
+LevelCreator(colors : int[], boardType : int)
+getName() : void
+saveLevel() : void
+setButtonColors(integers : int[]) : void
+listener() : ActionListener

Figure 9b : Final Class Diagram

4.3 Packages

For our implementation, we are going to define our packages for main components of our game. However, for visual aspects of our game, we are going to use JavaFX libraries. In this section we briefly described our new packages because they are already explained in Subsystem Services section.

4.3.1 New Packages

Packages which is written by team members.

4.3.1.1 Pieces Package

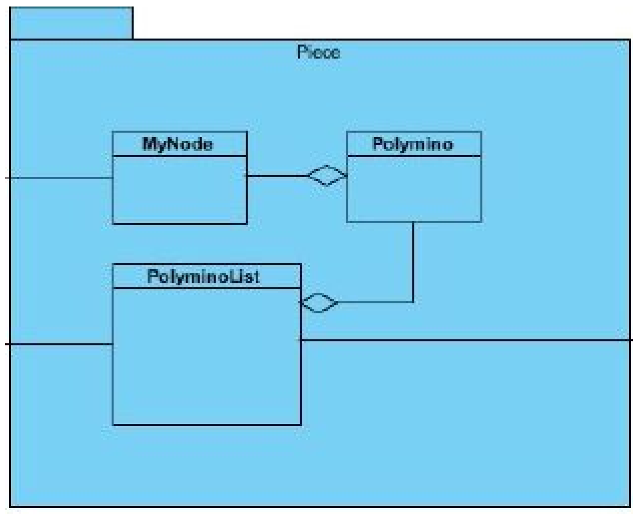


Figure 10 : Pieces Package

This package contains basic pieces, that user moves and places on board, of IQ Puzzler Pro and determines basic behaviour of them.

4.3.1.2 Level Package

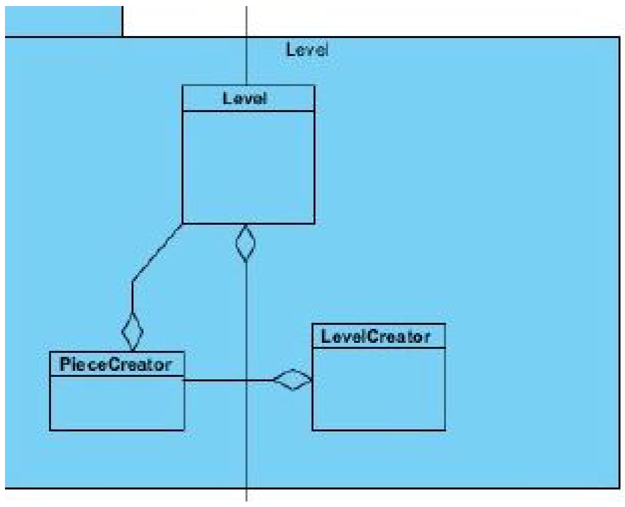


Figure 11 : Level Package Diagram

This package includes classes related to levels, such as creating, storing and loading.

4.3.1.3 Engine Package

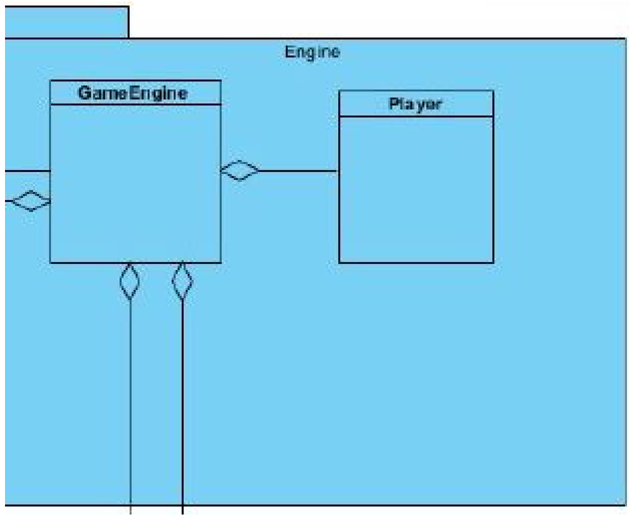


Figure 12 : Engine Package Diagram

This package contains the main parts that makes game playable. Interaction of pieces with each other and game environment will be defined in this package.

