CS 319 - Object-Oriented Software Engineering

System Design Report

Katamino

Group 1-G

Mert Epsileli

Yusuf Samsum

Fırat Yıldız

Burak Korkmaz

Faruk Ege Hatırnaz

Table of Contents

1. Introduction 4
   1.1 Purpose of the System 4
   1.2 Design Goals 4
   1.3 Definitions, Acronyms and Abbreviations 5
2. High Level System Architecture 6
   2.1 Subsystem Decomposition 6
   2.2 Hardware / Software Mapping 7
   2.3 Persistent Data Management 7
   2.4 Access Control and Security 7
   2.5 Boundary Conditions 7
3.Subsystem Services 8
   3.1 User Interface(UI) Management Subsystem 8
   3.2 Game Management Subsystem 10
   3.3 Game Objects Subsystem 11
   3.4 Input Management Subsystem 12
4. Low-Level Design 13
   4.1 Object Design Trade-offs 13
      4.1.1 Bridge Pattern 13
      4.1.2 Command Pattern 13
      4.1.3 Façade Pattern 14
   4.2 Final Object Design 14
   4.3 Packages 15
      4.3.1 Packages Introduced by Developer 15
      4.3.2 External Library Packages 17
   4.4 Class Interfaces 18
      4.4.1. Player Class 18
      4.4.2. GameAdmin Class 19
      4.4.3. LeaderBoardDataBase Class 21
      4.4.4. LeaderBoard Class 22
      4.4.5. Settings Class 23
      4.4.6. MainMenu Class 25

4.4.7. ControlManager Class ... 26
4.4.8. SolutionDatabase Class ... 29
4.4.9. Vertex Class ... 30
4.4.10. FrameManager Class ... 32
4.4.11. GameMapper Class ... 33
4.4.12. EventListener Class ... 34
4.4.13. MouseListener Class ... 34
5.0 Glossary & References ... 35

1. Introduction

1.1 Purpose of the System

Katamino is a children game which improves the visual memory.
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 was constituted over basicality and user-friendliness. Non-functional requirements and design goals are explained in the following lines.

- **Functionality**: One of our goals is the functionality. This goal is essentially about reaching up to the standards of the user and satisfying their needs.
- **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.
- **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 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 prioritize user-friendliness in our design because we want our users to be comfortable while they play our

game. We intend to provide this comfort 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 easier to use.
- **Portability**: We utilize Java language in the development process. This helps us to use the "Write Once, Run Anywhere" 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.

Possible Trade-Offs:

- Functionality vs Usability: 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: Our design is simple for understanding the game better. The User Interface does not have any unnecessary details to confuse users mind. It is very simple to play, and easy to learn

- 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.

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]

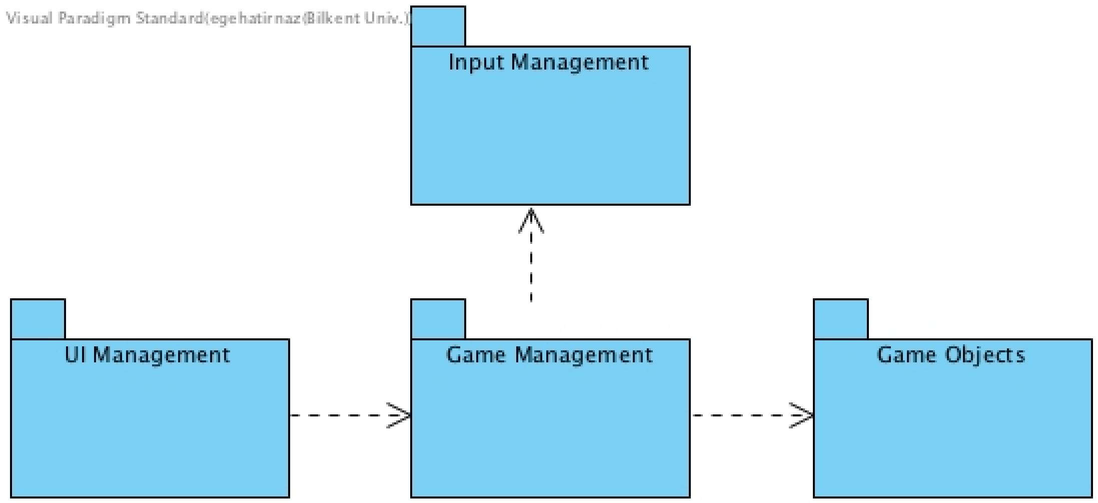
JOGL: Java Open GL

2. High Level System Architecture

2.1 Subsystem Decomposition

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

There will be 4 subsystems connected to each other: 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 information related to these subsystems will be handled at Section 3, respectively.

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.

The user needs to have a Mouse connected to the device to play the game.

The hardware requirements demand that the hardware should be able to run Java 2D Graphics Library with Hardware Acceleration.

To maintain the leaderboards, internet connection might be needed.

