CS 319 - Object-Oriented Software Engineering

System Design Report

Katamino

<u>Group 1-G</u>

Mert Epsileli

Yusuf Samsum

Fırat Yıldız

Burak Korkmaz

Faruk Ege Hatırnaz

1. Introduction

1.1 Purpose of the System

The system of Katamino aims to entertain users while solving fun puzzles. The puzzles in the system is about geometrical perception and it will have an easy and familiar user-interface to ease the user interactions. Each level of the game has a solution which the user needs to find. Three different game modes, “Classic Mode”, “Challenge Mode”, and “Dynamic Mode” , are constituted for players to have varied experiences through our game. Players can check their results and compare their ranking in the game which makes our game even more competitive than the original game.

We intend to make a system easy enough to use and make it perform smoothly to enhance the user experience.

1.2 Design Goals

As explained in the analysis report, our design were constituted over basicality and user-friendliness. Non-functional requirements and design goals are explained in the following lines.

- **Functionality**: One of our design goals is the functionality. This goal is essentially about reaching up to the standards of the user and satisfying their needs with minimal complexity.

  We try to reach our functionality goal by implementing a menu where it displays all possible selections that user could make (i.e, including all functions for the user within the system) and making the gameplay functional. For instance, user will drag the puzzle pieces with the mouse and double click it to turn the piece. This is one of our implementations to maintain functionality of our design.

- **Usability**: Our design goals include usability as we want our game to be played by many. We design to hide away the unnecessary details from the user and give them a game that is tempting and easy to play. Maintaining a less cluttered design is key to our usability goal.

We can achieve this goal with implementing a simple, plain system design while maintaining the fun and exciting elements within the game and convince our target audience to play the game.

- **Reliability**: Our system has to work as clean as possible with minimal downtimes and exceptions, thus reliability is one of our design goals.

  We intend to keep our reliability high with handling exceptions in the correct way (i.e, no game crashing exceptions) and designing our boundary conditions as thoughtful as possible.

- **User-friendliness**: This part goes hand in hand with the ease of learning the game. We prioritise user-friendliness in our design because we want our users to be comfortable when using our system design while they play our game.

  We intend to achieve this user-friendliness goal with easy navigation of menus, giving freedom to user with customizable settings such as the volume control and making the game itself easy to play.

- **Efficiency**: Even though our game is not complex itself, our design goals still include efficiency in terms of the memory usage. Increased performance means increased responsiveness and this would make our game more easy to use.

  We might obtain this goal by creating less objects within the game and re-use our sources (e.g, maintenance of Stage and Scene objects regarding UI or using minimal number of external sources such as images or sounds) whenever we can.

- **Portability**: We utilize Java language in the development process. This helps us to use the “WORA” philosophy of Java. This is made possible with JVM as the code would compile into a standard bytecode which could run any device supporting JVM. This approach would help maintain our design goal regarding portability.

  We reach our goal of portability by using Java as our primary language in our project. Perhaps with a few adjustments the game could run in Android as well as the game codes would be completed and only remaining part left would be the adjustments made for the Android OS.

  Possible Trade-Offs:

- Functionality vs Usability: As a design principle, designer would have to make sacrifices from the usability when designer wants high functionality or vice versa. The system needs to balance itself between functionality and usability. The system has to be easy to use with its basic functionality but this functionality must be robust enough to satisfy the needs of the player.

- Understandability vs Functionality: Another compromise is between understandability and functionality. High functionality brings less understandability. To balance, our design is simple for understanding the game better while still protecting a fair bit of functionality. The User Interface does not have any unnecessary details to confuse users mind, only the options that user is able to choose from.

- Efficiency vs Portability: Java is a high level language and it utilizes its own virtual machine. In order to have portability, we have to make sacrifices in terms of efficiency. This issue needs to be addressed carefully as the choice would affect the software quality and system performance.

- Development Time vs Performance : In our project, we did not decide to work with C++. Because it is hard to determine the memory space, and the classes are not enough as Java. So we decided to use Java for the project. Classes in Java provide simplicity for User Interface, and hierarchy. This way, our development time would be rapid while making a compromise from the performance of the system.

1.3 Definitions, Acronyms and Abbreviations

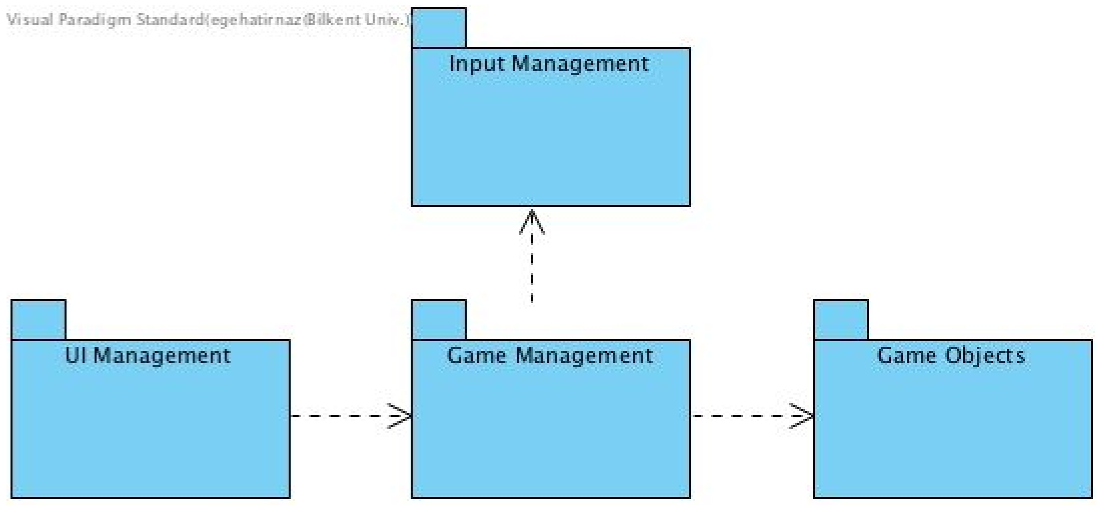
WORA: Write Once, Run Anywhere [1]
JVM: Java Virtual Machine [2]
JRE: Java Runtime Environment [3]
JDK: Java Development Kit [4]

2. High Level System Architecture

2.1 Subsystem Decomposition

We intend to decompose our system into smaller subsystems. The subsystems will be easier to modify and it will be more understandable and readable for backtracking or changes in the design.

There will be four subsystems connected to each other to separate the work done by each of those subsystems: UI Management, Game Management, Game Objects, Input Management



These subsystems will include their own related packages and classes. For instance, UI Management subsystem will include MainMenu class to form a connection between the end user and Game Management.

The detailed informations related to these subsystems will be handled at Section 3.

2.2 Hardware / Software Mapping

The game would require JDK version 8. JRE (Java Runtime Environment) needs to be executed to run our Java codes. Since that UI will be handled by JavaFX, having the version 8 JDK is necessary as minimum hardware requirement.

The user needs to have a Keyboard and Mouse connected to the device to play the game. User will type their usernames with the keyboard and play the game with the connected mouse.

To maintain the leaderboards, internet connection is be needed to make communications with the database.

2.3 Persistent Data Management

Game data will be stored in the hard drive. We intend to maintain an MySQL database for the Leaderboards. We will store the time and names of our players there to make a leaderboard. Our sound and image files will be stored unencrypted as it would have no effect on the security.

