CS 319 - Object-Oriented Software Engineering

Design Report

Q-Bitz

Group 1F

Burak Yaşar

Görkem Yılmaz

Hikmet Demir

Mert Duman

Mert Alp Taytak

Supervisor: Eray Tüzün

Contents

1. Introduction

1.1 Purpose of the System

Q-Bitz is a 2-D time-based game which has a main purpose of giving the users an enjoyable game experience while developing their mind skills. The user interface of the system is very user friendly for having simple and easily understandable menu with basic buttons. To increase user friendliness, there exists a How-to-Play button on the main screen to help the users who do not have enough knowledge to play the game. Player can choose the single player mode to challenge himself or the multiplayer mode if he/she wants to compete with other players. The main goal of the player in this system is to make a copy of the given pattern with the available pieces before any other player finishes. Q-Bitz will come with different modes which the user can select from the main menu. The tools which the players will have are a 4-by-4 grid and the necessary pieces in order to make a copy of the given model.

1.2 Design Goals

This project's design goals will be evaluated under two criteria as follows.

1.2.1 End User Criteria

This criteria will be evaluated based on with regard to two parts which are usability and good documentation.

1.2.1.1 Usability

Q-Bitz will provide a very friendly user interface which can be controlled easily with a mouse. This game will also provide a How-to-Play section. In that section, there will be a short but effective tutorial for the players who do not know the Q-Bitz very well. Tutorial will consist of game rules, gameplay and objectives of the Q-Bitz.

1.2.1.2 Good Documentation

Good documentation is also an essential part of a design. Q-Bitz will come out as a well-documented game in a way that all of the game files will be organized and easily understandable by other developers or for any user that would want to examine the software architecture of the system.

1.2.2 Maintenance Criteria

This criteria will be evaluated based on with regard to three parts which are efficiency, modifiability and portability.

1.2.2.1 Efficiency

Efficiency is one of the main goals of the system of Q-Bitz. In order to maintain efficiency, graphics of the system will run in a way that to keep the game performance smooth and without any problems. This can be accomplished with an efficient programming of the GUI. Also, the game will come in a package so that there will not be needing to load any game mode and it will make Q-Bitz more efficient, fast. Only the multiplayer part will have a loading property which depends on the internet connection speed.

1.2.2.2 Modifiability

The system of the Q-Bitz is being developed with the usage of the object-oriented programming rules and principles. The method will guarantee that Q-Bitz system has modifiability as a maintenance criteria. There will be multiple meaningful subsystems of the Q-Bitz system and all of the subsystem will be responsible for a different task of the Q-Bitz. In order to make sure that there will not be any malfunctions in the other systems when there is a minor a major change in a specific subsystem, the MVC architectural style will be used.

1.2.2.3 Portability

Since Q-Bitz will be implemented with the Java programming language, portability will become a criteria that Q-Bitz has very well. Because of the Java language, Q-Bitz will be able to run in Windows, Mac and Linux operating systems as long as the latest Java updates had been downloaded to the computers with those operating systems.

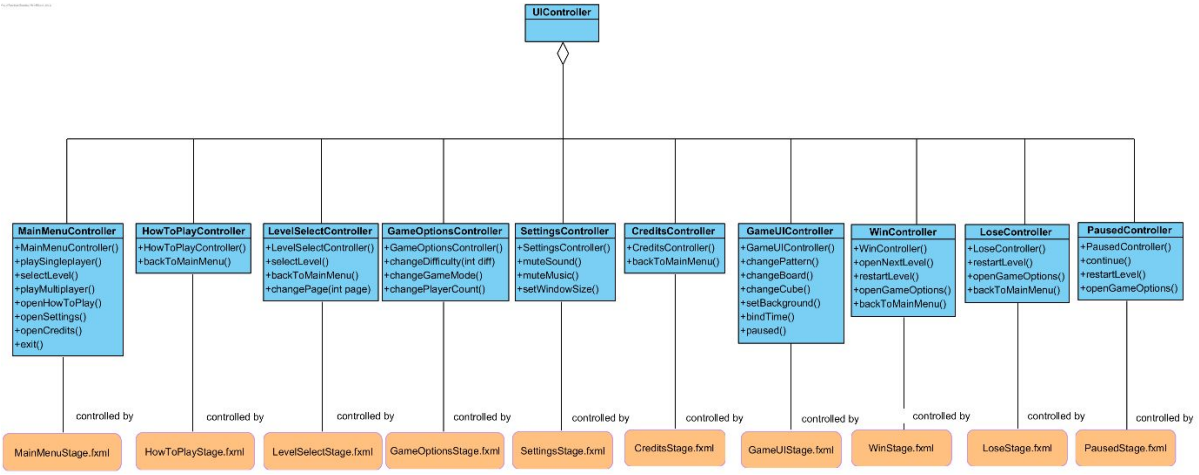
1.3 Definitions, Acronyms and Abbreviations