2.3 Persistent Data Management

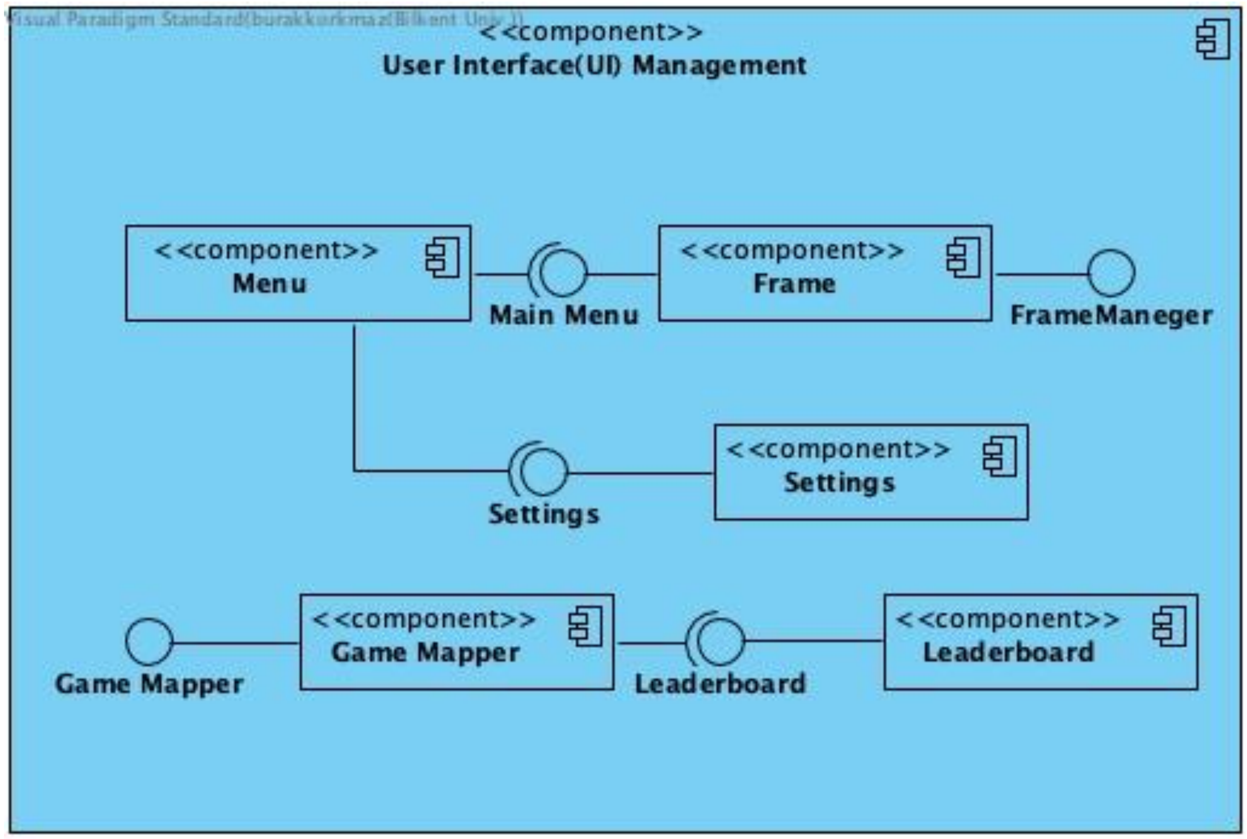
Game data will be stored in the hard drive. We intend to maintain an SQL database for the Leaderboards. We will store the time and names of our players there to make a leaderboard. Our sound and image files will be stored unencrypted as it would have no effect on the security.

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 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 in the game provided by the frame component. Every component in the UI management subsystem is a frame component.

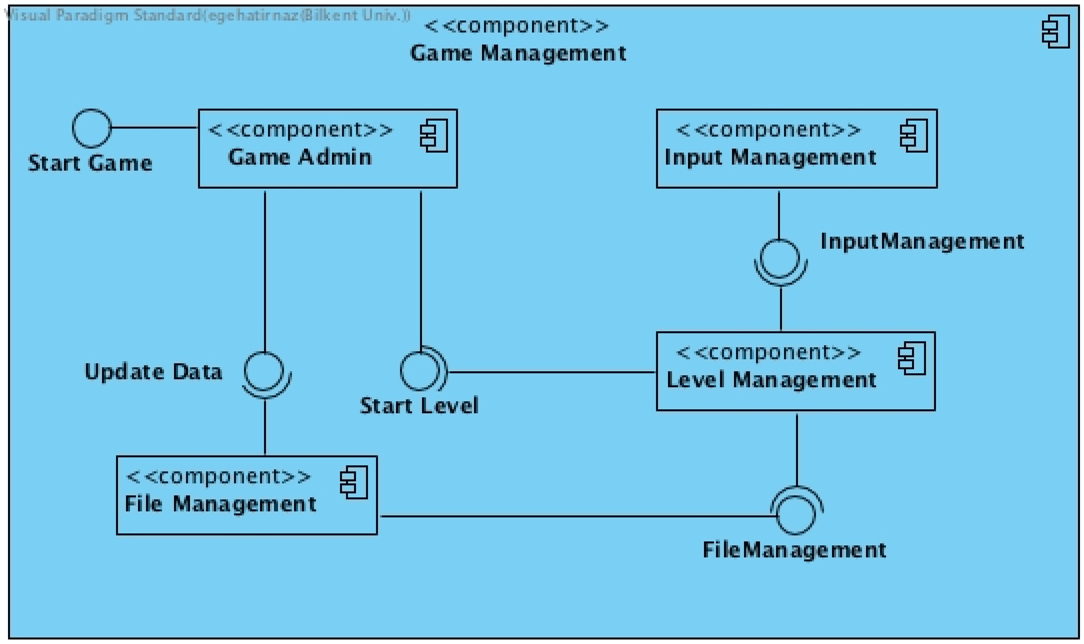
Menu Component: Game Modes in the game, “Classic Mode”, “Challenge Mode”, “Dynamic Mode”, “Credits”, “Leaderboard” and “How To Play” is selected and kept in the Menu Component. Menu component is the main bridge between the other components and Frame component. For example, when a mode is selected, Game Mapper component is triggered and send all UI data to Frame Component. This case is also valid for Settings and Leaderboard components.