2.4 Access Control and Security

Security measures regarding Access Control has to be taken as the leaderboard information would come from an external database. However, we will not be storing any user credentials that would be sensitive data. Only data we obtain is the username and the time score made by that user. We will not be having a password input or login area and therefore we will not be storing any passwords. For this reason, security does not play an important role in our design.

2.5 Boundary Conditions

The game will be initialized with default values decided by the developers.

The puzzle ends when the player fills all of the empty areas on the board with blocks and returns to the level list / main menu.

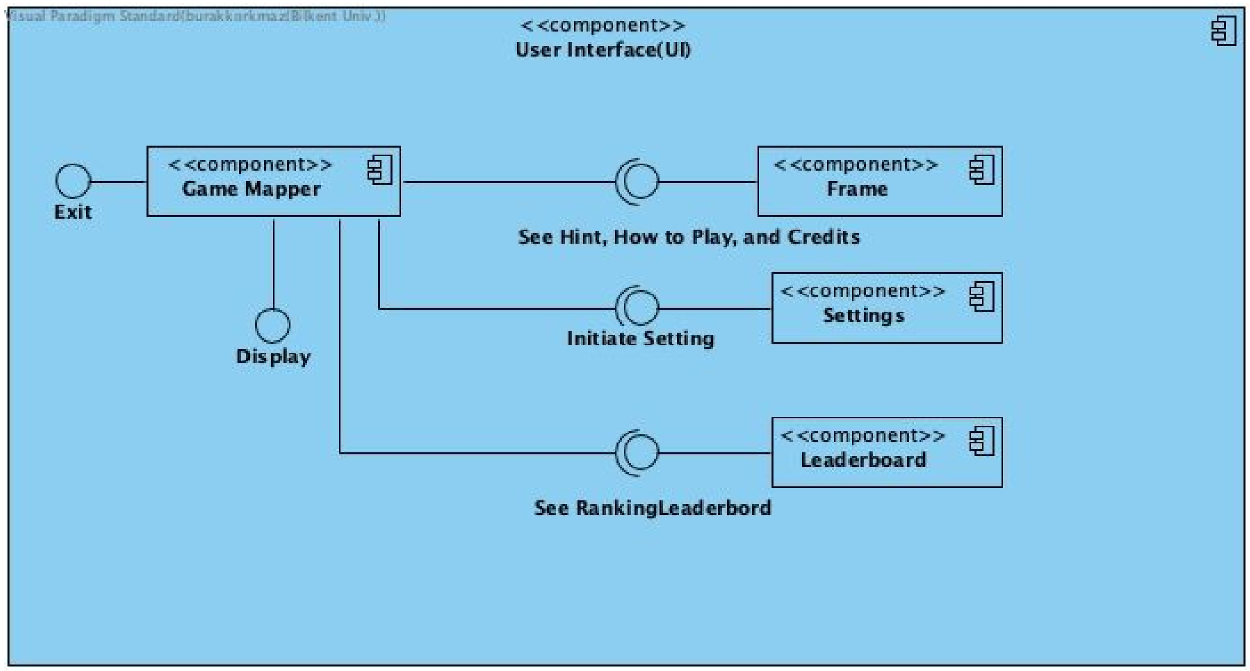
If the puzzle is completed, only then, the username and time score will be uploaded to the leaderboards and standings will be updated.

User can close the game by clicking on the exit in menus which should exit the game normally. (Like in exit code 0)

If an exception happens and the game crashes, the data will not be saved or updated. Game will discard any changes prior to the crash.

3.Subsystem Services

3.1 User Interface(UI) Management Subsystem



User Interface Management Subsystem provides the user interface of the Katamino. The components are listed as belows

- Frame Component
- Menu Component
- Game Mapper Component
- Settings Component
- Leaderboard Component

Frame Component: The user interface for the informations of the game are provided by the frame component. “Hints”, “How to Play”, and “Credits” are information frames and provides an interface by **Frame Component.** It is invoked by **Menu Component**.

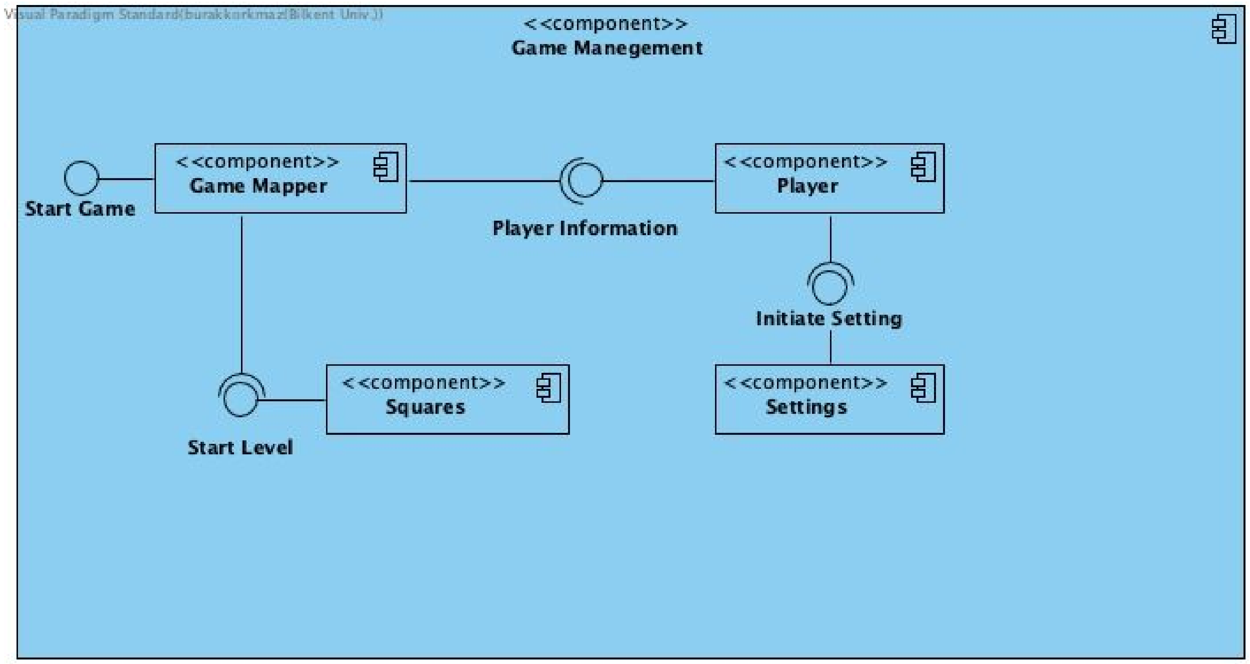
Menu Component: At the beginning of the game **Menu Component** provides a menu for users. User can exit the menu, starts the game with different game modes such as, “Classic Mode”, “Challenge Mode”, and “Dynamic Mode”. **Menu Component** invokes other components by different classes for menus such as “See Hints, How To Play, and Credits”, “Initiate Settings”, and “See Leaderboard” for **Frame Component, Settings Component**, and **Leaderboard Component** accordingly. is selected and kept in the Menu Component. Menu component is the main junction which invokes other components apart from **Game Mapper**. For example, when a mode is selected, Game Mapper component is triggered and send all UI data to Frame Component. This case is also valid for Settings, Credits, How To Play and Leaderboard components.