4.3.1.4 GameView Package

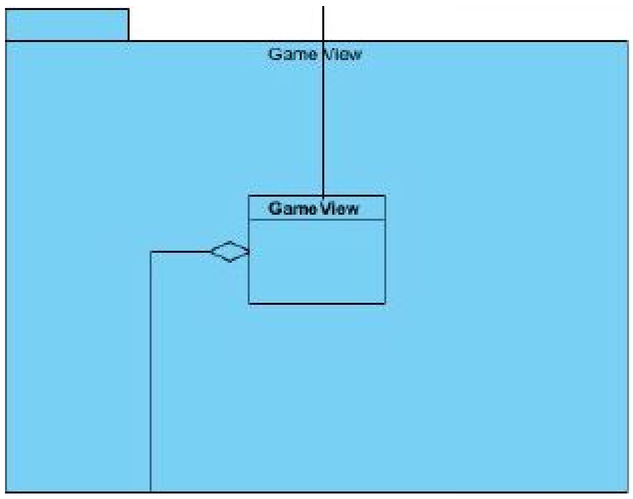


Figure 13 : GameView Package Diagram

This package contains everything related to GUI of the game.

4.3.1.5 UserInterfaceControl Package

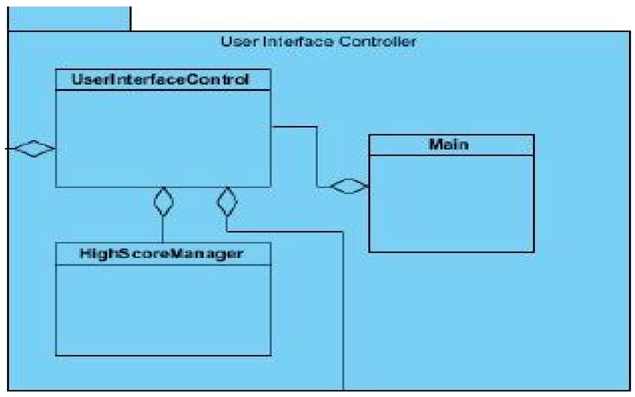


Figure 14 : UserInterface Controller Package Diagram

This package manages the user interface and general bounds of the game.

4.3.1.6 Settings Package

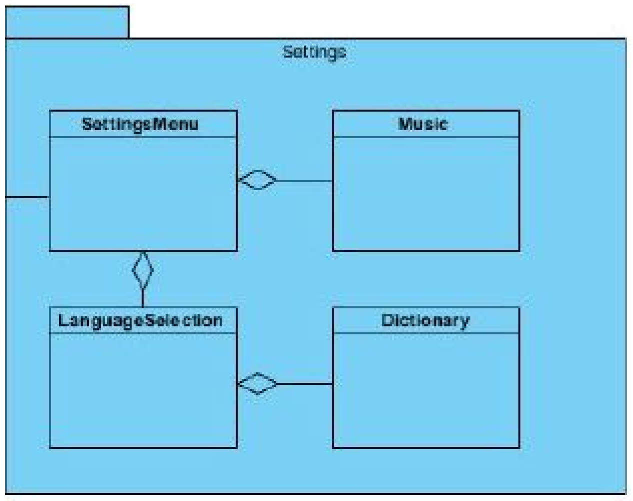


Figure 15 : Detailed Subsystem Diagram

This package includes parts which is related to the settings of the game.

4.3.2 Existing Packages

4.3.2.1 java.util Package

We are going to use this package to be able to use main functionalities of Java such arrayLists and Timer objects.

4.3.2.2 javafx.event

We are going to use this package for event handling including key and mouse events

4.3.2.3 javafx.application

We are going to use this package to initialize our application.

4.3.2.4 javafx.scene

We are going to use this package to present our game environment. It provides specifically cameras, groups, shapes, input and transformations.

4.3.2.5 javafx.scene.shape

We are going to use 3D shapes such as box and spheres to represent board and the pieces.

4.3.2.6 javafx.scene.input

This package provides key and mouse listener to interact with player

4.3.2.7 javafx.geometry

This package provides bounds of object in the application.[2]

4.4 Class Interfaces

In this section we explained interfaces of our classes in detail.