Game Mapper Component: Provides the interface of the gameplay structures such as blocks, boards, sticks, and time.. Main flow of the game is found in the Game Mapper. This component invokes also Leaderboard and Menu components.

Settings Component: Adjusting volume and language selections are located in the settings component. It is invoked by Menu Component.

Leaderboard Component: It connects the leaderboard database with the game. Leaderboards of the different game modes can be seen in this component. It is invoked by Menu Component. The data is come from Leaderboard database system.

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 Management
- Level Management
- Input Management
- File Management

Game Management:
This component creates an instance of the game and takes the input from the user. Game Management starts the game, initializes the puzzles and proceeds onto the next level when the solution has been found.

Level Management:
This component is responsible for creating the objects and placing those components in the puzzle level.

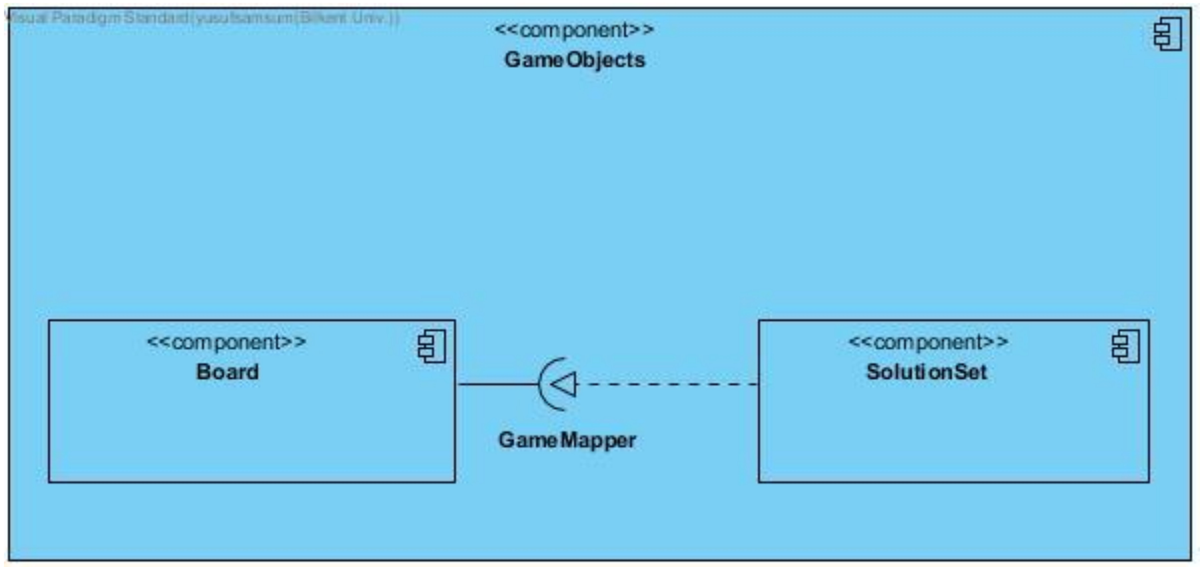
Input Management:

This component receives the user mouse input and takes the input information to Level Management to put the pieces into the puzzle.

File Management:

This component handles the files such as background images or soundtracks. This component is called from Level Management to initialize the external files stored in the game.

3.3 Game Objects Subsystem

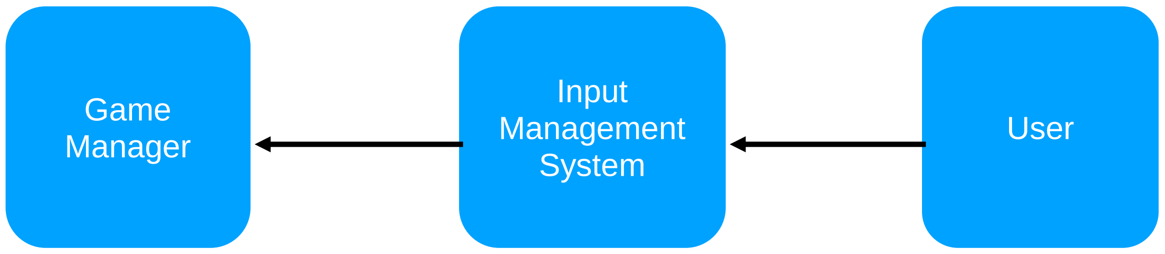


Game Objects subsystem keeps all game objects. Solution set realization is done by the GameMapper object. Solution set has the all blocks with appropriate level and sets. Board has the board width and height with respect to current level and mode.
All these components are realized in GameMapper class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

We are planning to try to use different object design patterns for different solutions.

4.1.1 Bridge Pattern

Our motivation to choose that design pattern is the incompleteness of our object design. The full set of objects is not completely known at current time. Also, the subsystems and components could be replaced later after the system has been deployed. In addition, the bridge pattern can be used for multiple implementations under the same interface and we thought that our different game modes can be fitted for that design pattern.