Game Mapper Component: Provides the interface of the gameplay structures such as blocks, boards, sticks, and time. It displays the main flow of the game. It records the changes which applied on gameplay from **Game Management Subsytem**. Moreover **Game Mapper Component** invokes Leaderboard and Menu components.

Settings Component: Adjusting volume and language selections are located in the settings component. **Settings Component** keeps user settings, and updates them while changing occurs. It is invoked by **Menu Component**.

Leaderboard Component: Leaderboard Component provides a user interface for user’s ranking in the leaderboard. It gets the leaderboard information from the database with the game. Leaderboards of the different game modes can be provided differently with related themes for game modes. It is invoked by Menu Component, and Game Mapper.

3.2 Game Management Subsystem



Game Management Subsystem is the system responsible for composing the game in terms of UI, I/O and binding the objects to the game. This subsystem is composed of 4 components:

- Game Mapper
- Player
- Settings
- Squares

Game Mapper Management:
This component creates an instance of the game and takes the input from the user. **Game Mapper** starts the game, initializes the puzzles and proceeds onto the next level when the solution has been found. Moreover, it handles the files such as background images or soundtracks. **Game Mapper** controls the system of the game which acts as an Admin.

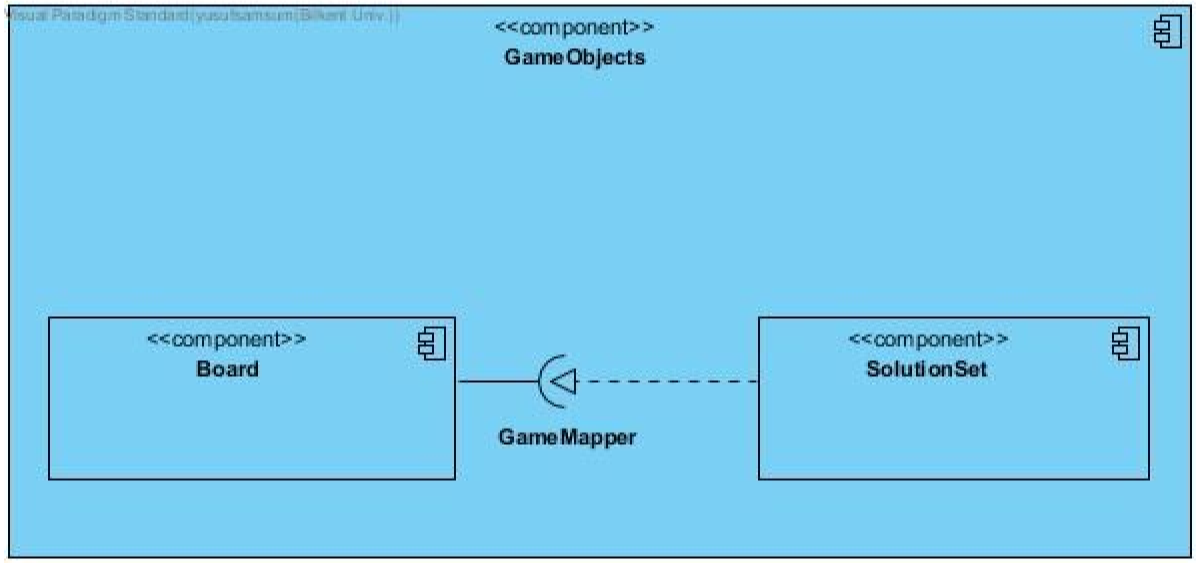
Squares Management:
This component is responsible for creating the objects and placing those components in the puzzle level. **Squares** component handles the model of the game and its component. It controls the relationship between the

gameplay and levels. It updates the **Game Mapper** with the changes in gameplay.

Player Management:
This component receives the user information and takes the information into **Game Mapper** to keep the record of the time in gameplay.

Settings Management:
This component can use and sets the settings of the game and player. This component is called from **Player.**.

3.3 Game Objects Subsystem

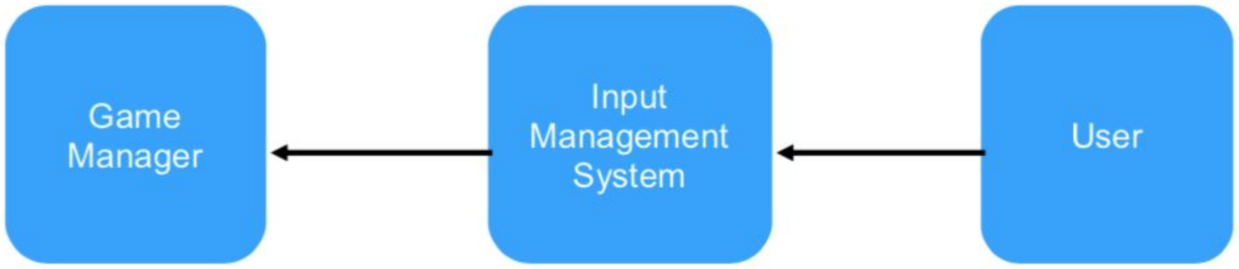


Game Objects subsystem keeps all game objects. Solution set realization is done by the GameMapper object. Solution set has the all blocks with appropriate level and sets. Board has the board width and height with respect to current level and mode.
All of these components are realized in GameMapper class and they create GameObjects component.

Solution set is fed by vertices and adjacency list in the Vertex class. These solution set has the Block IDs which keeps the ID of each block. A block consists of one vertex and its adjacency list. Adjacency list represents the continuation of the block's shape. Each block shape, which means a vertex, has an ID, which means block ID. Solution set is a 2D block ID array and row of this array each solution set. This sets are kept in the database and send to the GameMapper when they call.

Board is a part of GameMapper and it is determined by current level and current game mode. Each game mode and current represents a rectangle and that is because we need only the coordinates for representing the board.

3.4 Input Management Subsystem



As it is known, the game will be managed according to inputs given by the player. These inputs will be given by the mouse. Before entering the game, players should also enter their nickname in order to be used in the ranking and check whether it is a suitable one which means it is used by another user. In the Input Management Subsystem, all the inputs will be organized and served to the classes and methods needs. **void mouseClicked(MouseEvent e), void mousePressed(MouseEvent e), void mouseReleased(MouseEvent e), void mouseEntered(MouseEvent e), void mouseExited(MouseEvent e)** and **void handleEvent(Event evt)** methods which explained in below will get their inputs after some controls in the Input Management System. Also the validity of the nicknames given by players will be checked in the Input Management System. Therefore, the constructor methods of player class will also get the student nicknames from Input Management System after checking whether it is suitable.

4. Low-Level Design

4.1 Object Design Trade-offs

4.1.1 Strategy Pattern

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from client that use it. We chose that design pattern for implementing board and blocks since it provides us flexibility we need. As you can observe in our analysis report, we use Square class either for creating blocks and the board's pieces. That is why we need to use Rectangle class which is a wrapper class for drawing rectangle in Square class. However, instead of using "is a" association, we preferred "has a" association between our classes and objects. We encapsulate our Square class with Vertex class and GameMapper class. That gives us the flexibility of using Square class whatever way we want and we can change the board's and block's squares immediately at run time by our strategy classes. Although, this pattern requires more memory and less performance at run time, the flexibility that is given to us has much more importance for us. Also, since our program uses data structures in efficient way, the design pattern does not affect in terms of memory and performance. That is why we decided to use Strategy class for our game play mechanism.

