CS 319 - Object-Oriented Software Engineering

Design Report

Iteration 2

Q-Bitz

Group 1F

Burak Yaşar

Görkem Yılmaz

Hikmet Demir

Mert Duman

Mert Alp Taytak

Supervisor: Eray Tüzün

Teaching Assistant: Gülden Olgun

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):** The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.
- **Java Development Kit (JDK):** The JDK is a software package that contains a variety of tools and utilities that make it possible to develop, package, monitor and deploy applications that build for any standard Java platform.

- **Java Virtual Machine (JVM):** It is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.
- **Graphical User Interface(GUI):** This means a user interface that includes graphical elements, such as windows, icons and buttons.
- **Java Runtime Environment(JRE):** It is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

2. System Architecture

To see the final object design, please go to section 4.2.

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.

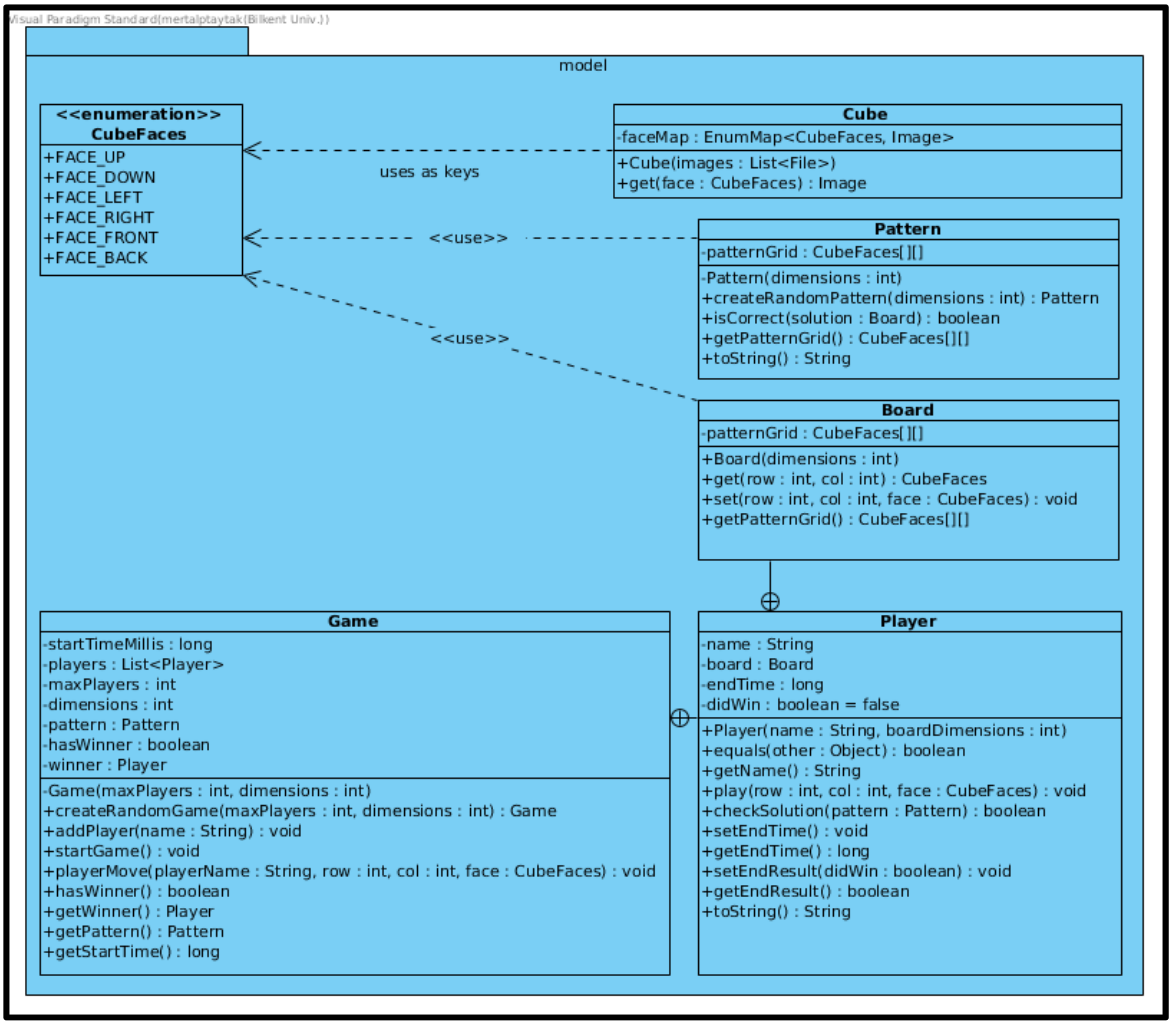

Figure 1

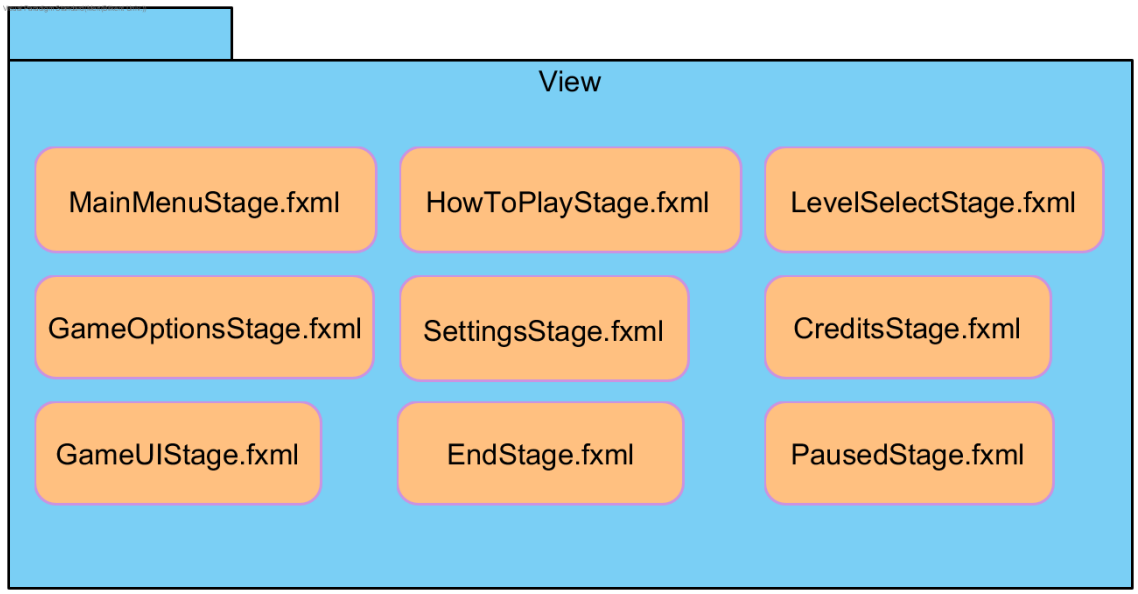

Figure 2

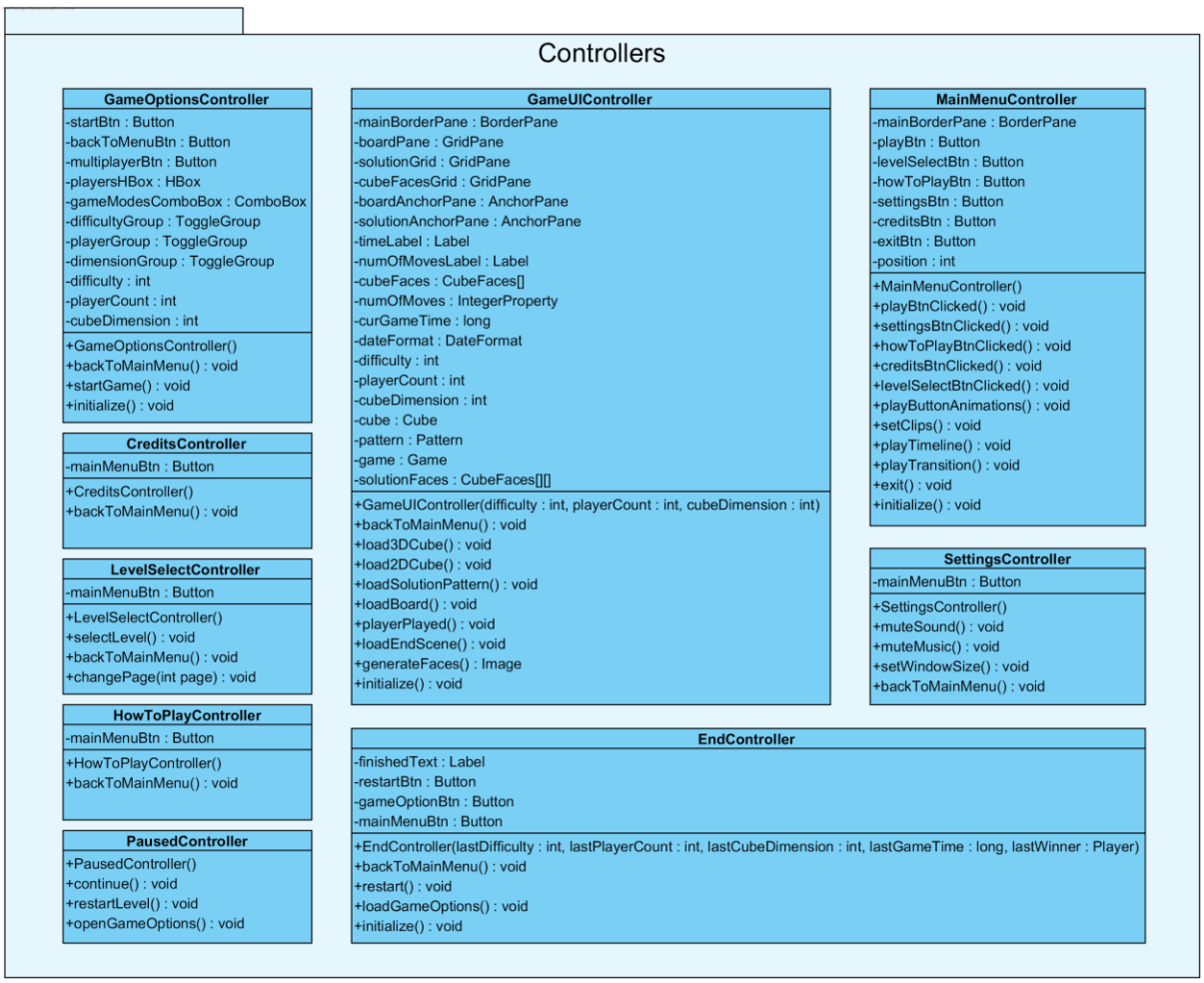

Figure 3