4.1.2 Command Pattern

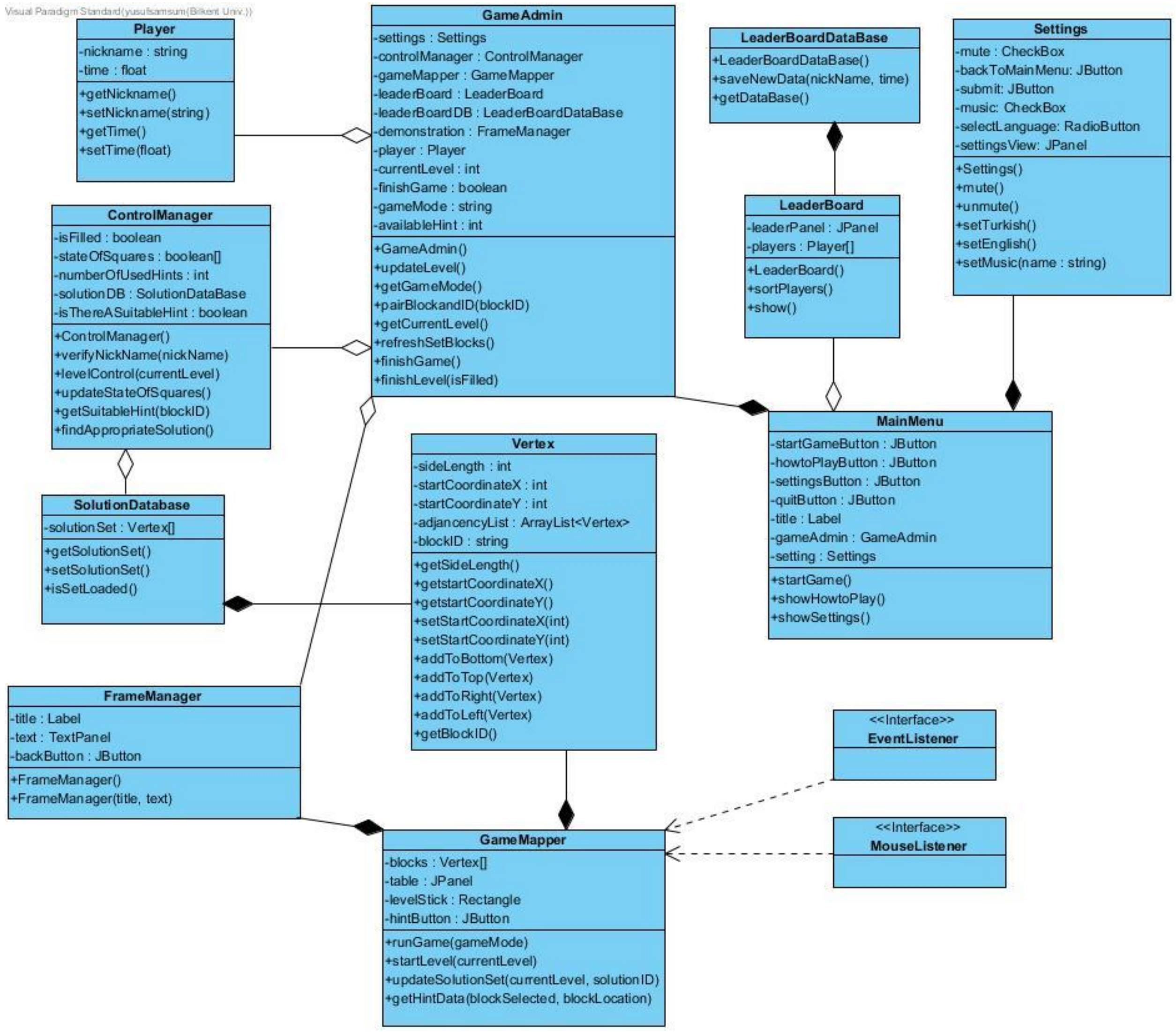
Command pattern seems to be suitable for our UI design. Our UI design consists of many menus because of the game modes. Actually, we offer 3 different type of application in the same game menu and we will implement our game like 3 different game. After that we are planning to merge operation for our games. Therefore, command pattern is the most appropriate pattern for our UI design purposes.

4.1.3 Façade Pattern

Our implementation consists of 4 subsystems. Each subsystem has UI class on their own (See section 3.1). Therefore, we can follow façade design pattern easily and we can provide a unified interface to a set of objects in a subsystem. For example, in one of our subsystem is Leaderboard. Leaderboard subsystem has an interface class for its own sake. Although this implementation strengthens us in terms of efficiency, in some cases there can be misused, leading to non-portable codes.

4.2 Final Object Design

In our final object design decision seems to be façade pattern but still we are working on it. Therefore, our object design is currently like that:



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 JOGL

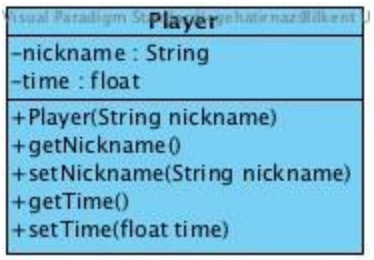
This package will provide classes which are useful for computer graphics. Java Open GL has strong qualifications about graphics and it is faster than FX.

4.3.2.6 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):** Creates a player object with given nickname and time of the object becomes 0 as default value.

Methods:

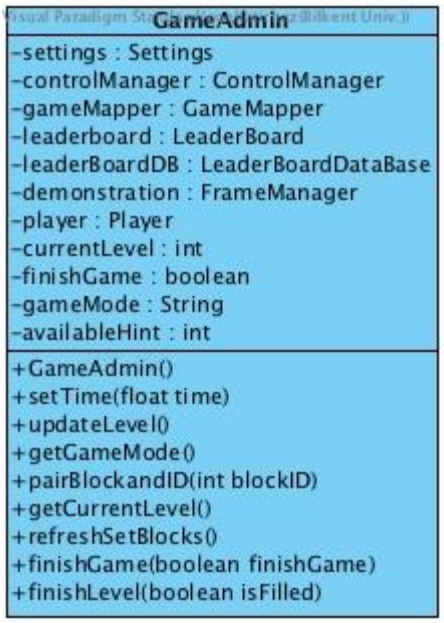
- **String getNickname():** Get the nickname of a player object.