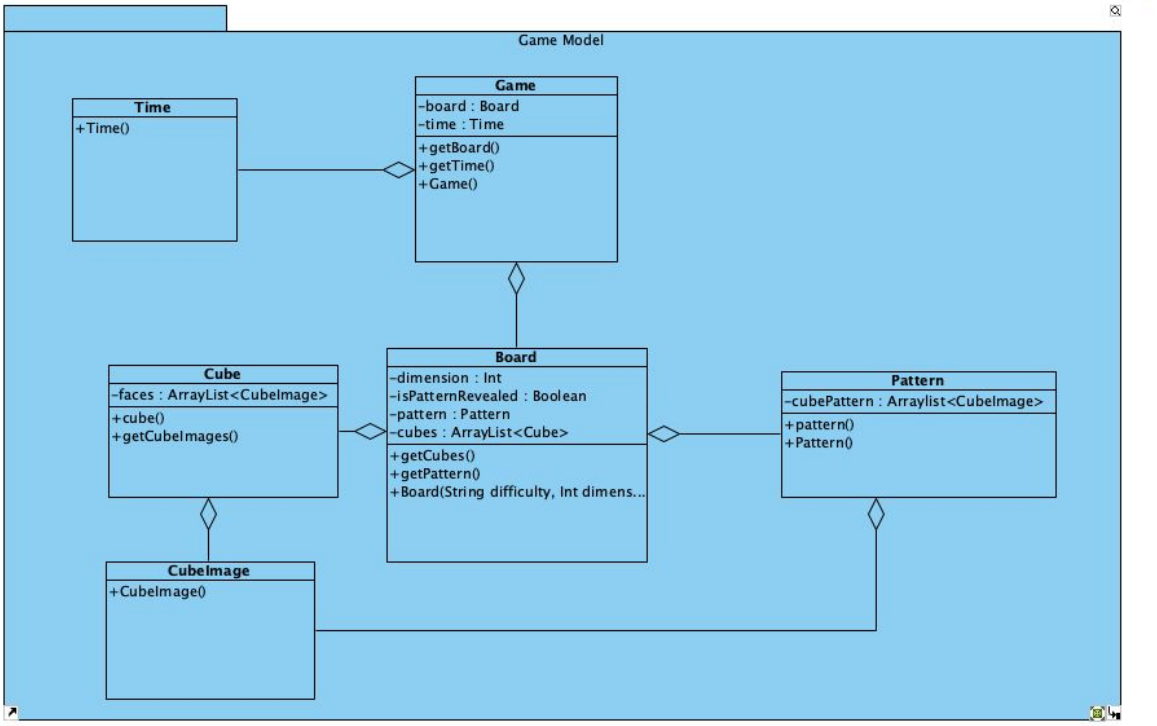
Model View Controller (MVC)
Java Development Kit (JDK)
Java Virtual Machine (JVM)
Graphical User Interface(GUI

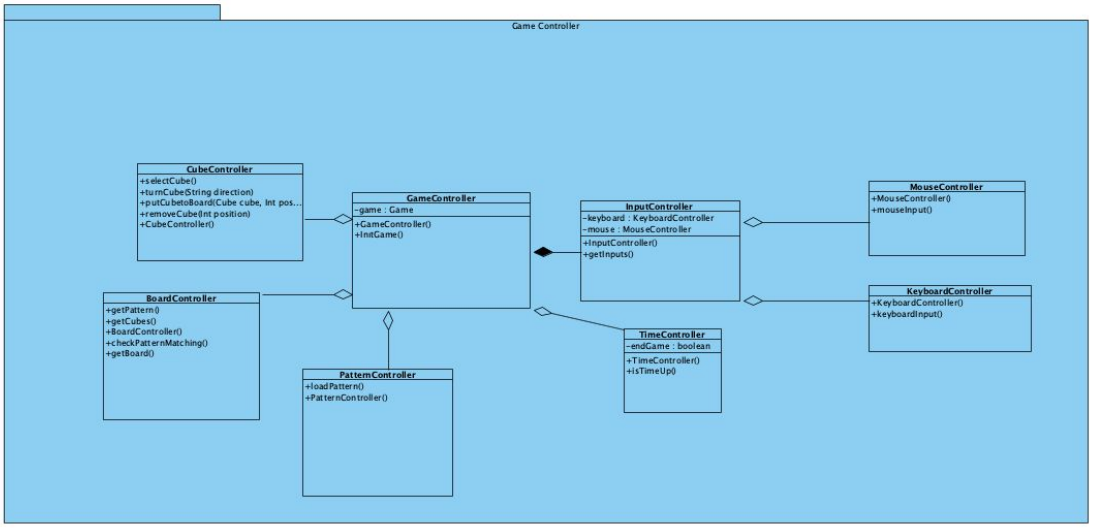
2. System Architecture

2.1 Subsystem Decomposition

In our project we use the Model View Controller (MVC) design pattern. The reason we are using MVC is because it is a flexible pattern that allows the developers to work independently without waiting for each other. This flexibility and independence also extends to the maintainability of the system in the form that changes can be made without breaking the system. Moreover, MVC allows using other design patterns in the subsystems to further modularize the system.


UIController
MainMenuController
+MainMenuController()
+playSingleplayer()
+selectLevel()
+playMultiplayer()
+openHowToPlay()
+openSettings()
+openCredits()
+exit()
HowToPlayController
+HowToPlayController()
+backToMainMenu()
LevelSelectController
+LevelSelectController()
+selectLevel()
+backToMainMenu()
+changePage(int page)
GameOptionsController
+GameOptionsController()
+changeDifficulty(int diff)
+changeGameMode()
+changePlayerCount()
SettingsController
+SettingsController()
+muteSound()
+muteMusic()
+setWindowSize()
CreditsController
+CreditsController()
+backToMainMenu()
GameUIController
+GameUIController()
+changePattern()
+changeBoard()
+changeCube()
+setBackground()
+bindTime()
+paused()
WinController
+WinController()
+openNextLevel()
+restartLevel()
+openGameOptions()
+backToMainMenu()
LoseController
+LoseController()
+restartLevel()
+openGameOptions()
+backToMainMenu()
PausedController
+PausedController()
+continue()
+restartLevel()
+openGameOptions()
controlled by
MainMenuStage.fxml
HowToPlayStage.fxml
LevelSelectStage.fxml
GameOptionsStage.fxml
SettingsStage.fxml
CreditsStage.fxml
GameUIStage.fxml
WinStage.fxml
LoseStage.fxml
PausedStage.fxml