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. Keyboard and mouse will be used to select the buttons, to pick and put the cube, to rotate the cube. Speakers will be used to play the music and sounds of the game. Speakers are not essential for the gameplay, it's just required for music and sounds. Game will need Keyboard and mouse to properly work.

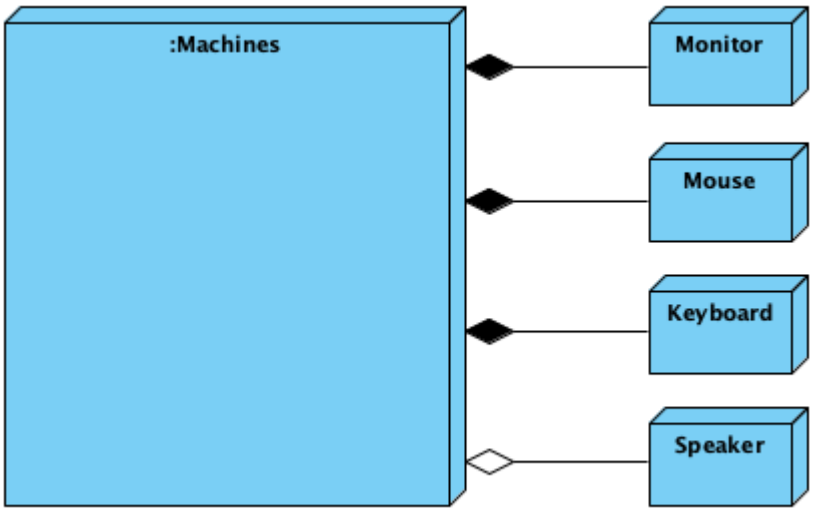

Figure 4

Every machine both (client and host) in the above diagram has these components. And below diagram states our deployment for different types of machine where each of these machine has different behavior.