- **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. GameAdmin Class



```
GameAdmin
-settings : Settings
-controlManager : ControlManager
-gameMapper : GameMapper
-leaderboard : LeaderBoard
-leaderBoardDB : LeaderBoardDataBase
-demonstration : FrameManager
-player : Player
-currentLevel : int
-finishGame : boolean
-gameMode : String
-availableHint : int
+GameAdmin()
+setTime(float time)
+updateLevel()
+getGameMode()
+pairBlockandID(int blockID)
+getCurrentLevel()
+refreshSetBlocks()
+finishGame(boolean finishGame)
+finishLevel(boolean isFilled)
```

Attributes:

- **Settings settings:** A Setting object which is used in order to provide connection between GameAdmin class and Settings class which controls music, volume and control setting.

- **ControlManager controlManager:** A controlManager object to control whether game or level is finished, or not in GameAdmin class.

- **GameMapper gameMapper:** A GameMapper object which is used in order to provide connection between GameAdmin class and GameMapper class which controls demonstration of the game play screen and starting of the game.

- **LeaderBoard leaderboard:** A leaderboard object which is used in order to provide connection between GameAdmin class and LeaderBoard class which controls leaderboard properties.

- **LeaderBoardDataBase leaderBoardDB:** A LeaderBoardDataBase which is used in order to provide connection between GameAdmin class and

LeaderBoardDataBase class which controls the database of players and their time. A new player will be added to database by this object

- **FrameManager demonstration:** This FrameManager object creates the main frame of the whole game. The frame will not change for every different choice which made by user.

- **Player player:** A player object which is used in order to create a new player and provide connection between GameAdmin class and this player. The nickname and time of this player are used for LeaderBoardDataBase class.

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

- **boolean finishGame:** Control that whether game is finished, or not. When currentLevel equals to maximum level + 1, finishGame becomes true.

- **String gameMode:** Shows the chosen game mode by user. GameAdmin class sends this data to GameMapper class to start the game.

- **int availableHint:** Shows the number of the hints which can be given to user. For every given hint, availableHint will decrease 1.

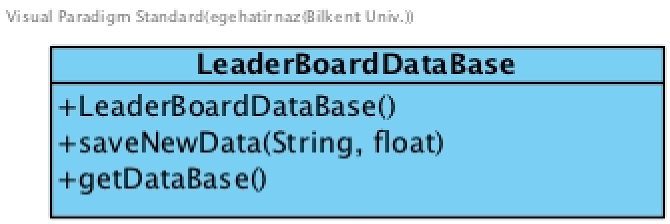
Constructor:

- **GameAdmin ():** Creates a GameAdmin object with default values (Default constructor).

Methods:

- **void updateLevel():** When the level finished, updates the level. This means current level will increase 1.
- **String getGameMode():** Get the chosen gameMode by the user.
- **void pairBlockandID(int blockID):** That method assigns a random ID for each block in the game.
- **int getCurrentLevet():** Get the currentLevel.
- **void refreshSetBlocks():** Refresh the set of blocks. Set of blocks are the given blocks to user in order to fill the game board. In other terms, it gives to user a new solution set.
- **void setTime(float time):** Set the player object’s time as given time.
- **void finishGame(boolean finishGame):** It finishes the game when finishGame attribute in the GameAdmin class becomes true.
- **void finishLevel(boolean isFilled):** It finishes the game when isFilled attribute of ControlManager class becomes true.It calls updateLevel() method and controls whether game is finished, or not.

4.4.3. LeaderBoardDataBase Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
LeaderBoardDataBase
+LeaderBoardDataBase()
+saveNewData(String, float)
+getDataBase()

Constructor:

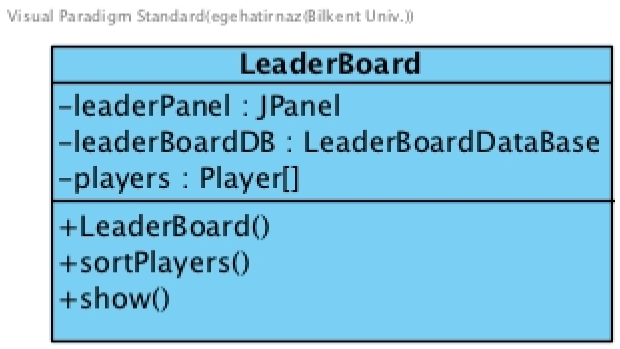
- **LeaderBoardDataBase()**: Creates an object of the class in order to be used in other classes. (Default constructor.)

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.

4.4.4. LeaderBoard Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
LeaderBoard
-leaderPanel : JPanel
-leaderBoardDB : LeaderBoardDataBase
-players : Player[]
+LeaderBoard()
+sortPlayers()
+show()

Attributes:

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

- **LeaderBoardDataBase leaderBoardDB:** A LeaderBoardDataBase object which is used to take information from database by using getDataBase() method of LeaderBoardDataBase class.