4.4.1 MyNode Class

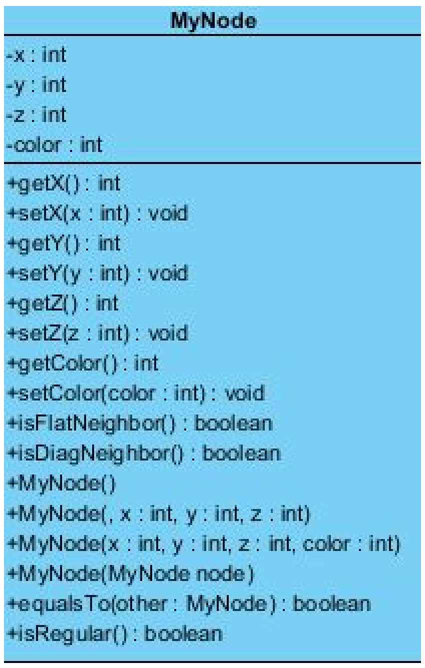


Figure 16 : MyNode Class Diagram

Attributes:

- int x : represents x coordinate of the node
- int y : represents y coordinate of the node
- int z : represents z coordinate of the node
- int color: represents the color of the node

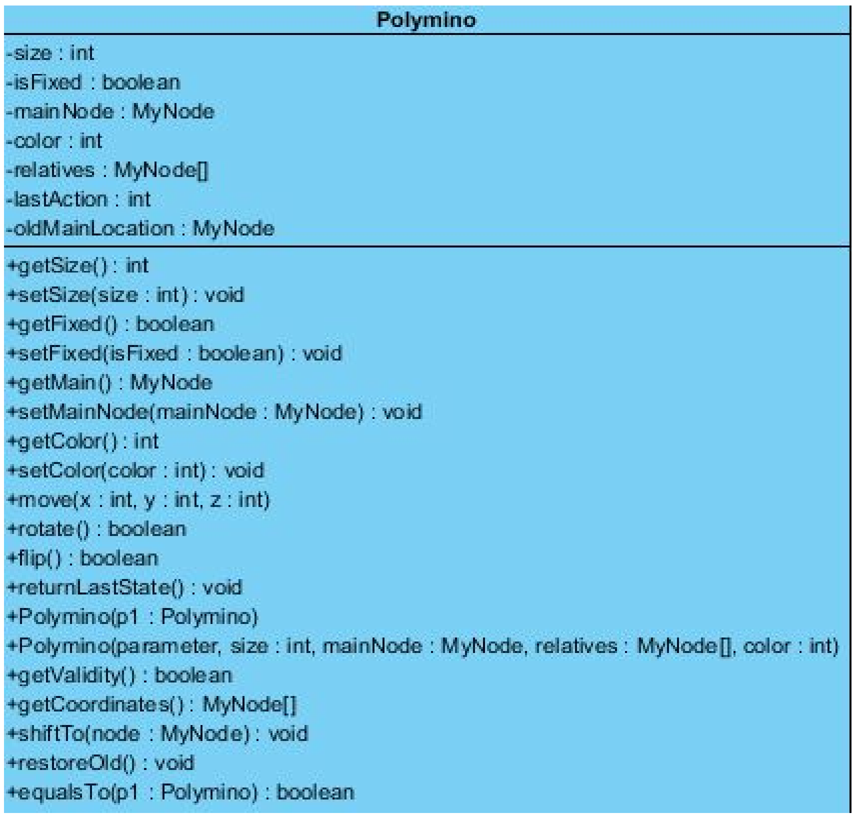
Constructors:

- MyNode (int x, int y, int z, int color) : Creates a MyNode with specified x, y, z values and color.
- MyNode () : Default constructor for MyNode class.
- MyNode (MyNode node) : Copy Constructor for MyNode class.

Methods:

- boolean isFlatNeighbor(MyNode node) : Checks if node is a flat neighbor, i.e. angle between two nodes is 90° and it is impossible to place another node between two nodes.
- boolean isDiagNeighbor(MyNode node) : Checks if node is a diagonal neighbor, i.e. angle between two nodes is 45° and it is impossible to place another node between two nodes.
- boolean equalsTo(MyNode other) : Checks if attributes of two nodes are equal to each other.
- boolean isRegular() : Checks if the node is a regular node or if it placed in diagonal zone.

4.4.2 Polymino Class



```
Polymino
-size : int
-isFixed : boolean
-mainNode : MyNode
-color : int
-relatives : MyNode[]
-lastAction : int
-oldMainLocation : MyNode
+getSize() : int
+setSize(size : int) : void
+getFixed() : boolean
+setFixed(isFixed : boolean) : void
+getMain() : MyNode
+setMainNode(mainNode : MyNode) : void
+getColor() : int
+setColor(color : int) : void
+move(x : int, y : int, z : int)
+rotate() : boolean
+flip() : boolean
+returnLastState() : void
+Polymino(p1 : Polymino)
+Polymino(parameter, size : int, mainNode : MyNode, relatives : MyNode[], color : int)
+getValidity() : boolean
+getCoordinates() : MyNode[]
+shiftTo(node : MyNode) : void
+restoreOld() : void
+equalsTo(p1 : Polymino) : boolean
```