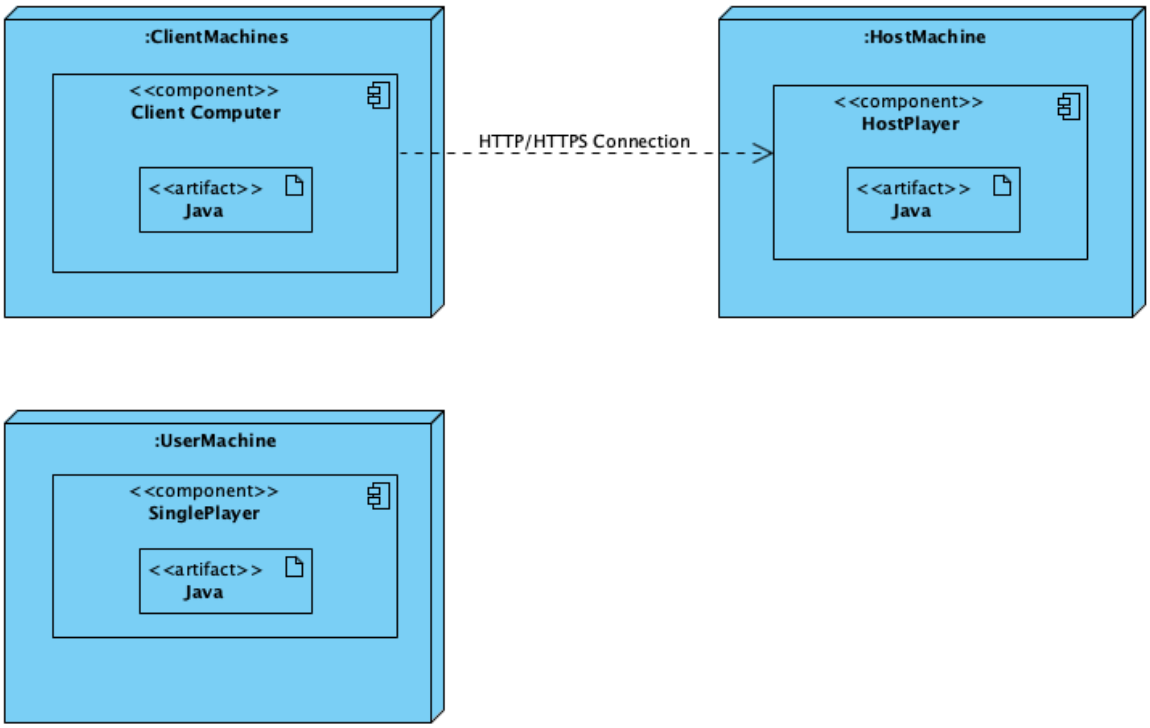

Figure 5

Our system has three different types of nodes. First node represents single player machine which allow users can play the Qbitz on their own machine. Second and third node represents multiplayer case, where second node is our host machine and third node stands for the client machines. Both client and host machines are connected to same local area network. Host machine opens a socket for each client machine in the network. Host machine and client machines communicate with each other over those defined sockets. Every client machine send its data over socket and host machine gather and process all the data and send back to required data back to clients.

2.3 Persistent Data Management

The game requires extra storage space (size of the JAR file planned to be around 50MB) 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. Also, the temporary files (such as state of the players, time, matched pattern count, etc.) corresponding multiplayer game will be hold into the host machine. After the multiplayer session is over, temporary files and data created during the

game is deleted. Therefore, there are no external databases and only persistent data is locally stored game files.

We thought that an implementation of a network is necessary for the multiplayer part of the Q-Bitz since a multiplayer game can be played with the usage of a network. There are two main types of network which are authoritative and non-authoritative. For the Q-Bitz, we decided on using a common approach of the non-authoritative network type and that is the peer to peer approach. In this network system there is no central entity and every peer (in our case the player) can control the game state. Every player constantly receives and sends data from/to other peers and that data is the representation of the game state. For example in the beginning of a multiplayer game of the Q-Bitz, preferably the first player who enters becomes the host and other players who enter become the client. Host player creates sockets for each individual client and enables the data transaction. With the usage of the P2P (peer to peer) network system, players of the Q-Bitz can enjoy a thrilling multiplayer game experience.

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.

	Model	View	Controller	Local Data	Multiplayer Data
Host	No Access	No Access	Read Write	No Access	Read Write
Multiplayer User	Read Write	Read Write	Read Write	Read	No Access
Single Player User	Read Write	Read Write	Read Write	Read Write	No Access

Figure 6

In the previous we have the access control matrix. Here, the host denotes the player that acts as the initiator between players and it is one of the players in the game. More specifically, the host connects to every other player and sets up the environment to create a peer-to-peer network. Since it only establishes connection between the players it only

requires access to the network related controllers and multiplayer data. In the table, multiplayer data denotes the host data that is used to create the network. Each multiplayer user runs their own model of the game that gets updated via the controller that connects to the peer-to-peer network. Thus, they have full access to MVC classes of the game. Following, the local data consists of the game visuals and sounds. Therefore, in order to construct a playable game local data is required. Hence the read access for both types of players. Lastly, we have the single player user. The name single player user is self-explanatory. The player needs full access to every game related class and local data to play the game. Hence, it has access to everything except the multiplayer data. For which it has no use.

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 Normal 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. The game will save the persistent data (highscore) to the file that will be determined later. Terminating the application will be handled by Java Runtime Environment.

2.5.3 Termination with 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 cannot be recovered from, then the application will crash and garbage collector will clean up to free the used system resources.