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

Constructor:

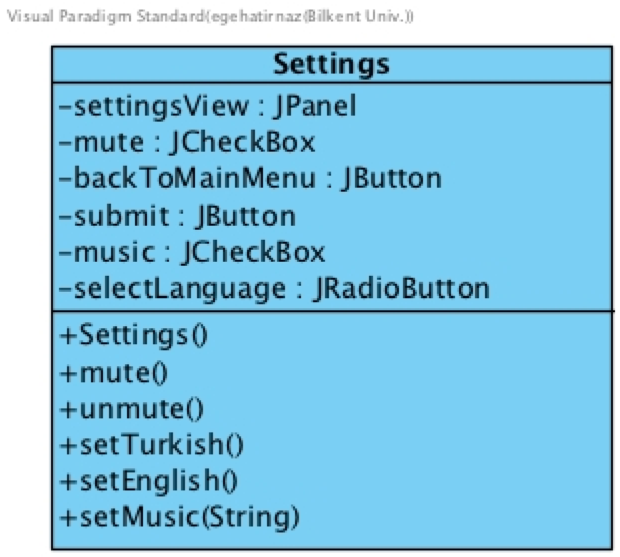
- **LeaderBoard():** Creates an empty leaderboard (Default constructor).

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.

4.4.5. 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.

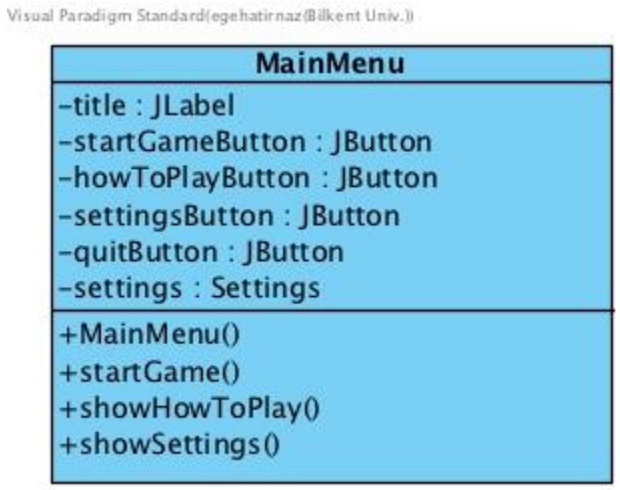
- **void unmute():** If users uncheck ‘mute JCheckBox’, this method makes the game voiced.

- **void setTurkish():** If users chooses ‘Turkish JRadioButton’, this method makes the language of game Turkish.

- **void setEnglish():** If users chooses ‘English JRadioButton’, this method makes the language of game English.

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

4.4.6. 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:

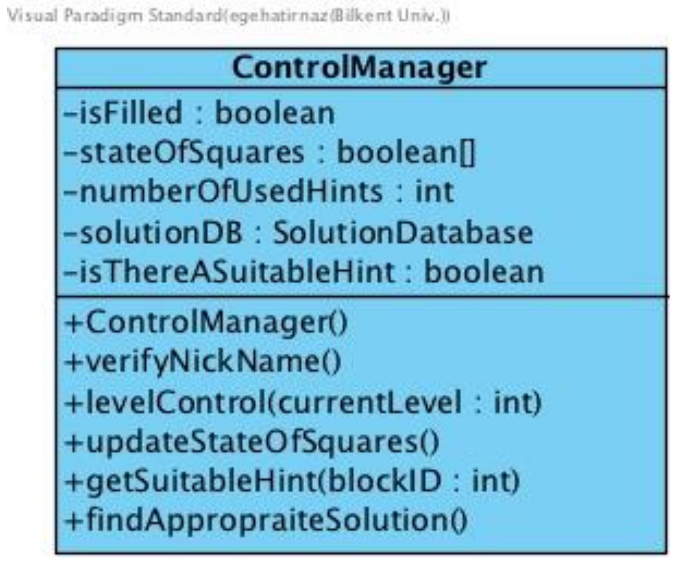
- **MainMenu():** Creates a MainMenu object (Default constructor).

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.

4.4.7. ControlManager Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
ControlManager
-isFilled : boolean
-stateOfSquares : boolean[]
-numberOfUsedHints : int
-solutionDB : SolutionDatabase
-isThereASuitableHint : boolean
+ControlManager()
+verifyNickName()
+levelControl(currentLevel : int)
+updateStateOfSquares()
+getSuitableHint(blockID : int)
+findAppropraiteSolution()

Attributes:

- **boolean isFilled:** A boolean which is used to control whether the gameboard is filled with given blocks, or not. When it is filled, isFilled attribute becomes true.

- **boolean[] stateOfSquares:** Filling state of board's squares are determined by this boolean array.

- **int numberOfUsedHints**: Shows the number of the hints which are be used by player.

- **SolutionDatabase solutionDB:** A SolutionDatabase object which is used to find appropriate solutions when users want a new solution set.

- **boolean isThereASuitableHint:** Shows that whether there is a suitable hint, or not with respect to positioned blocks by user. If there is at least one suitable hint, isThereASuitableHint becomes true.

Constructor:

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

Methods:

- **boolean verifyNickName(String nickname):** Controls whether the nickname of new user is taken previously, or not. If it is not taken yet, returns true.

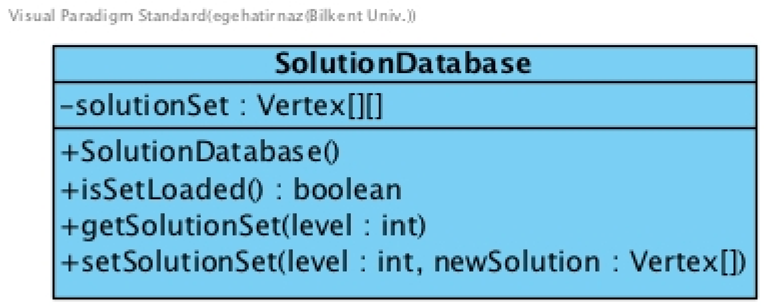
- **boolean levelControl(int currentLevel):** That method checks whether the game in the current level or not and return true if game level and the level in control are same.