4.1.2 Factory Design Pattern

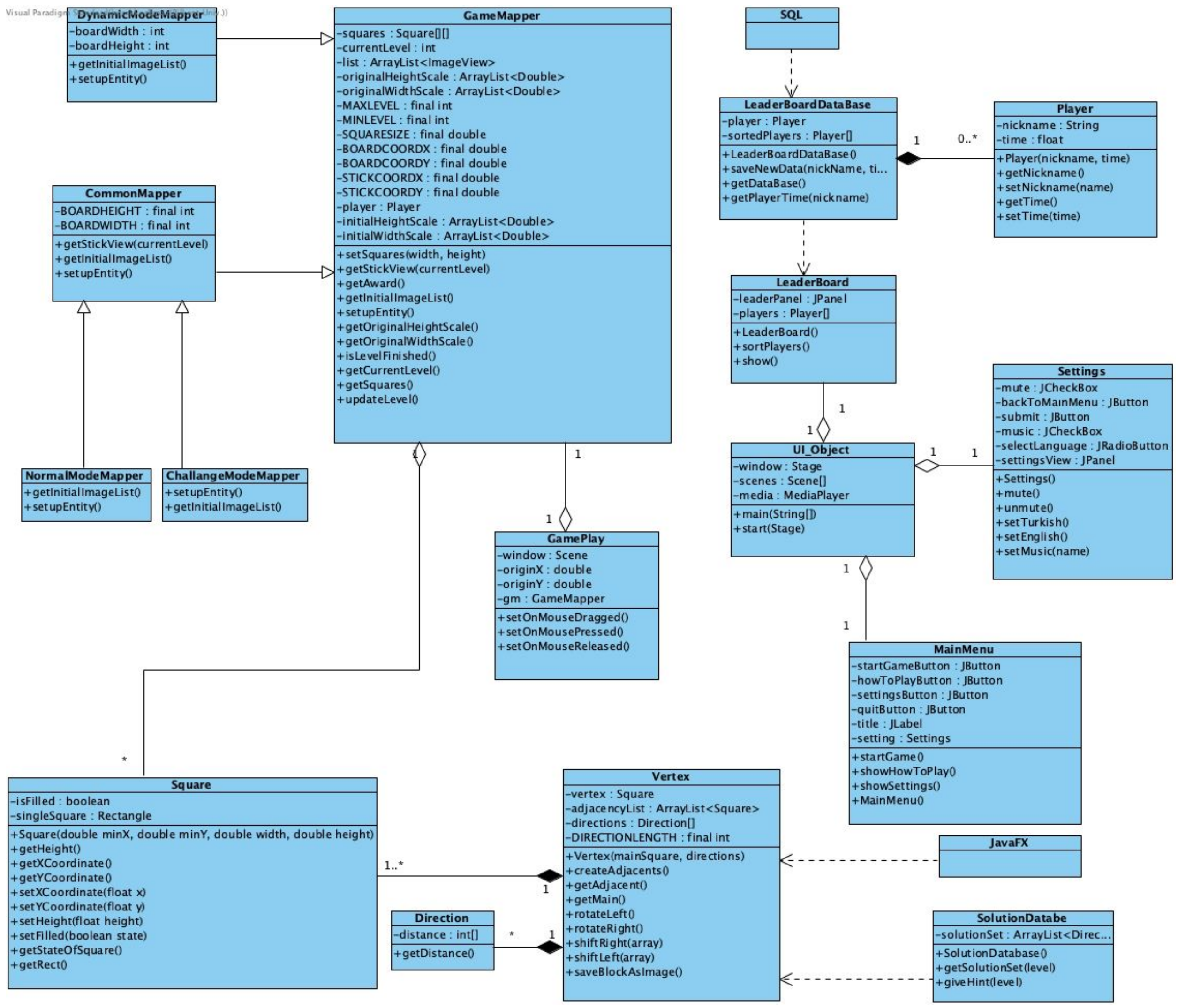
We used that approach to design our game modes. In this game, we have three different modes as follows: normal mode, challenge mode and dynamic mode. All of these modes are using almost same methods but not same board size and solution set. Therefore, we implement factory classes such as CommonMapper and DynamicModeMapper. This provides us flexibility for creating game objects. GameMapper have no idea about which mode is executing and which entities executed. Although, makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions, this pattern provides flexibility and simplification for us.

4.1.3 Facade Design Pattern

Facade provides a unified interface to a set of interfaces in a subsystem. Therefore, we implement GamePlay class for building a wall against GameMapper class. We use that class to simplify the game qualifications and call the caller methods. This pattern cause to diminish in performance rate but it simplifies the usage of the GameMapper class.

4.2 Final Object Design

Final object design is created with respect to design pattern that mentioned. You can clearly see that mapper, modes and block classes are designed in proper way.



4.3 Packages

In the implementation there will be two different types of packages. One of them is Developer Packages and the other one is External Packages.

4.3.1 Packages Introduced by Developer

4.3.1.1 Menu Package

This package will include the classes which are responsible for menu interface and menu methods.

4.3.1.2 Settings Package

This package will include the classes which are responsible for settings menu and the methods of this menu.

4.3.1.3 GameMap Package

This package will include the classes which are responsible for gameplay screen and some methods to demonstrate this screen.

4.3.1.4 Database Package

This package will include the classes which are responsible for players and adding or getting player information from solution and leaderboard database.

4.3.1.5 Controller Package

This package will include the classes which are responsible for controlling game actions and user inputs.

4.3.2 External Library Packages

4.3.2.1 java.sql.DriverManager

This package will provide classes which are responsible for providing connection between database and java.

4.3.2.2 java.sql.PreparedStatement

This package will provide classes which are responsible for sending orders to database from java.

4.3.2.3 java.sql.SQLException

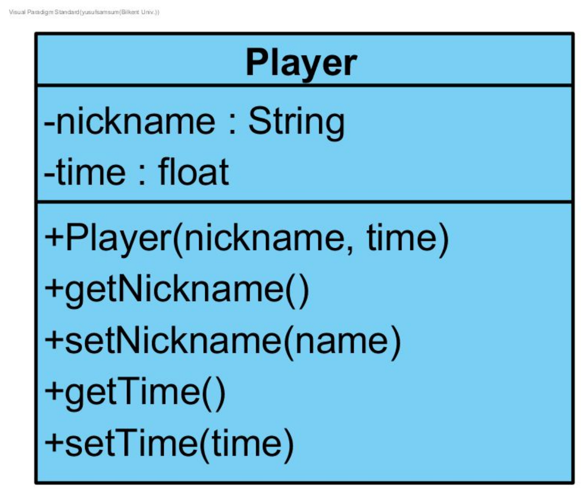
This package will provide classes which are responsible for showing error which are related to database.

4.3.2.4 java.sql.ResultSet
This package will provide classes which are responsible for getting data from database.

4.3.2.5 Java FX
This package will provide classes that are useful for implementing UI design.

4.4 Class Interfaces

4.4.1. Player Class



Attributes:

- **String nickname:** The nickname which is chosen by a user before starting game. It must be different than old nicknames.