Figure 17 : Polymino Class Diagram

Attributes:

- int size : Number of the nodes in the polyomino
- boolean isFixed : Checks whether polyomino has fixed point on the board
- MyNode mainNode: Indicates the coordinates of the main node of the polymino. Remaining nodes of a polymino defined by their relative position to main node.
- int color : Represents the color of the polyomino
- Node[ ] relatives : Relatives is an array of MyNode objects. However, their locations are not their actual location, but their relative locations to main node of the polymino.
- int lastAction : Shows if the last action was flip, rotate or move. Represented as 2, 1 or 0 respectively.
- Node oldMainLocation : Holds position of the main node before last move.

Constructors:

- Polyomino(int size, MyNode mainNode, MyNode[] relatives, int color): Constructor for Polymino class. Creates a polymino with specified attributes.
- Polyomino(Polymino polymino): Copy constructor for Polymino class.

Methods:

- boolean move(int x, int y, int z) : Moves polymino to given x,y,z direction. Final position of main node will be (x0+x, y0+y, z0,z). Returns true if successful.
- boolean rotate(): Rotates polymino by holding coordinates of main node constant, according to classical rotation with respect to a fixed point -main node- algorithms used in mathematics[1]. Rotation takes place in XY plane. Returns true if successful.
- boolean flip(): uses same principles as in rotate() method. However, this time rotation takes place in XZ or YZ plane depending on initial positioning of polymino.

- void returnLastState() : Takes the last action back, and restores previous position of polymino.
- boolean getValidity () ; Returns true if specified nodes for polymino are appropriate, i.e. all nodes must be connected to main node directly or indirectly via their flat or diagonal neighbors.
- [] MyNode getCoordinates() : returns all nodes that composes the polymino.
- void shiftTo(MyNode node) : Shifts main node to the location of node. All relative nodes move their positions according to their relative positions to main node.
- boolean equalsTo (Polymino polymino) : returns true if two polyminos are same.

4.4.3 PolyminoList Class

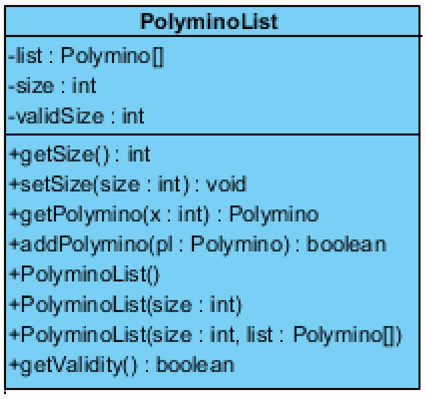

PolyminoList
-list : Polymino[]
-size : int
-validSize : int
+getSize() : int
+setSize(size : int) : void
+getPolymino(x : int) : Polymino
+addPolymino(pl : Polymino) : boolean
+PolyminoList()
+PolyminoList(size : int)
+PolyminoList(size : int, list : Polymino[])
+getValidity() : boolean

Figure 18 : PolyminoList Class Diagram

Attributes:

- Polymino[] list : Holds a fixed size of array of polyminos. Since maximum number of polyminos are limited, using a fixed sized array is appropriate.
- int size : Maximum number of the polyominos in list.
- int validSize : Since the array is a fixed size array, this attribute allows program to trace array.

Constructors:

- PolyominoList(int size) : Initializes a list with given maximum size.
- PolyominoList(int size, Polyomino[] list) : creates a list with given maximum size and adds polyminos to list.

Methods:

- Polymino getPolymino(int x) : Return the polyomino in the list with index x
- boolean addPolymino (Polyomino pl) : adds polyomino to the list and returns true if it is done successfully
- boolean getValidity () : Returns true if all polyminos in list is valid.