- **void updateStateOfSquares():** That method update the game field automatically inside the game.

- i**nt getSuitableHint(int gameLevel, Block [] blockList):** The method will return an index of suitable block will be hinted considering of the gameLevel and blockList parameters(that hint will be between blockList array).

- **Block[] findAppropraiteSolution():** In order to process the game, Game Manager should choose one of the solutions for the played field. This method helps Manager to find the best solution in order to offer player and lead the game. It will return the best solution Block array when it find the best solution and Game Manager will measure the player’s preferences by the help of the solution set.

4.4.8. SolutionDatabase Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
SolutionDatabase
-solutionSet : Vertex[][]
+SolutionDatabase()
+isSetLoaded() : boolean
+getSolutionSet(level : int)
+setSolutionSet(level : int, newSolution : Vertex[])

Attributes:

- **Vertex[][] solutionSet:** Every single object of Vertex class represents a block. solutionSet means, the set (array) of blocks (vertices) which will be used in order to fill the game board and finish the level. solutionSet is different for every level. Therefore, the array is two dimensional. One of the dimensions determines the level, the other one keeps the solution sets. n^th level needs n blocks so solutionSet[n][]’s first n elements (0., 1., 2., …, n-1.) shows the first solutionSet of n’th level. The other n elements (n., n+1., n+2., …, 2n-1.) shows the second solution set. It can continue like this.

Constructor:

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

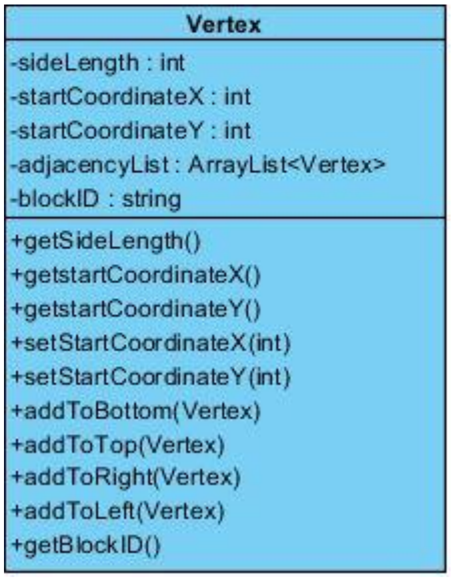
Methods:

- **Vertex[] getSolutionSet(int level):** Get all solution sets of given level. When users want to refresh solution, a new solution can obtain from solutionSet of given level.

- **void setSolutionSet(int level, Vertex[] newSolution):** Adds a new solution for given level to solution database. newSolution is the array of blocks[vertex].

- **boolean isSetLoaded():** Controls whether new solution set which will be given to player is one of the given ones previously, or not.

4.4.9. Vertex Class



```
Vertex
-sideLength : int
-startCoordinateX : int
-startCoordinateY : int
-adjacencyList : ArrayList<Vertex>
-blockID : string
+getSideLength()
+getstartCoordinateX()
+getstartCoordinateY()
+setStartCoordinateX(int)
+setStartCoordinateY(int)
+addToBottom(Vertex)
+addToTop(Vertex)
+addToRight(Vertex)
+addToLeft(Vertex)
+getBlockID()
```

Attributes:

- **int sideLength:** The side length of the squares(vertices) which create the blocks. This will be used to draw a square.

- **int startCoordinateX:** The X coordinate of a square's left top corner. This will be used to draw a square.

- **int startCoordinateY:** The Y coordinate of a square's left top corner. This will be used to draw a square.

- **ArrayList<Vertex> adjacency List**: The array of vertices, these vertices will create a block.

- **String blockID:** Every different block (adjacencyList) will have an ID. The name of this ID is blockID.

Constructor:

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

Methods

- **int getSideLength():** Get the side length of a square(vertex).

- **int getStartCoordinateX():** Get the X coordinate of a square’s (vertex) left top corner.

- **int getStartCoordinateY():** Get the Y coordinate of a square’s (vertex) left top corner.

- **void setStartCoordinateX(int X):** Set the X coordinate of a square’s (vertex) left top corner.

- **void setStartCoordinateY(int Y):** Set the Y coordinate of a square’s (vertex) left top corner.

- **void addToBottom(Vertex thePreviousOne):** Draw a new square (vertex) to the bottom of the previous vertex (thePreviousOne) by setting the startCoordinateY of new vertex to (startCoordinateY of previous one – sideLength).

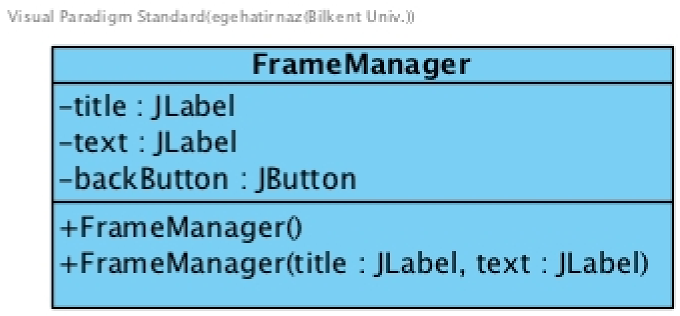
- **void addToTop(Vertex thePreviousOne):** Draw a new square (vertex) to the top of the previous vertex (thePreviousOne) by setting the startCoordinateY of new vertex to (startCoordinateY of previous one + sideLength).

