
# CS 319 - Object-Oriented Software Engineering

# System Design Report

## **Katamino**

Group 1-G

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 has 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. We intend to make a system easy enough to use and levels in suitable difficulty to satisfy the user when they solve the puzzle.

## 1.2 Design Goals

- **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 prioritise 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 more easy 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.

- 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.

### 1.3 Definitions, Acronyms and Abbreviations

**WORA:** Write Once, Run Anywhere
**JVM:** Java Virtual Machine
**JRE:** Java Runtime Environment
**JDK:** Java Development Kit

# 2. High Level System Architecture

### 2.1 Subsystem Decomposition

### 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

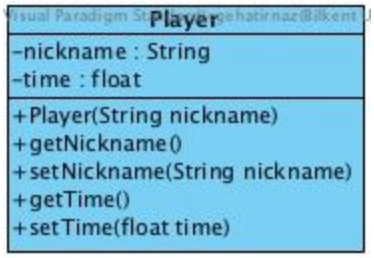
# 3.Subsystem Services

3.1 User Interface Management Subsystem
3.2 Game Management Subsystem
3.3 Game Objects Subsystem
3.4 Input Management Subsystem

# 4. Low-Level Design

4.1 Object Design Trade-offs
4.2 Final Object Design
4.3 Packages
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


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:

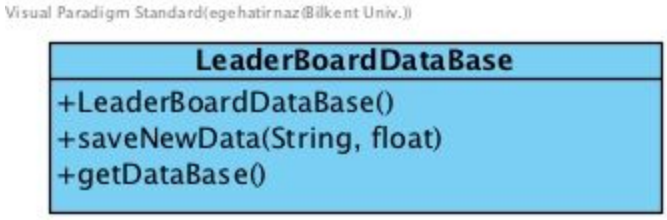
- **GameAdmin ():** Creates a GameAdmin object with default values. (Default constructor.)

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):**
- **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




Constructor:

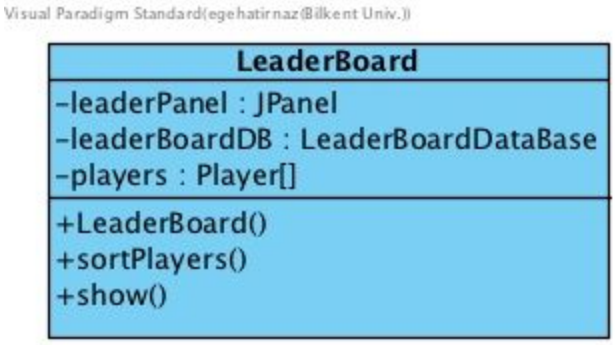
- **LeaderBoardDataBase()**: Creates an object of the class in order to be used in other classes. (Default constructor.)

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




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:

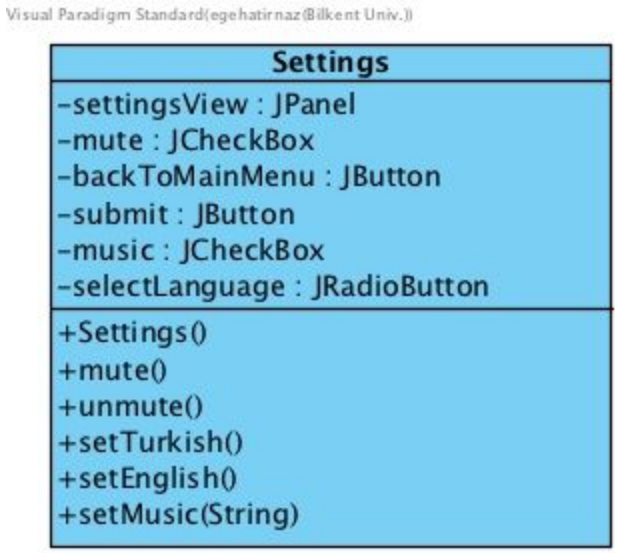
- **LeaderBoard():** Creates an empty leaderboard. (Default constructor.)

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


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:

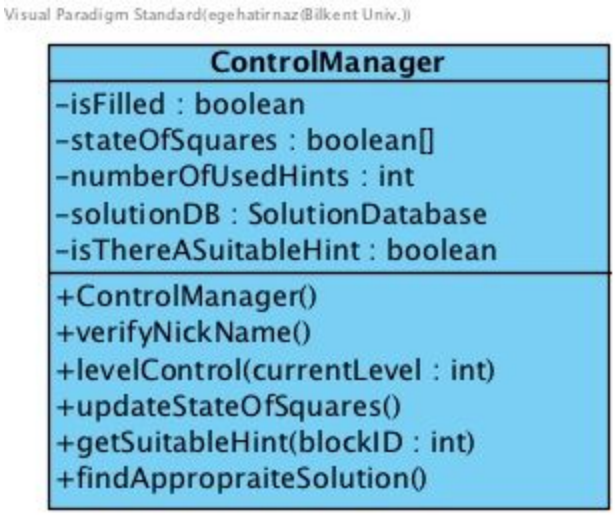
- **MainMenu():** Creates a MainMenu object. (Default constructor.)