4.4.4 Level Class

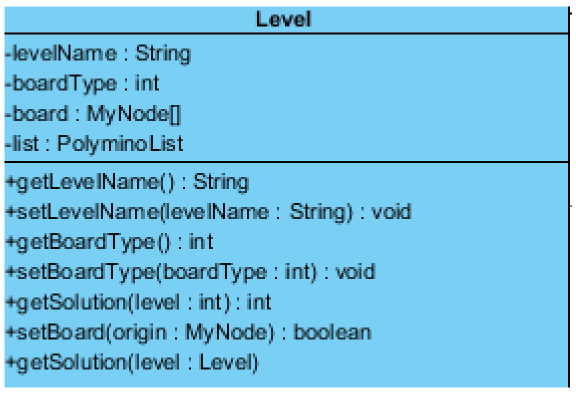

Level
-levelName : String
-boardType : int
-board : MyNode[]
-list : PolyminoList
+getLevelName() : String
+setLevelName(levelName : String) : void
+getBoardType() : int
+setBoardType(boardType : int) : void
+getSolution(level : int) : int
+setBoard(origin : MyNode) : boolean
+getSolution(level : Level)

Figure 19 : Level Class Diagram

Attributes:

- string levelName: Stores the name of the level in order to retrieve level from file.
- int boardType: Indicates the board type: flat , diagonal or 3d

- PolyominoList list: Polyominoes which does not contain (-1,-1,-1) coordinate, are the fixed ones at the beginning of the level. The ones with (-1,-1,-1) coordinate will be shown randomly on the game map.

Constructor:

- Level(File level, String name) : Constructor for Level class, retrieves level from file.
- Level(PolyominoList list, String name, int boardType) : Constructor for Level class, creates level with given PolyminoList.

Methods:

- int getSolution(int level): finds number of solutions for the level recursively and returns it.
- Level getSolution(Level level): returns a solution for the level recursively

4.4.5 GameEngine Class

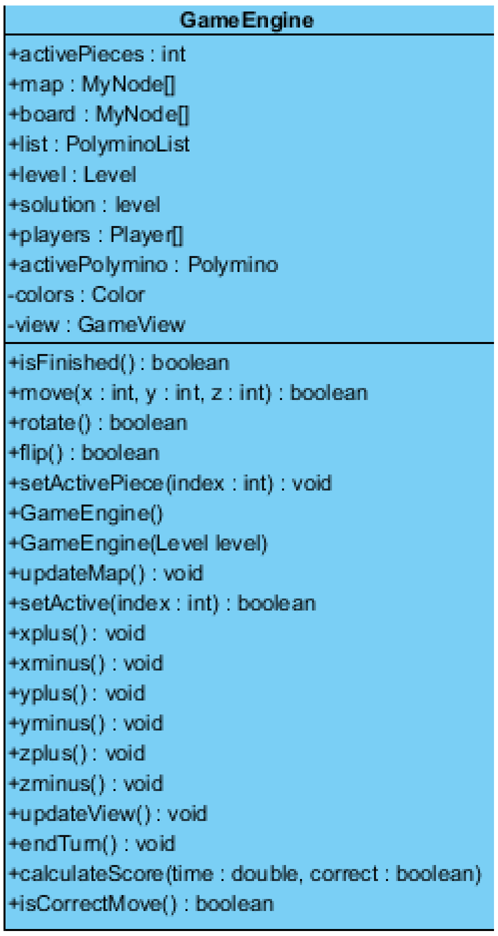


Figure 20 : GameEngine Class Diagram

Attributes:

- int activePieces: Indicate the index of which polyomino to move, flip or rotate.
- Node[] map: Coordinate system for the game map.
- Node [] board: Board which should be filled by pieces.
- PolyominoList list: Contains list of the pieces.
- Level level: indicates which level to play.
- Level solution: solution of the level for the 2 player game.

Constructor:

- GameEngine(Level level) : Creates a new game by using level.

Methods:

- boolean isFinished(): returns the boolean to show whether the game is finished
- boolean move(int x, int y, int z): moves the polyomino according to given parameters and returns whether is successful
- boolean rotate(): rotates the polyomino and returns whether is successful.
- boolean flip(): Flips the polyomino and returns whether is successful.
- void updateMap() : Updates map.
- void xplus() : Increases x coordinate of active polymino by 1.
- void xminus() : Decreasesx coordinate of active polymino by 1
- void yplus() : Increases y coordinate of active polymino by 1
- void yminus() : Decreases y coordinate of active polymino by 1
- void zplus() : Increases z coordinate of active polymino by 1
- void zminus() : Decreases z coordinate of active polymino by 1
- void endTurn() : Ends the turn of the player in multiplayer mode.
- void CalculateScore(double time, boolean correct) : Calculates player’s score for 1 turn by using time spent and correctness of move.
- boolean isCorrectMove() : In player mode, checks if the move of player is correct or not.