- **float time**: The finishing time of the user

Constructor:

- **Player (String nickname, time):** Creates a player object with given nickname and time.

Methods:

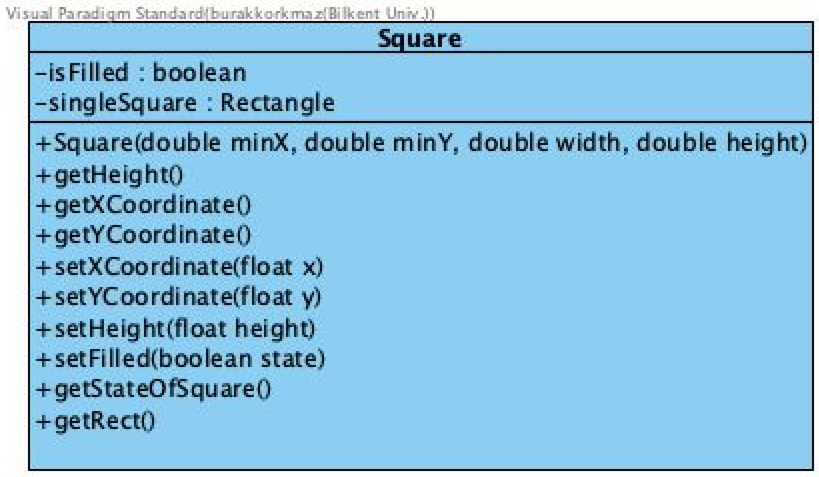
- **String getNickname():** Get the nickname of a player object.

- **void setNickname(String nickname):** Set the player object's nickname as given nickname.

- **float getTime():** Get the time of a player object.

- **void setTime(float time):** Set the player object's time as given time.

4.4.2. Square Class



Attributes:

- **boolean isFilled:** A boolean which shows that whether the square is filled or not.

- **Rectangle singleSquare:** A Rectangle object which is used in order to draw game board in GameMapper class.

Constructor:

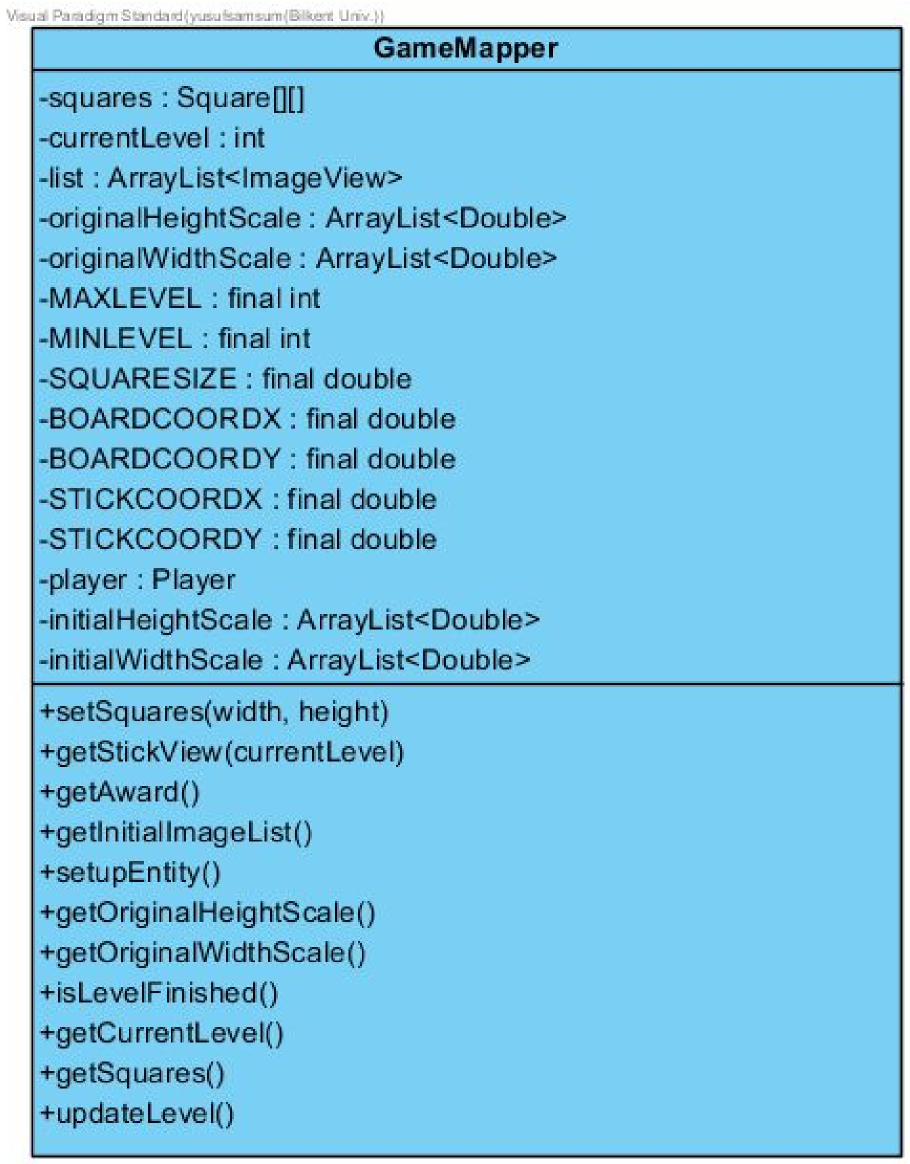
- **Square (double minX, double minY, double width, double height):** Creates a singleSquare with given properties and the isFilled attribute is false.

Methods:

- **getXCoordinate():** Get the x coordinate of the singleSquare.

- **getYCoordinate():** Get the Y coordinate of the singleSquare.
- **setXCoordinate(float x):** Set the x coordinate of the singleSquare to given x.
- **setYCoordinate(float y):** Set the y coordinate of the singleSquare to given y.
- **setHeight(float height):** Set the height of the singleSquare to given height.
- **setFilled(boolean state):** Set the isFilled attribute to given state.
- **getStateOfSquares():** Return 'isFilled' attribute of the class.'
- **getRect():** Get the singleSquare attribute.

4.4.3. GameMapper Abstract Class


Visual Paradigm Standard(yusufsamsum(Bilkent Univ.))
GameMapper
-squares : Square[][]
-currentLevel : int
-list : ArrayList<ImageView>
-originalHeightScale : ArrayList<Double>
-originalWidthScale : ArrayList<Double>
-MAXLEVEL : final int
-MINLEVEL : final int
-SQUARESIZE : final double
-BOARDCOORDX : final double
-BOARDCOORDY : final double
-STICKCOORDX : final double
-STICKCOORDY : final double
-player : Player
-initialHeightScale : ArrayList<Double>
-initialWidthScale : ArrayList<Double>
+setSquares(width, height)
+getStickView(currentLevel)
+getAward()
+getInitialImageList()
+setupEntity()
+getOriginalHeightScale()
+getOriginalWidthScale()
+isLevelFinished()
+getCurrentLevel()
+getSquares()
+updateLevel()

Attributes:

- **Square[][] squares:** Two dimensional array of Square objects. The game board is drawn by using these squares.

- **int currentLevel:** The level which will be played by the user. After passing a level, current level increases one.