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




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:**

- **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.

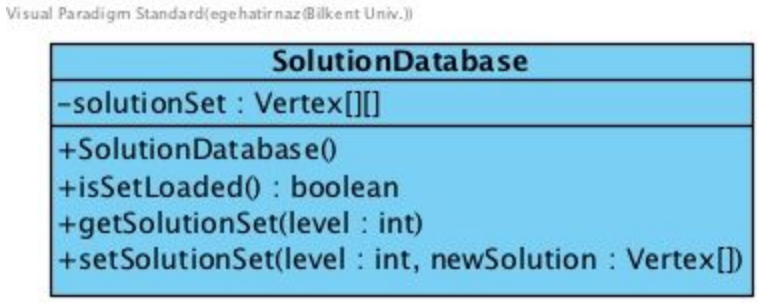
Constructor:

- **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.
- **levelControl(int currentLevel):** Ne döndürüyor bilmiyorum.
- **void updateStateOfSquares():**
- **getSuitableHint(blockID):** Ne döndürüyor bilmiyorum.
- **findAppropraiteSolution():** Ne döndürüyor bilmiyorum.

### 4.4.8. SolutionDatabase Class




Attributes:

- **Vertex[][] solutionSet:** Every single object of Vertex class represents a block. solutionSet means, the set (array) of blocks (vertexes) 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.)

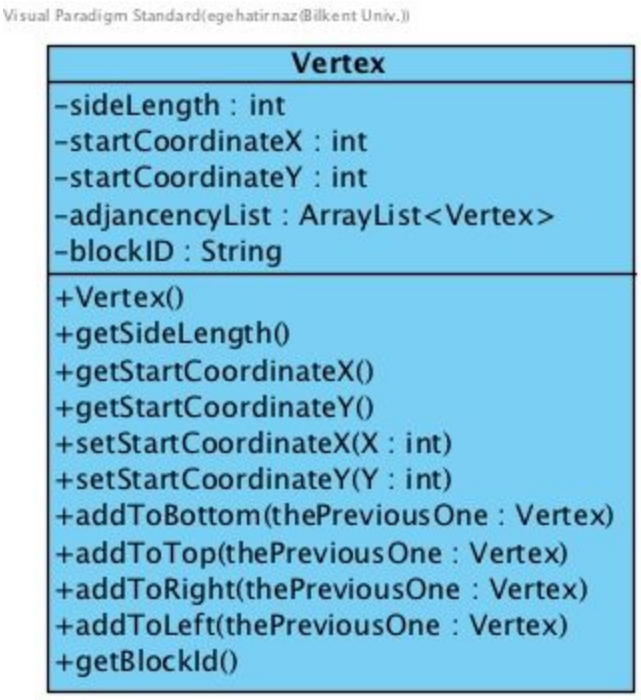
Methods:

- **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




Attributes:

- **int sideLength:** The side length of the squares(vertexes) 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> adjancencyList**: The array of vertexes, these vertexes will create a block.

- **String blockID:** Every different block (adjancencyList) 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


Attributes:

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

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

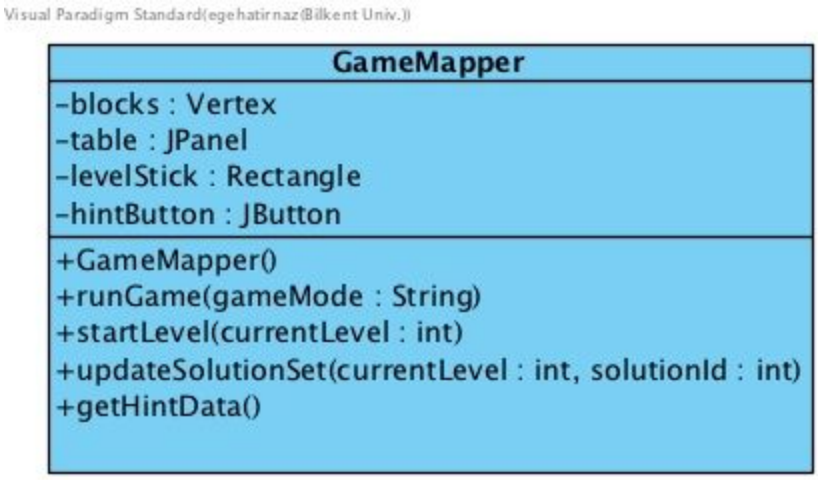
- **JButton backButton:** Creates a “return/back” button. If the user clicks on it, the user goes back.

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




Attributes:

- **Vertex[] blocks:** There will be blocks in game play screen. These blocks are created by vertex. Therefore “blocks attribute” is an array of vertexes.
- **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.
- **updateSolutionSet(int currentLevel,...)**

- **getHintData():**

### 4.4.12. EventListener Class


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


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.

# Glossary & References