4.4.6 Player Class

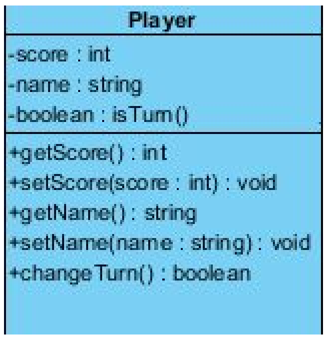


Figure 21 : Player Class Diagram

Attributes:

- int score: indicate the score of the player
- String name: indicates name of the player
- boolean isTurn: indicates if it is player's turn.

Constructor:

- Player(String name) : Creates a Player object with given name.

Methods:

- boolean changeTurn( ): makes it other players turn.

4.4.7 UserInterfaceControl Class

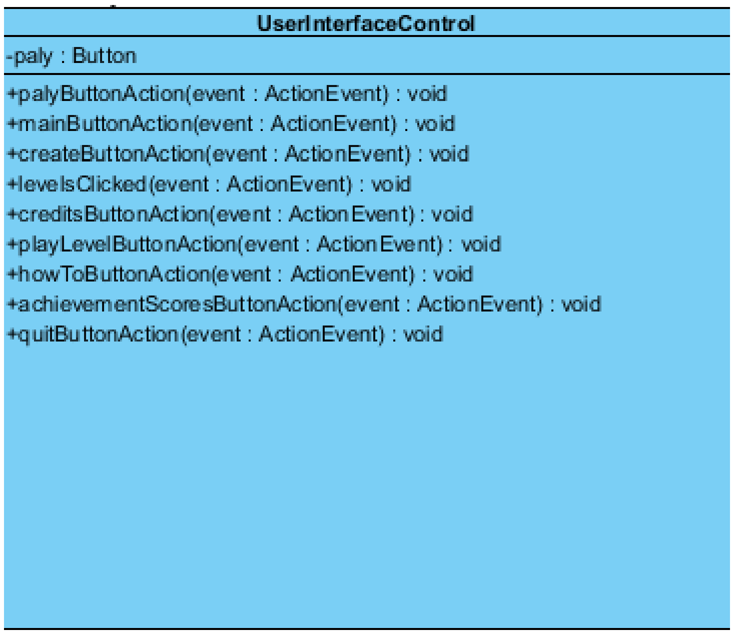

UserInterfaceControl
-paly : Button
+palyButtonAction(event : ActionEvent) : void
+mainButtonAction(event : ActionEvent) : void
+createButtonAction(event : ActionEvent) : void
+levelsClicked(event : ActionEvent) : void
+creditsButtonAction(event : ActionEvent) : void
+playLevelButtonAction(event : ActionEvent) : void
+howToButtonAction(event : ActionEvent) : void
+achievementScoresButtonAction(event : ActionEvent) : void
+quitButtonAction(event : ActionEvent) : void

Figure 22 : UserInterfaceControl Class Diagram

Attributes:

- JButton play: Start the game

Methods:

- void playButtonAction(ActionEvent event): Start the game with this button
- void mainButtonAction(ActionEvent event): Go to main menu with this button
- void createButtonAction(ActionEvent event): When level design is completed, level will created by this button.
- void levelsClicked(ActionEvent event): It takes clicked levels in Levels part
- void creditsButtonAction(ActionEvent event): User goes to Credits window from main menu with this button.
- void playLevelButtonAction(ActionEvent event): User will play selected level with this button

- void howToButtonAction(ActionEvent event): User can get information about how to play the game with this button.
- void achievementScoresButtonAction(ActionEvent event): User can get information about achievements with this button
- void quitButtonAction(ActionEvent event): User will quit the game with this button

4.4.8 Music Class

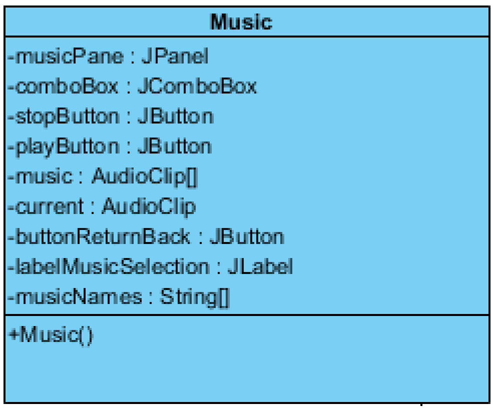


Figure 23 : Music Class Diagram

Attributes:

- JPanel musicPane: Panel which holds music selection menu
- JComboBox comboBox: Shows available musics that user can choose from
- JButton stopButton: Stops the music
- JButton playButton: Plays the music
- AudioClip[] music: contains all musics
- AudioClip current: holds current music which is playing in the background
- JButton buttonReturnBack: return back to the main menu
- JLabel labelMusicselection: Displays name of the current menu (Music Selection)
- String[] musicName: String array that holds music names to display to the user

Methods:

- void Music(): Plays the selected music

4.4.9 LanguageSelection Class

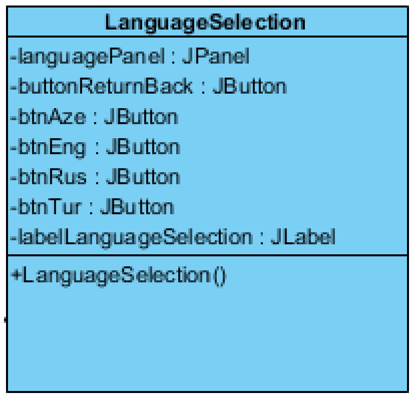


Figure 24 : MyNode Class Diagram

Attributes:

- JPanel languagePanel: Panel which holds the language selection menu
- JButton buttonReturnBack: return to the main menu
- JButtton btnAze: button to choose Azerbaijani language
- JButtton btnEng: button to choose English language
- JButtton btnRus: button to choose Russian language
- JButtton btnTur: button to choose Turkish language
- JLabel labelLanguageSelection: Label that displays the name of the current menu

Methods:

- void LanguageSelection(): Change language according to users opinion

4.4.10 Settings Menu Class

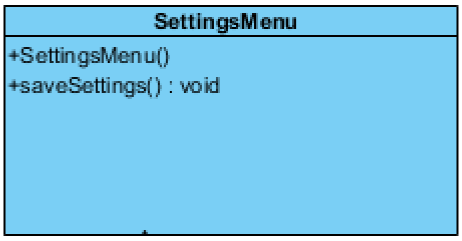


Figure 25 : SettingMenu Class Diagram

Attributes:

Constructor:

SettingsMenu(): Constructor for the setting menu initializes view

Methods:

void saveSettings(): It save adjusted settings to the game.

4.4.11 Dictionary Class

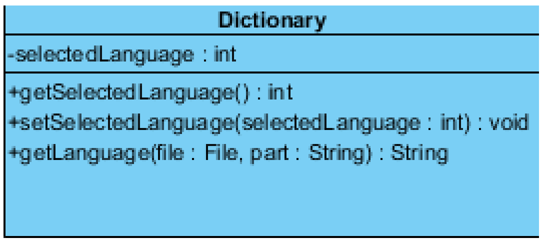

Dictionary
-selectedLanguage : int
+getSelectedLanguage() : int
+setSelectedLanguage(selectedLanguage : int) : void
+getLanguage(file : File, part : String) : String

Figure 26 : Dictionray Class Diagram

Attributes:

- int selectedLanguage : Specifies which language is selected by user.

Methods:

- String getLanguage (File file, String part) : Returns the specified part in given language

4.4.12 HighScoreManager Class

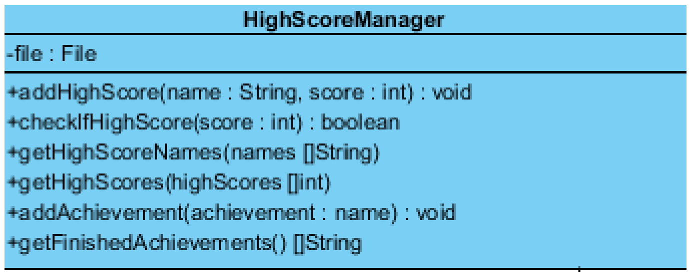

HighScoreManager
-file : File
+addHighScore(name : String, score : int) : void
+checkIfHighScore(score : int) : boolean
+getHighScoreNames(names []String)
+getHighScores(highScores []int)
+addAchievement(achievement : name) : void
+getFinishedAchievements() []String

Figure 27 : HighScore Manager Class Diagram

Attributes:

- File file : This is the file that stores high scores and achievements

Methods:

- void addHighScore(String name, int score) : Adds an highscore to list in a proper place with name.
- boolean checkIfHighScore (int score) : Checks if the given score could be placed in highscore list.
- []String getHighScoreNames() : Returns an array of names in highscores list in order.
- []int getHighScores() : Returns an array of highscores in highscores list in order.
- void addAchievement(String name) : Adds specified achievement.
- []String getAchievements() : Returns list of achievements earned by player.