Game Model
Time
+Time()
Game
-board : Board
-time : Time
+getBoard()
+getTime()
+Game()
Cube
-faces : ArrayList<CubeImage>
+cube()
+getCubeImages()
Board
-dimension : Int
-isPatternRevealed : Boolean
-pattern : Pattern
-cubes : ArrayList<Cube>
+getCubes()
+getPattern()
+Board(String difficulty, Int dimens...
Pattern
-cubePattern : Arraylist<CubeImage>
+pattern()
+Pattern()
CubeImage
+CubeImage()


Game Controller
CubeController
+selectCube()
+turnCube(String direction)
+putCubetoBoard(Cube cube, Int pos...
+removeCube(Int position)
+CubeController()
GameController
-game : Game
+GameController()
+IntGame()
InputController
-keyboard : KeyboardController
-mouse : MouseController
+InputController()
+getInputs()
MouseController
+MouseController()
+mouseInput()
KeyboardController
+KeyboardController()
+keyboardInput()
BoardController
+getPattern()
+getCubes()
+BoardController()
+checkPatternMatching()
+getBoard()
TimeController
-endGame : boolean
+TimeController()
+isTimeUp()
PatternController
+loadPattern()
+PatternController()

The model subsystem consists of the classes that are used to model the game state. Based on the game mode and settings chosen the model can vary.

The controller subsystem interacts with both the model and the view subsystems to facilitate the communication between them.

The view subsystem consists of .fxml files that are a result of the JavaFX framework. Each .fxml file describes a single view of the user interface.

2.2 Hardware/Software Mapping

Our programming language of choice for this project is Java. The project requires Java Runtime Environment (JRE) to run the software, an operating system to store game files and an internet connection for the multiplayer mode. For hardware, along with a computer and a monitor, a keyboard, a mouse and speakers are required for user interaction with the system.

2.3 Persistent Data Management

The game requires extra storage space on the computer to store game files. These game files consist of level objects, level setup, music and configuration files. Single player modes directly make use of the game files stored locally. Multiplayer modes construct levels using the game files stored on the host computer. The host computer keeps its own model of the game played and synchronizes players through the network. After the multiplayer session is over, data created during the game is deleted. Therefore, there are no external databases and only persistent data is locally stored game files.

2.4 Access Control and Security

The game only accesses files that comes with itself and in multiplayer the connection is only used to inform the host computer about player moves and inform the local system about changes to the game state. Also, the multiplayer requires knowing the address of the host to join a session and every player assuming a ready state to start the session. Therefore, the game makes safe use of the system resources and allows the players to get out of an untrusted environment before starting a multiplayer session. Any security risk that

comes from the possibly exploitable nature of the connection between the host and the players are beyond the scope of this project and as a result is unaddressed.

2.5 Boundary Conditions

2.5.1 Initialization

The game will be distributed and installed via a .jar file. The .jar file will contain game files which then will be extracted on the desired location. After the installation, the player can use the .jar file to play the game.

2.5.2 Termination

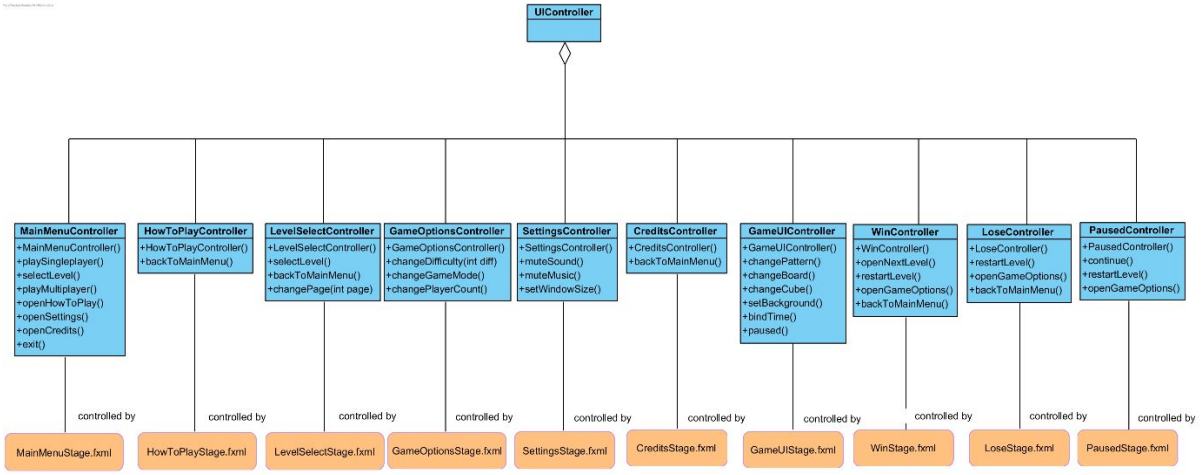
The players can use various exit buttons in the game to quit the game any time they want. Also, it is possible to close the game through the operating system process stopping methods such as clicking the exit button on the application window or killing the process through a task manager.