- **ArrayList<ImageView> list:** List of the all block which are ImageView. After that the elements of this list are added to a Group object in order to be displayed in the screen.

- **ArrayList<Double> originalHeightScale:** Original Height of the blocks after the blocks is selected with mouse, which kept in double array.

- **ArrayList<Double> originalWidthScale:** Original Width of the blocks after the blocks is selected with mouse, which kept in double array.

- **final int MAXLEVEL:** Number of the last level of the game.

- **final int MINLEVEL:** Number of the first level of the game.

- **final int SQUARESIZE:** The size of a side of a board square.

- **final int BOARDCOORDX:** The default X coordinate of the board.

- **final int BOARDCOORDY:** The default Y coordinate of the board.

- **final int STICKCOORDX:** The default X coordinate of the stick.

- **final int STICKCOORDX:** The default Y coordinate of top the stick.

- **Player player:** A player object which will play the game.

- **ArrayList<Double> initialHeightScale:** Initial Height of the blocks before the blocks is selected with mouse, which kept in double array.

- **ArrayList<Double> initialWidthScale:** Initial Width of the blocks before the blocks is selected with mouse, which kept in double array.

Methods:

- **void setSquares(int width, int height):** Set the coordinates of the all squares in ‘squares’ array according to given width, height.

- **ImageView getStickView(int currentLevel):** Bodiless abstract method. It will be defined in child classes.

- **ImageView getAward():** Returns the award ImageView after end of the level.

- **ArrayList<ImageView> getInitialImageList():** Bodiless abstract method. It will be defined in child classes.

- **void setupEntity():** Bodiless abstract method. It will be defined in child classes.

- **ArrayList<Double> getOriginalHeightScale():** Returns the 'OriginaleHeightScale' attribute.

- **ArrayList<Double> getOriginalWidthScale():** Returns the 'OriginaleWidthScale' attribute.

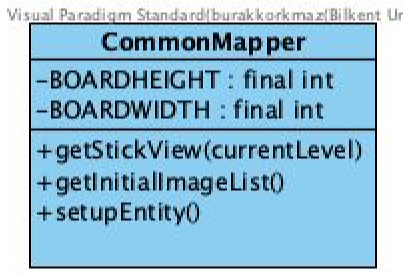
- **boolean isLevelFinished():** Controls the squares of board, whether they are filled, or not. After that, if they are filled, it returns true. This mean the level finished. If they are not filled, it returns false. This means the level did not finish.

- **int getCurrentLevel():** Returns the current level.

- **Square[][] getSquares:** Returns the 'squares' attribute which is two dimensional array of Square class in order to generate board.

- **void updateLevel():** Increases the current level, if the level finished.

4.4.5. CommonMapper Abstract Class



Attributes:

- **final int BOARHEIGHTX:** The height of the game board. It will be 5 * SQUARESIZE in Normal and Challange game mods.

- **final int BOARDWIDTHY:** The height of the game board. It will be 13 * SQUARESIZE in Normal and Challange game mods.

Methods:

- **ImageView getStickView(int currentLevel):** Returns the stick ImageView in appropriate location according to given currentLevel.

- **ArrayList<ImageView> getInitialImageList():** Bodiless abstract method. It will be defined in child classes.

- **void setupEntity():** Bodiless abstract method. It will be defined in child classes.

4.4.6. NormalModeMapper Class

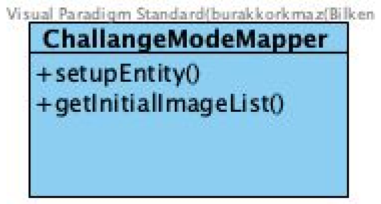
Visual Paradigm Standard(burakkorkmaz(Bilk
NormalModeMapper
+getInitialImageList()
+setupEntity()

Methods:

- **ArrayList<ImageView> getInitialImageList():** Returns the ‘list’ attribute which storages the blocks as ImageView.

- **void setupEntity():** The method sets coordinates of blocks of the gameplay according to 'BOARDCOORDX', 'BOARDCOORDY' and 'SQUARESIZE'.

4.4.7. ChallangeModeMapper Class

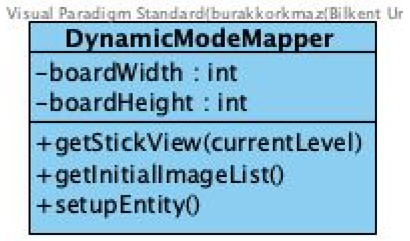


Methods:

- **ArrayList<ImageView> getInitialImageList():** Returns the ‘list’ attribute which storages the blocks as ImageView.

- **void setupEntity():** The method sets coordinates of blocks of the gameplay according to 'BOARDCOORDX', 'BOARDCOORDY' and 'SQUARESIZE'.

4.4.8. DynamicModeMapper Class



Attributes:

- **int boardWidth:** The width of the game board in dynamic mode. It will change in every level.

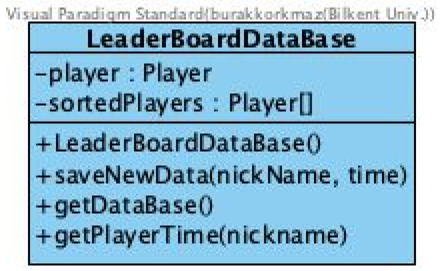
- **int boardHeight:** The height of the game board in dynamic mode. It will change in every level.

Methods:

- **ArrayList<ImageView> getInitialImageList():** Returns the ‘list’ attribute which storages the blocks as ImageView.

- **void setupEntity():** The method sets coordinates of blocks of the gameplay according to 'BOARDCOORDX', 'BOARDCOORDY' and 'SQUARESIZE'.

4.4.9. LeaderBoardDataBase Class


Visual Paradigm Standard(burakkorkmaz(Bilkent Univ.))
LeaderBoardDataBase
-player : Player
-sortedPlayers : Player[]
+LeaderBoardDataBase()
+saveNewData(nickName, time)
+getDataBase()
+getPlayerTime(nickname)

Attributes:

- **Player player:** A player object to be saved to database.

- **Player[] sortedPlayers:** Create an array of players in the database. The array is sorted with respect to time of players.

Constructor:

- **LeaderBoardDataBase()**: Creates an object of the class in order to be used in other classes. (Default constructor.)

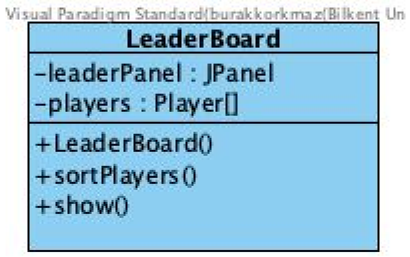
Methods:

- **void saveNewData(nickname, time):** Saves the player with given nickname and finishing time to database.

- **Player[] getDataBase():** Takes every nicknames and the time of these nicknames from the database. After that creates a player object for every single nickname and time. Lastly, creates an array of these players.

- **getPlayerTime(nickname):** Get the time of the player with given nickname.

4.4.10. LeaderBoard Class



Attributes:

- **JPanel leaderPanel:** A panel which includes leaderboard. There will be nicknames and time in this leaderboard.

- **Player[] players:** All players which are in LeaderBoardDataBase. The nicknames and time of these players will be sorted to create a leaderboard.