4.4.13 GameView Class

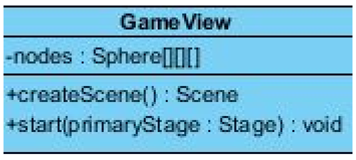


Figure 28 : GameView Class Diagram

Attributes:

- Shpere[][][] nodes: This is list of 3D shapes which is representation of the pieces and the board.

Methods:

- Scene createScene() : This method returns scene of the game it is initialize the scene.
- void start(Stage primaryStage) : This method is inherited from javaFX application which initialize the game.

4.4.14 LevelCreator Class

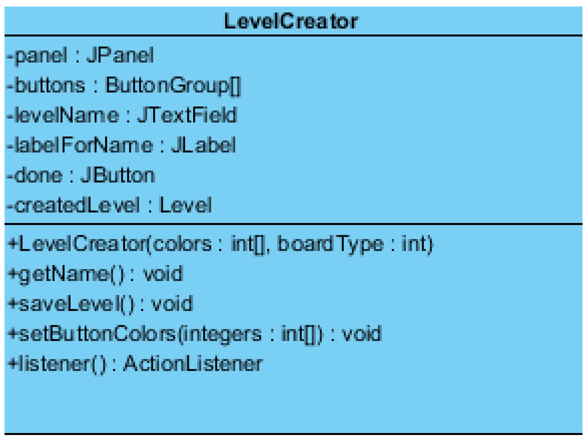


Figure 29 : LevelCreator Class Diagram

Attributes:

- JPanel panel : View of the level creator class
- ButtonGroup[] buttons : Button list which initialized by piece colours
- JTextField levelName : Text field which takes level name as input
- JLabel labelForName : label for the name
- JButton done : checks level validity button
- Level createdLevel : created level from the buttons with given name

Methods:

- LevelCreator(int[] colors, int boardType) : constructor for the level creator initialized by colors which come from piece creator
- void getName() : get level name
- void saveLevel() : save level into proper file system
- void setButtonColors(int[] integers) : initialize colors of the buttons and group the buttons.
- ActionListener listener() : button listener for the view action

4.4.15 PieceCreator Class

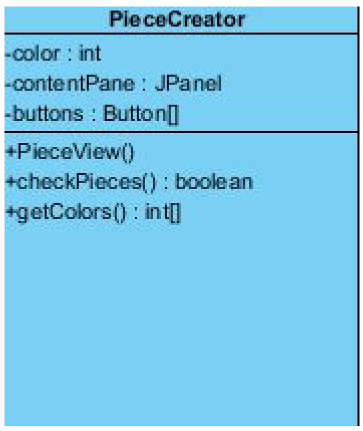


Figure 30 : MyNode Class Diagram

Attributes:

- int color : active color for painting
- JPanel contentPane : view of the piece creator
- Button[] buttons : grid of the game nodes which will be painted

Methods:

- PieceView() : constructor for the pieces
- boolean checkViews() : check all buttons painted and united
- []int getColors() : gets color of the all pieces

4.4.15 Main Class

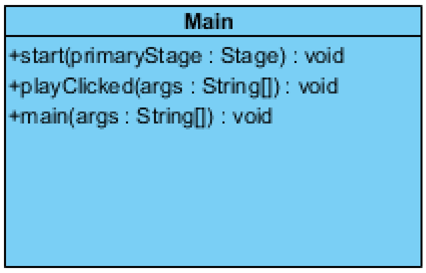

Main
+start(primaryStage : Stage) : void
+playClicked(args : String[]) : void
+main(args : String[]) : void

Figure 31 : MyNode Class Diagram

Methods:

- void start(Stage primaryStage) : his method is inherited from javaFX application which initialize the software
- void playClicked(String[] args) : calls the function which perform action when button clicked
- void main(String[] args) : main method for the application

5. Improvement Summary

- We revised and wrote againg our “Trade-offs” section.
- We removed “Criteria” section.
- We revised “Access Control and Security” section.
- We revised “Persistent Data Management” section.
- We added “Subsystem Decomposition” section.
- We added “Boundary Conditions” section.
- We added Class Diagrams in a visible format.
- We added a package diagram for “Subsystem Decomposition” section.
- We added other missing diagrams.

6. Glossary & References

[1] “Rotation(Mathematics).” *Wikipedia*, Wikimedia Foundation, 5 Nov. 2018, en.wikipedia.org/wiki/Software_design.
[2]“Oracle Documentation.”, 9 Dec. 2013, docs.oracle.com/javafx/2/api/overview-summary.html.