2.5.3 Failure

Most exceptions will be handled by the application to either find a solution to the problem or informing the user and returning to a stable state. If a failure can not be recovered from, then the application will crash and garbage collector will clean up to free the used system resources.

3. Subsystem Services

3.1 View: User Interface Subsystem



Since we will be implementing our game with the Java language, we will make use of its JavaFX library to handle the view (UI) side of our application. We do not need any view classes because with JavaFX, all of this is handled with the corresponding FXML files that will be loaded onto stages. So, in this section we'll be describing the controller classes for the FXML files in our program.

3.1.1 UIController Class

This class will simply hold references to every other controller. It will be our bridge class between the game manager and UI controllers.

3.1.2 MainMenuController Class

Constructor:

- MainMenuController MainMenuController(): This is the default constructor of MainMenuController.

Methods:

- void playSingleplayer(): This method will start the single player game session. The player will be sent to the game options selection stage.

- void selectLevel(): This method will send the player to the stage to select the level to play.
- void playMultiplayer(): This method will start the multiplayer game session. The player will be sent to the game options selection stage.
- void openHowToPlay(): This method will show a stage where information about how to play the game is displayed.
- void openSettings(): This method will navigate the user to game settings.
- void openCredits(): This method will navigate the user to a stage where credits due will be displayed.
- void exit(): This method will exit the program.

3.1.3 HowToPlayController Class

Constructor:

- HowToPlayController HowToPlayController(): This is the default constructor of HowToPlayController.

Methods:

- void backToMainMenu(): This method will navigate back to the main menu.

3.1.4 LevelSelectController Class

Constructor:

- LevelSelectController LevelSelectController(): This is the default constructor of LevelSelectController.

Methods:

- void selectLevel(): This method will start the game with the currently selected level.
- void changePage(int page): This method will navigate between level pages. Input page is the number of the page to navigate to.
- void backToMainMenu(): This method will navigate back to the main menu.

3.1.5 GameOptionsController Class

Constructor:

- GameOptionsController GameOptionsController(): This is the default constructor of GameOptionsController.

Methods:

- void changeDifficulty(int diff): This method will change the board size to 3x3, 4x4 or 5x5, which is given by the input diff.
- void changeGameMode(): This method will change the game mode.
- void changePlayerCount(): This method will change the number of players to play with.

3.1.6 SettingsController Class

Constructor:

- SettingsController SettingsController(): This is the default constructor of SettingsController.

Methods:

- void muteSound(): This method will mute/unmute the sound.
- void muteMusic(): This method will mute/unmute music.
- void setWindowSize(): This method will change the window size.

3.1.7 CreditsController Class

Constructor:

- CreditsController CreditsController(): This is the default constructor of CreditsController.

Methods:

- void backToMainMenu(): This method will navigate back to the main menu.

3.1.8 GameUIController Class

Constructor:

- GameUIController GameUIController(): This is the default constructor of GameUIController.

Methods:

- void changePattern(): This method will change the current pattern displayed.
- void changeBoard(): This method will change (reset, mix) the current board.
- void changeCube(): This method will change the cube the player is holding.
- void setBackground(): This method will change the background of the game UI.
- void bindTime(): This method will keep decreasing the game time after the game has started.
- void paused(): This method will be called when the player pauses the game and it will display the paused stage.

3.1.9 WinController Class

Constructor:

- WinController WinController(): This is the default constructor of WinController.

Methods:

- void openNextLevel(): This method will open the next level on the level list.
- void restartLevel(): This method will restart the current level.
- void openGameOptions(): This method will open the game options stage and allow the user to modify them.
- void backToMainMenu(): This method will navigate back to the main menu.

3.1.11 LoseController Class

Constructor:

- LoseController LoseController(): This is the default constructor of LoseController.

Methods:

- void restartLevel(): This method will restart the current level.

- void openGameOptions(): This method will open the game options stage and allow the user to modify them.
- void backToMainMenu(): This method will navigate back to the main menu.

3.1.12 PausedController Class

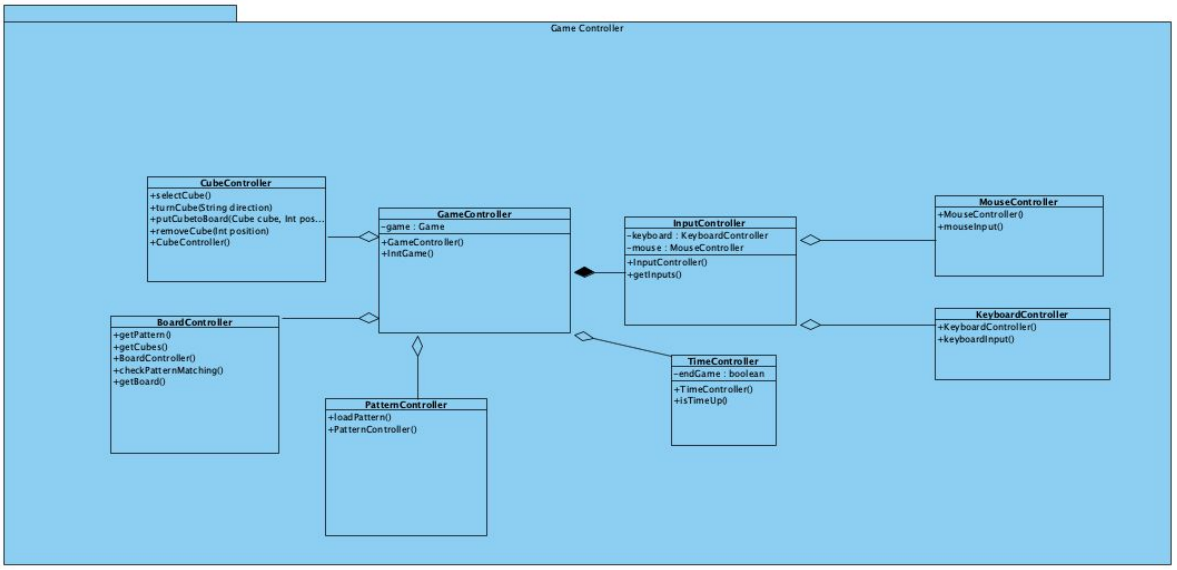
Constructor:

- PausedController PausedController(): This is the default constructor of PausedController.

Methods:

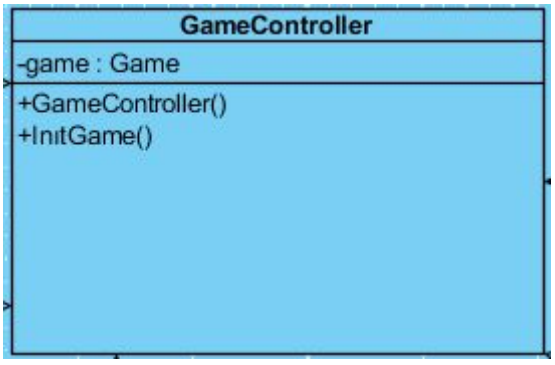
- void continue(): This method will unpause the game.
- void restartLevel(): This method will restart the current level.
- void openGameOptions(): This method will open the game options stage and allow the user to modify them.

3.2 Controller: Game Management Subsystem



In the following controller classes will be explained.

3.2.1 GameManager



Attributes:

- Game game: This is the main game instance; the attributes of the game is controlled and modified with GameManager.

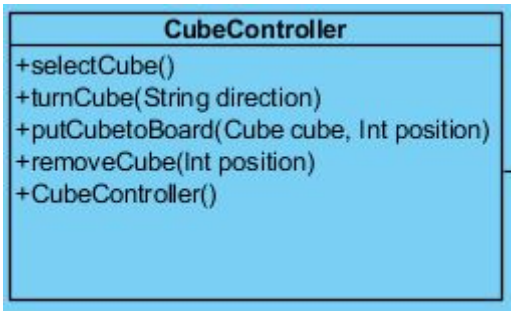
Constructor:

- GameManager GameManager(): Default constructor of GameManager class.

Methods:

- void initGame(): this method initialize the Game with its pattern, cubes and specifies time if the mode requires so.

3.2.2 CubeController



Constructor:

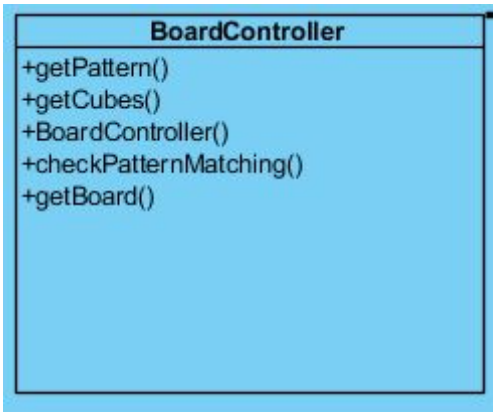
- CubeController CubeController(): CubeController is the default constructor of this class which provides interaction with Cube objects to select them and put it into the board or remove from the specified cell and turn them to get different face of the Cube.

Methods:

- Cube selectCube(): this method will be used to select cube from either outside of the board or inside of a board cell.

- void turnCube(String direction): this method will be used to turn the cube either in horizontal direction or vertical direction which is given in parameter.
- void putCubeToBoard(Cube cube, int position): this method enables a selected cube to be placed into the given position in the board.
- void removeCube(int position): this method enables a cube to be removed from the specified position in the board.

3.2.3 BoardController



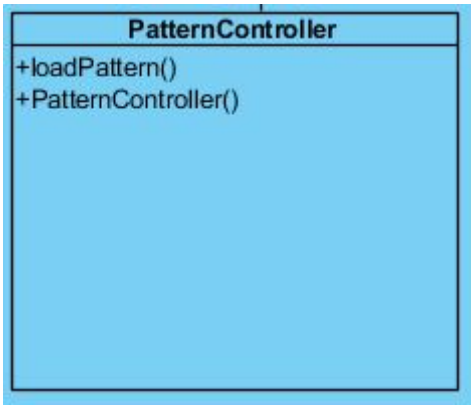
Constructor:

- BoardContoller BoardController: Default constructor of BoardController which can control the Board attributes and modifies them.

Methods:

- Pattern getPattern(): This method will be used to get the pattern which is made for the players to solve.
- ArrayList<Cube> getCubes(): This method is used to get the cubes which will be placed to the board to match the pattern
- boolean checkPatternMatching(): this method checks whether the pattern which is maden by player is matched with the actual pattern.
- Board getBoard(): this method is used to get the initialized the board with its dimension (such as 4*4 or 5*5).

3.2.4 PatternController



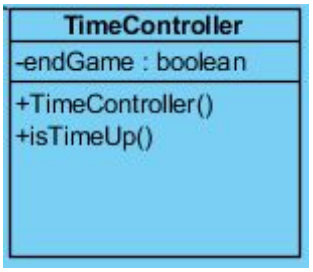
Constructor:

- PatternController PatternController(): this is the default constructor of PatternController.