- **void addToRight(Vertex thePreviousOne):** Draw a new square (vertex) to the right of the previous vertex (thePreviousOne) by setting the startCoordinateX of new vertex to (startCoordinateX of previous one + sideLength).

- **void addToLeft(Vertex thePreviousOne):** Draw a new square (vertex) to the bottom of the previous vertex (thePreviousOne) by setting the startCoordinateX of new vertex to (startCoordinateX of previous one – sideLength).

- **String getBlockId():** Get the blockID of vertex.

4.4.10. FrameManager Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
FrameManager
-title : JLabel
-text : JLabel
-backButton : JButton
+FrameManager()
+FrameManager(title : JLabel, text : JLabel)

Attributes:

- **JLabel title:** Creates the title area of the frame.

- **JLabel text:** Creates the text area of the frame under the title.

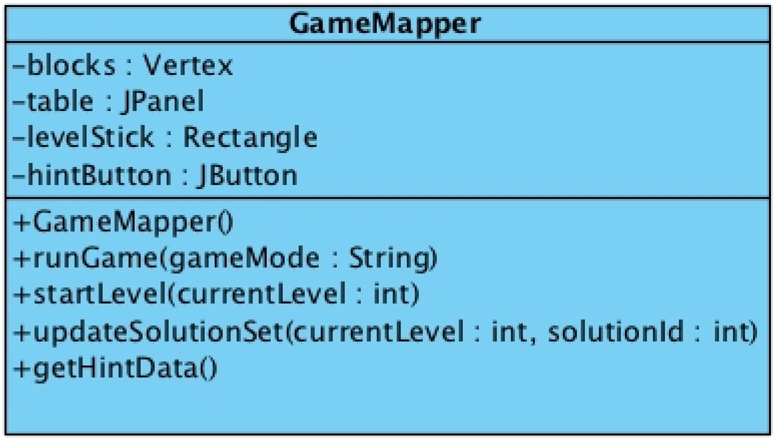
- **JButton backButton:** Creates a “return/back” button. If the user clicks on it, the user goes back.

Constructor:

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

- **FrameManager(JLabel title, JLabel text):** Creates a FrameManager object with predefined strings for title and text area.

4.4.11. GameMapper Class


GameMapper
-blocks : Vertex
-table : JPanel
-levelStick : Rectangle
-hintButton : JButton
+GameMapper()
+runGame(gameMode : String)
+startLevel(currentLevel : int)
+updateSolutionSet(currentLevel : int, solutionId : int)
+getHintData()

Attributes:

- **Vertex[] blocks:** There will be blocks in game play screen. These blocks are created by vertex. Therefore “blocks attribute” is an array of vertices.

- **JPanel table:** The game table is created from JPanel.

- **Rectangle levelStick:** There will be a stick on the game table. This stick will be drawn as a rectangle.

- **JButton hintButton:** Creates a “hint” button. If users click it, it shows a hint to user if there is an appropriate solution.

Constructor:

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

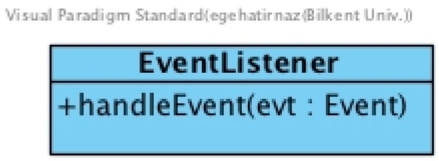
Methods:

- **void runGame(String gameMode):** Starts the game in given gameMode.

- **void startLevel(int currentLevel):** Starts the current level of the player.

- **Block [] updateSolutionSet(int currentLevel):** This method will update the solution set of the problem according to descending levels and return the set.

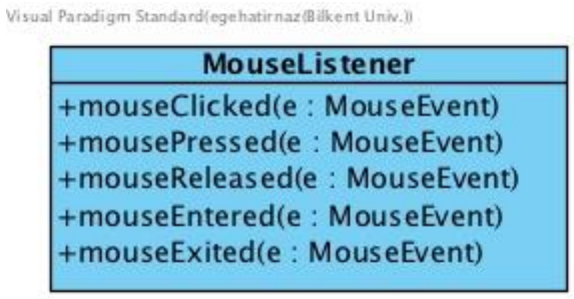
4.4.12. EventListener Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
EventListener
+handleEvent(evt : Event)

Methods:

- **void handleEvent(Event evt):** Gets called whenever an event occurs of the type for which the EventListener interface was registered.

4.4.13. MouseListener Class


Visual Paradigm Standard(egehatirnaz(Bilkent Univ.))
MouseListener
+mouseClicked(e : MouseEvent)
+mousePressed(e : MouseEvent)
+mouseReleased(e : MouseEvent)
+mouseEntered(e : MouseEvent)
+mouseExited(e : MouseEvent)

Methods:

- **void mouseClicked(MouseEvent e):** Invoked when mouse has been clicked (press & release).
- **void mousePressed(MouseEvent e):** Invoked when mouse button has been pressed.
- **void mouseReleased(MouseEvent e):** Invoked when mouse button has been released.
- **void mouseEntered(MouseEvent e):** Invoked when mouse enters a component.
- **void mouseExited(MouseEvent e):** Invoked when mouse exits a component.

5.0 Glossary & References

[1] Write Once, Run Anywhere (https://en.wikipedia.org/wiki/Write_once,_run_anywhere)

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

[3] Java Runtime Environment (https://techterms.com/definition/jre)

[4] Java Development Kit (https://en.wikipedia.org/wiki/Java_Development_Kit)