Constructor:

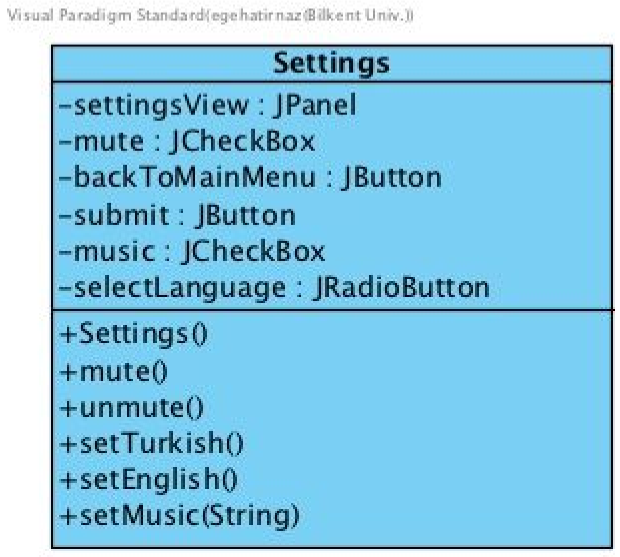
- **LeaderBoard():** Creates an empty leaderboard. (Default constructor.)

Methods:

- **void sortPlayers():** Sorts the players array of LeaderBoard Class with respect to time players. The player with shortest time will be the 0. element of the players array.

- **void show():** Transfers the information of sorted array to leaderPanel. After that creates a real leaderboard and shows it.

4.4.11. Settings Class



Attributes:

- **JCheckBox mute:** Creates a JCheckBox. If users check this JCheckBox, the game will be mute.

- **JButton backToMainMenu:** Creates a “backMenu" button which is used to return the main menu, if users click on it.

- **JButton submit:** Creates a “submit” button. If the user clicks on it, the settings are submitted.

- **JCheckBox music:** Creates a JCheckBox. Users will be able choose the game music from given musics by using this JCheckBox.

- **JRadioButton selectLanguage:** Creates a radio button. There will be two option which are “Turkish” and “English”. Users will choose one as game language.

- **JPanel settingsView:** All these JCheckBoxes, JButtons and JRadioButton will be shown in this JPanel.

Constructor:

- **Settings():** Creates a settings object. (Default constructor.)

Methods:

- **void mute():** If users check ‘mute JCheckBox’, this method makes the game mute.

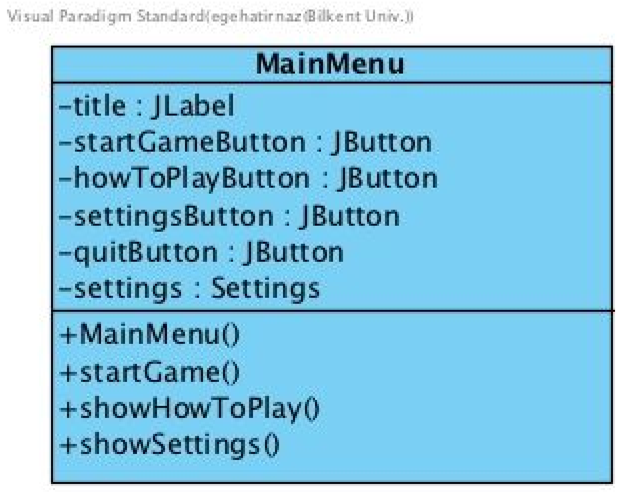
- **void unmute():** If users uncheck ‘mute JCheckBox’, this method makes the game voiced.

- **void setTurkish():** If users chooses ‘Turkish JRadioButton’, this method makes the language of game Turkish. It includes the array of words in Turkish and reaches the fxml files to change their context.

- **void setEnglish():** If users chooses ‘English JRadioButton’, this method makes the language of game English. It reaches the fxml files according to arrays inside which includes word array.

- **void setMusic(String name):** Changes the music according to chosen music on ‘music JCheckBox’. “name” parameter is the name of the music.

4.4.12. MainMenu Class



Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
MainMenu
-title : JLabel
-startGameButton : JButton
-howToPlayButton : JButton
-settingsButton : JButton
-quitButton : JButton
-settings : Settings
+MainMenu()
+startGame()
+showHowToPlay()
+showSettings()

Attributes:

- **JButton startGameButton:** Creates a “startGame” button. If users click on it, users go to ‘choosing Nickname’ screen.

- **JButton howtoPlayButton:** Creates a “howToPlay” button. If the user clicks on it, ‘How to Play?’ screen is opened.

- **JButton settingsButton:** Creates a “settings” button. If the user clicks on it, ‘settings’ screen is opened.

- **JButton quitButton:** Creates a “quit” button. If the user clicks on it, the user exits from the game.

- **JLabel title:** Creates a JLabel for “Main Menu” title.

- **Settings settings:** A Settings object which is used to show settings panel.

Constructor:

- **MainMenu():** Creates a MainMenu object. (Default constructor.)

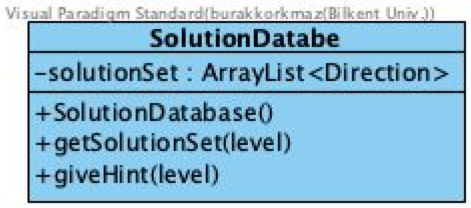
Methods:

- **void startGame():** Starts the game.

- **void showHowToPlay():** Shows a panel which includes how to play information.

- **void showSettings():** Shows settings panel which are created in Settings class.

4.4.13. SolutionDatabase Class


Visual Paradigm Standard(burakkorkmaz(Bilkent Univ.))
SolutionDatabe
-solutionSet : ArrayList<Direction>
+SolutionDatabase()
+getSolutionSet(level)
+giveHint(level)

Attributes:

- **ArrayList<Direction> solutionSet:** The array of Direction object. These Directions are used to draw blocks. Each Direction is used to draw a block. Then these blocks are saved as Image and displayed as ImageView.

Constructor:

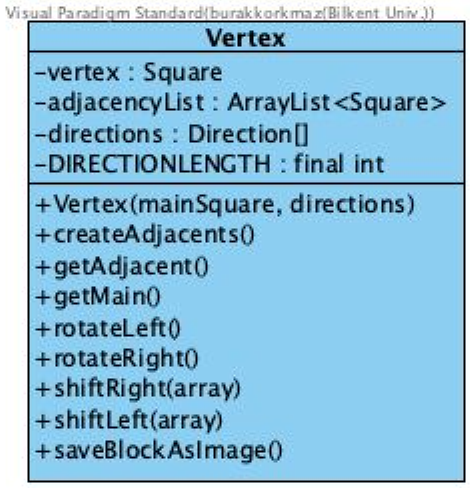
- **SolutionDatabase():** Creates a SolutionDatabase object. (Default constructor.)

Methods:

- **ArrayList<Direction> getSolutionSet(int level):** Get an Direction arraylist according to given level. These directions are used to draw the blocks of given level.

- **ImageView getHint(level):** Returns a block's ImageView is in correct location according to given level.