Methods:

- void loadPattern(): this method loads a random proper pattern in order to be solved.

3.2.5 TimeController



Attributes:

- boolean endGame: this variable checks whether the game is finished because of time is up.

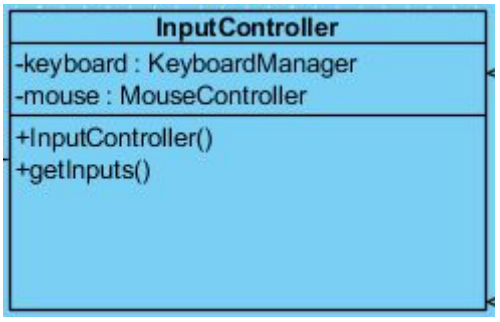
Constructor:

- TimeController TimeController(): default constructor of TimController.

Methods:

- boolean isTimeUp(): this method specifies whether the time is up or not.

3.2.6 InputManager



Attributes:

- KeyboardManager keyboard: this instance is in InputController to manage with the keyboard interactions of the player.
- MouseManager mouse: this instance is in InputController to manage with the mouse interactions of the player.

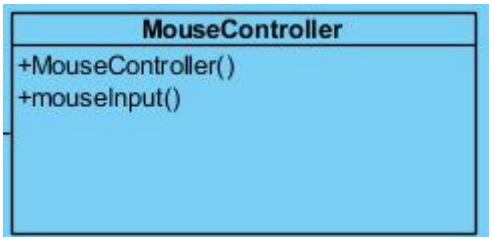
Constructor:

- InputManager InputManager(): Default constructor of InputManager.

Methods:

- getInputs(): this method will be used to get any input action received by the player.

3.2.7 MouseManager



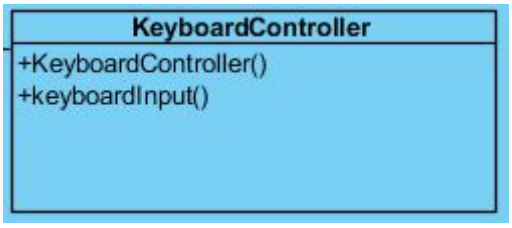
Constructor:

- MouseController MouseController(): Default constructor of MouseController which can control the mouse inputs.

Methods:

- void mouseInput(): this method controls the mouse inputs which is received from the users.

3.2.8 KeyboardManager



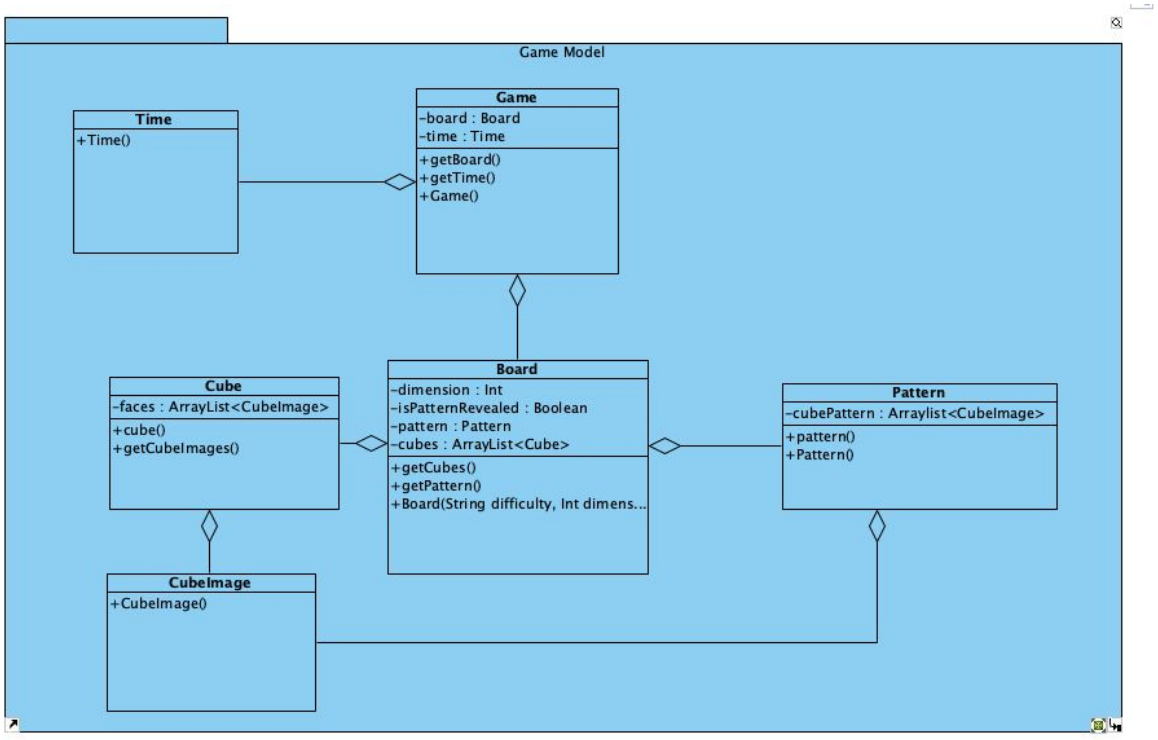
Constructor:

- KeyboardController KeyboardController(): Default constructor of KeyboardController which can control the keyboard inputs .

Methods:

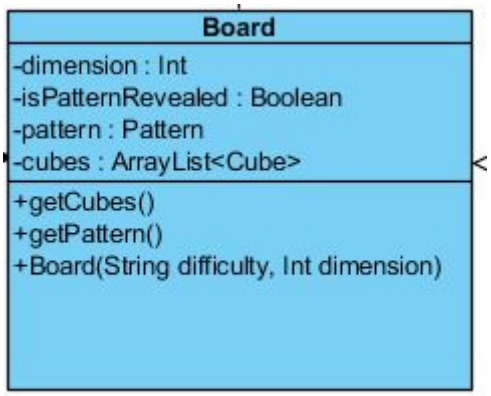
- void keyboardInput() this method controls the key inputs which is received from the players.

3.3 Model: Game Object Subsystem



In the following model classes will be explained.

3.3.1 Board



Attributes:

- Int dimension: This variable will represent the size of the board(3x3, 4x4 or 5x5)
- boolean isPatternRevealed: This variable will check pattern is completed correctly or not. It will be True if player pattern and game pattern is exactly same.
- Pattern pattern: This variable will represent the pattern of the game
- ArrayList<Cube> cubes: This variable will hold the cubes of the game. Player will put this cubes to board, turn them, remove them etc.

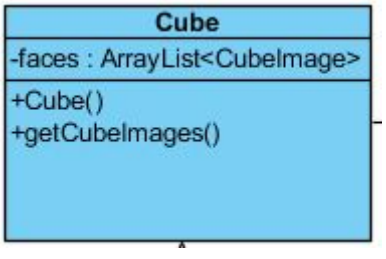
Constructor:

- Board(String difficulty, Int dimension): Default constructor of Board class.

Methods:

- ArrayList<Cube> getCubes(): This method will get the cubes from Cube class
- pattern getPattern(): This method will get the pattern from Pattern class

3.3.2 Cube



Attributes:

- ArrayList<CubeImage> faces: This variable will hold the 6 faces of the cube. Every face will be a CubeImage object.

Constructor:

- Cube(): Default constructor of Cube class.

Methods:

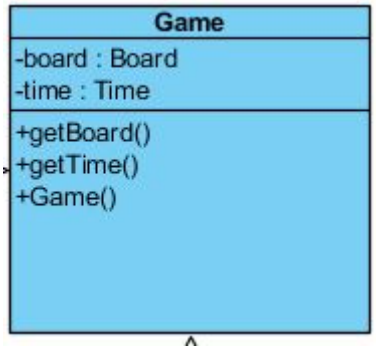
- getCubeImages(): This method will be used to get cube images from CubeImage class

3.3.3 Time

Constructor:

- Time(): Default constructor of Time class.

3.3.4 Game



Attributes:

- Board board: This variable will hold the board of the game
- Time time: This variable will hold the time

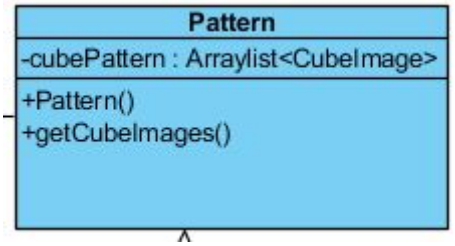
Constructor:

- Game(): Default constructor of Game class.

Methods:

- getBoard(): This method will be used for the getting board from Board class
- getTime(): This method will be used for the getting time from Time class

3.3.5 Pattern



Attributes:

- ArrayList<CubeImage>: This variable will hold the CubeImage

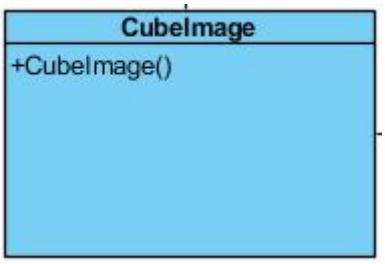
Constructor:

- Pattern(): Default constructor of Pattern class. It

Methods:

- getCubeImages(): This method will be used for the getting cube images from CubeImage class.

3.3.6 CubeImage



Constructor:

- CubeImage(): Default constructor of CubeImage class.

4. Low-Level Design

4.1 Object Design Trade-Offs

Usability vs Functionality:

Our system aims to provide players to play the game easily. We will try to keep the simplicity of original game without breaking its main functionality and provide more game modes for much fun and better experience. Therefore, while we protect the simplicity of the game, we try to improve its functionalities.