4.4.14. Vertex Class


Visual Paradigm Standard(burakkorkmaz(Bilkent Univ.))
Vertex
-vertex : Square
-adjacencyList : ArrayList<Square>
-directions : Direction[]
-DIRECTIONLENGTH : final int
+Vertex(mainSquare, directions)
+createAdjacents()
+getAdjacent()
+getMain()
+rotateLeft()
+rotateRight()
+shiftRight(array)
+shiftLeft(array)
+saveBlockAsImage()

Attributes:

- **Square vertex:** Main square of the blocks. Other squares are drawn with respect to main square.

- **ArrayList<Square> adjacencyList:** The other squares which create a block.

- **Direction[] directions:** The directions of other squares according to main square. These directions are used to shape the block.

- **final int DIRECTIONLENGTH:** The size of the ‘directions’. It is 4 in this game because there are 5 squares in a block totally. One of them main square which is ‘vertex’. The other 4 blocks are drawn according to ‘directions’, so there are 4 directions.

Constructor:

- **Vertex(Square mainSquare, Direction[] directions):** Assigns ‘vertex’ to mainSquare and ‘directions’ to ‘directions’.

Methods

- **void createAdjacents():** Creates the ‘adjacencyList’ according to ‘vertex’ and ‘directions’. This method determines the coordinates of other squares.

- **ArrayList<Square> getAdjacents():** Get the ‘adjacencyList’ attribute.

- **Square getMain():** Get the ‘vertex’ attribute.

- **void rotateLeft():** Rotate the each square in ‘adjacencyList’ 90 degree left about the ‘vertex’. For example, an element of adjacencyList is in the bottom of ‘vertex’. When it is rotated, it becomes in the right of ‘vertex’.

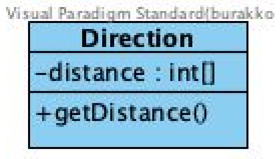
- **void rotateRight():** Rotate the each square in ‘adjacencyList’ 90 degree left about the ‘vertex’. For example, an element of adjacencyList is at the bottom of ‘vertex’. When it is rotated, it becomes at the left of ‘vertex’.

- **void shiftRight(int[] array):** This method shifts the given array to the right. For example, if given array is [1,2,3], shifted array is [3,1,2]. This method is used in rotateRight() method.

- **void shiftLeft(int[] array):** This method shifts the given array to the right. For example, if given array is [1,2,3], shifted array is [2,3,1]. This method is used in rotateLeft() method.

- **void saveBlockAsImage():** This method saves the block which is drawn by using ‘vertex’ and ‘adjacencyList’ as an Image to the computer. This Image is called in gameplay screen as ImageView.

4.4.15. Direction Class



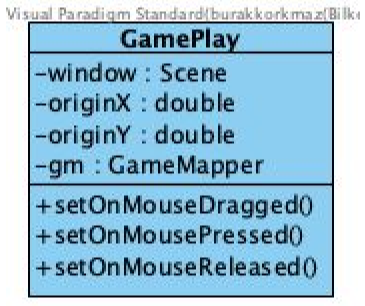
Attributes:

- **int[] distance:** Distance of a square from the main square. It is used in Vertex class. The first element of this array shows the distance rightward. Second element shows the distance downward. Third element shows the distance leftward. Fourth element shows the distance upward. For example, when the array is [1,1,0,0], the square is at the one unit right and one unit down of the main square. It means, the square is at the right down cross of main square.

Methods:

- **int[] getDistance():** Get the ‘distance’ attribute.

4.4.16. GamePlay Class



Attributes:

- **Scene window:** The 'GamePlay Scene' which is called in UI_Object Class to display the gameplay screen.

- **double originX:** The X coordinate of the mouse cursor.

- **double originY:** The Y coordinate of the mouse cursor.

- **GameMapper gm:** GameMapper object will provide all entities of the game objects that are blocks and board.

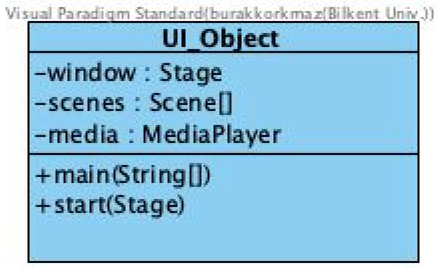
Methods:

- **setOnMouseDragged():** The action listener methods of JavaFX library which is used in the gameplay to execute specified operations when mouse dragged.

- **setOnMousePressed():** The action listener methods of JavaFX library which is used in the gameplay to execute specified operations when mouse pressed.

- **setOnMouseReleased():** The action listener methods of JavaFX library which is used in the gameplay to execute specified operations when mouse released.

4.4.17. UI_Object



Attributes:

- **Stage window:** The start frame of the game which includes "Start", "How To Play", "Credits", "Settings" and "Quit Game" buttons.
- **Scene[] scenes:** The array which includes changing scenes of the game which are 'Nickname Scene', 'ChooseMode Scene', 'Settings Scene', 'How To Play Scene', 'GamePlay Scene'.
- **MediaPlayer media:** The content which includes the game music.

Methods:

- **start(Stage):** The method calls all the scenes inside the scenes array to run modes and scenes of the game.

- **main(String[]):** The main method drive all the events of the game.

5. Improvement Summary

There were useless classes in class diagram. They were deleted. In addition, there were useless attributes and methods in some class, they were deleted. Lastly, some new methods and classes were added to the diagram. These new methods and classes were explained. Some of them were abstract classes, so some child classes were added to these abstract classes.

Design patterns were discussed and changed more appropriate ones. Level logic and game modes are added to class diagram. Instead of using bridge and command patterns we decided to use factory and strategy patterns. Game modes added by factory patterns. Square-GameMapper and Square-Vertex relations created by using strategy pattern. Finally, game play class created by the principle of facade design pattern.

In our subsystems we made some changes. We have changed our Game Management subsystem structure so that it enabled us to come up with a more condensed system design: Game Management subsystem now includes Game Mapper, Player, Settings and Square classes, which simplify the implementation. In the old design we had 'Input Management' component inside the Game Management subsystem. We have avoided to include it within Game Management subsystem in our second design iteration as it was already implemented in another subsystem. In UserInterface (UI) Management we changed Menu component with the Game Mapper component which is now controls Frame, Settings, and Leadeboard. It also displays the gameplay.

In our initial design, we had included JOGL in our External Library Packages. In our second iteration however, we have made a decision to work with JavaFX instead of JOGL for our visual components.

Glossary & References

[1] D. S. Ferru and M. Atzori, “Write-Once Run-Anywhere Custom SPARQL Functions,” 2016 IEEE Tenth International Conference on Semantic Computing (ICSC), 2016.

[2] Java Virtual Machine (Bill Venners, Inside the Java Virtual Machine Chapter 5).

[3] Techterms.com. (2018). JRE (Java Runtime Environment) Definition. [online] Available at: https://techterms.com/definition/jre [Accessed Nov. 12, 2018].

[4] En.wikipedia.org. (2018). Java Development Kit. [online] Available at: https://en.wikipedia.org/wiki/Java_Development_Kit [Accessed Nov. 12, 2018].

[5] Rodgers Cadenhead "Creating Animation with Java" [online] Available at: http://www.informit.com/articles/article.aspx?p=30419 [Accessed Dec. 3, 2018].

[6]Scot W. Ambler, [online] Available at: https://www.ogis-ri.co.jp/otc/swec/process/am-res/am/artifacts/deploymentDiagram.html [Accessed: Nov. 6, 2018].