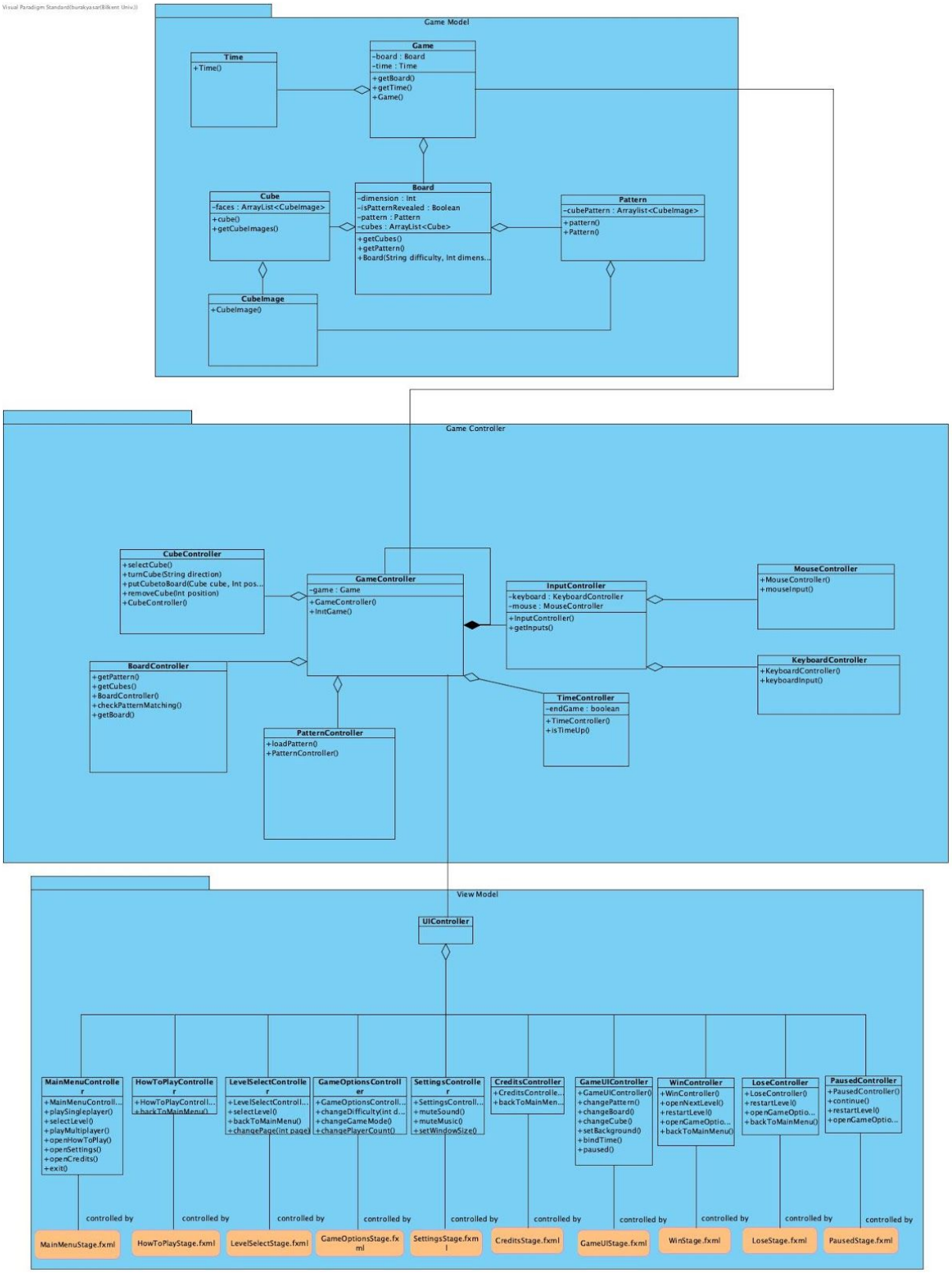
Memory vs Performance:

Our system should consider to supply high performance since the it is an interactive game with the user. Also, according to our design the game has multiplayer mode and it requires some memory space for the database to provide more players to play the game. This leads to consuming more memory than the systems which have no multiplayer system. However, we will mainly focus on the performance rather than the memory. Also, with the object oriented design we will improve the game efficiency and with using abstractions and eliminating unnecessary implementation we can use less memory space.

Portability vs Efficiency:

Our game can be portable and platform independent since we chose Java to build the game. However, different platforms requires some different specifications and since we cannot provide all these specifications, game may not work on every device very well.

4.2 Final Object Design



4.3 Packages

4.3.1 javafx.scene

Provides the core set of base classes for the JavaFX Scene Graph API.

4.3.2 javafx.scene.control

The JavaFX User Interface Controls (UI Controls or just Controls) are specialized Nodes in the JavaFX Scenegraph especially suited for reuse in many different application contexts.

4.3.3 javafx.fxml

Contains classes for loading an object hierarchy from markup.

4.3.4 javafx.scene.image

Provides the set of classes for loading and displaying images.

4.3.5 javafx.scene.input

Provides the set of classes for mouse and keyboard input event handling.

4.3.6 javafx.scene.chart

The JavaFX User Interface provides a set of chart components that are a very convenient way for data visualization.

4.3.7 javafx.util

Contains various utilities and helper classes.

4.4 Class Interfaces

4.4.1 Input Listener

This interface will be invoked if an action occurs with keyboard or mouse, this listener will be notified such events(keyboard or mouse).

4.4.2 Key Listener

This interface will be invoked if an action occurs with keyboard. A key can be pressed or released by user and this listener provides tracking such events.

4.4.3 Mouse Listener

This interface will be invoked if an action occurs with mouse which is received from user so that any mouse action can be tracked.

5. Glossary & References

Bruegge, B., & Dutoit, A. H. (2014). Object-oriented software engineering: using UML, patterns, and Java. Harlow, Essex: Pearson.

“Q-Bitz.” *Timberdoodle Co*, https://timberdoodle.com/products/q-bitz.

“Q-Bitz.” *Amazon.co.uk: Toys & Games*, 7 May 2010, https://www.amazon.co.uk/Green-Board-Games-44002-Q-Bitz/dp/B0031P91LK.

“Q-Bitz Jr.” *TOYTAG*, https://www.toytag.com/products/q-bitz-jr.

TimeToPlayMag. “Q-Bitz from MindWare.” *YouTube*, YouTube, 3 Sept. 2013, https://www.youtube.com/watch?v=j4PEAbkT780.

https://docs.oracle.com/javase/8/javafx/api/