If the game terminates accidentally during the gameplay, the data of the game will be lost. The game will store the high scores in local files (i.e JSON file) but it will not store the gameplay data in other words one player cannot continue from where he/she left if application terminates with failure.

3. Subsystem Services

3.1 View: User Interface Subsystem

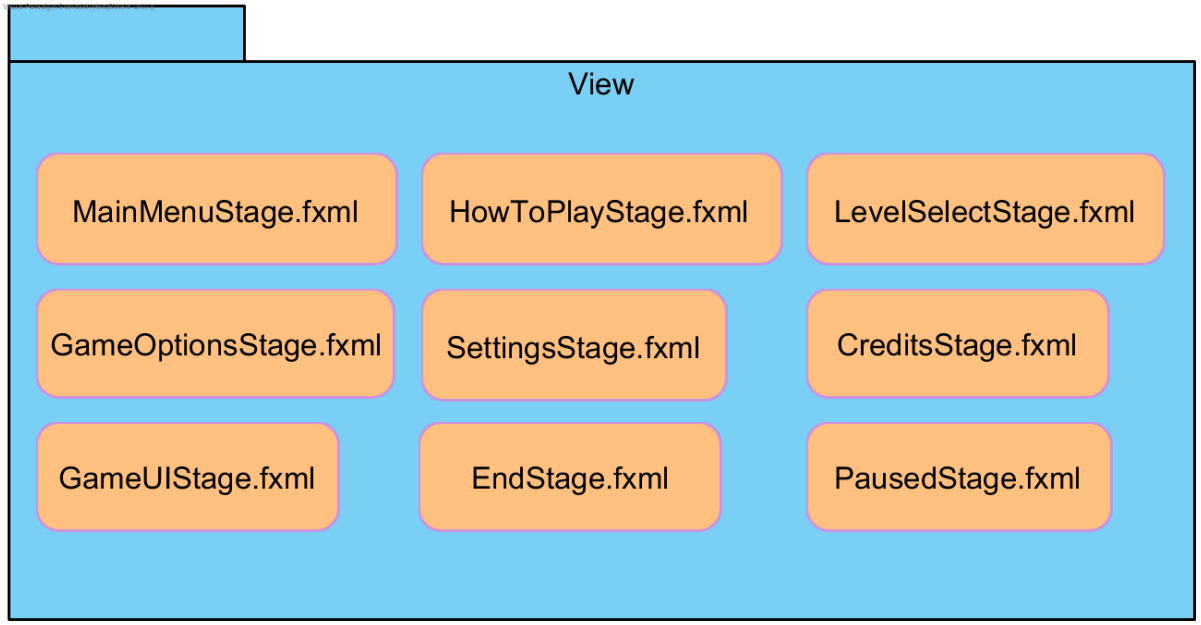

Figure 7

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 purposes of these FXML files and what they display.

3.1.1 MainMenuStage.fxml

This FXML is responsible for displaying the view of our main menu. It consists of all the buttons that allows the player to navigate to other menus, namely the "Play" button, "Level Select" button, "How To Play" button, "Settings" button, "Credits" button and "Exit" button.

3.1.2 LevelSelectStage.fxml

In the level select stage, the available levels are displayed to the player, with a "Main Menu" button to navigate back to main menu.

3.1.3 HowToPlayStage.fxml

In here, A short description of how to play game is displayed and there is a “Main Menu” button to navigate back to main menu.

3.1.4 SettingsStage.fxml

In here, There are two checkbox for sound and music, player can open/close sound and music with these checkboxes. Also, there is a “Main Menu” button to navigate back to main menu.

3.1.5 CreditsStage.fxml

In here, Developers of the game will be displayed and there is a “Main Menu” button to navigate back to main menu.

3.1.6 GameOptionsStage.fxml

There are 3 radio buttons for selecting difficulty, another 4 radio buttons for selecting player numbers, a list for selecting game mode. Also, there are two buttons for “Main Menu” and “Start”. “Main Menu” button returns to main menu and “Start” button will start the game.

3.1.7 GameUIStage.fxml

The pattern is displayed on the top left corner and player board is displayed on the top right corner. The faces of the cubes are displayed on the bottom right corner. Also, there is a “Main menu” botton to navigate back to main menu.

3.1.8 EndStage.fxml

There are 3 buttons. “Restart” for restarting the game with same game options, “Game Options” to navigate back to game options page and “Back to Main Menu” button for returning to main menu page.

3.1.9 PausedStage.fxml

PausedStage will be displayed when the player clicks on the pause button during a singleplayer game. There are 3 buttons on the stage. One for continuing the current game, one for restarting the game with the same game options and the other for going back to game options menu.

3.2 Controller: Game Management Subsystem

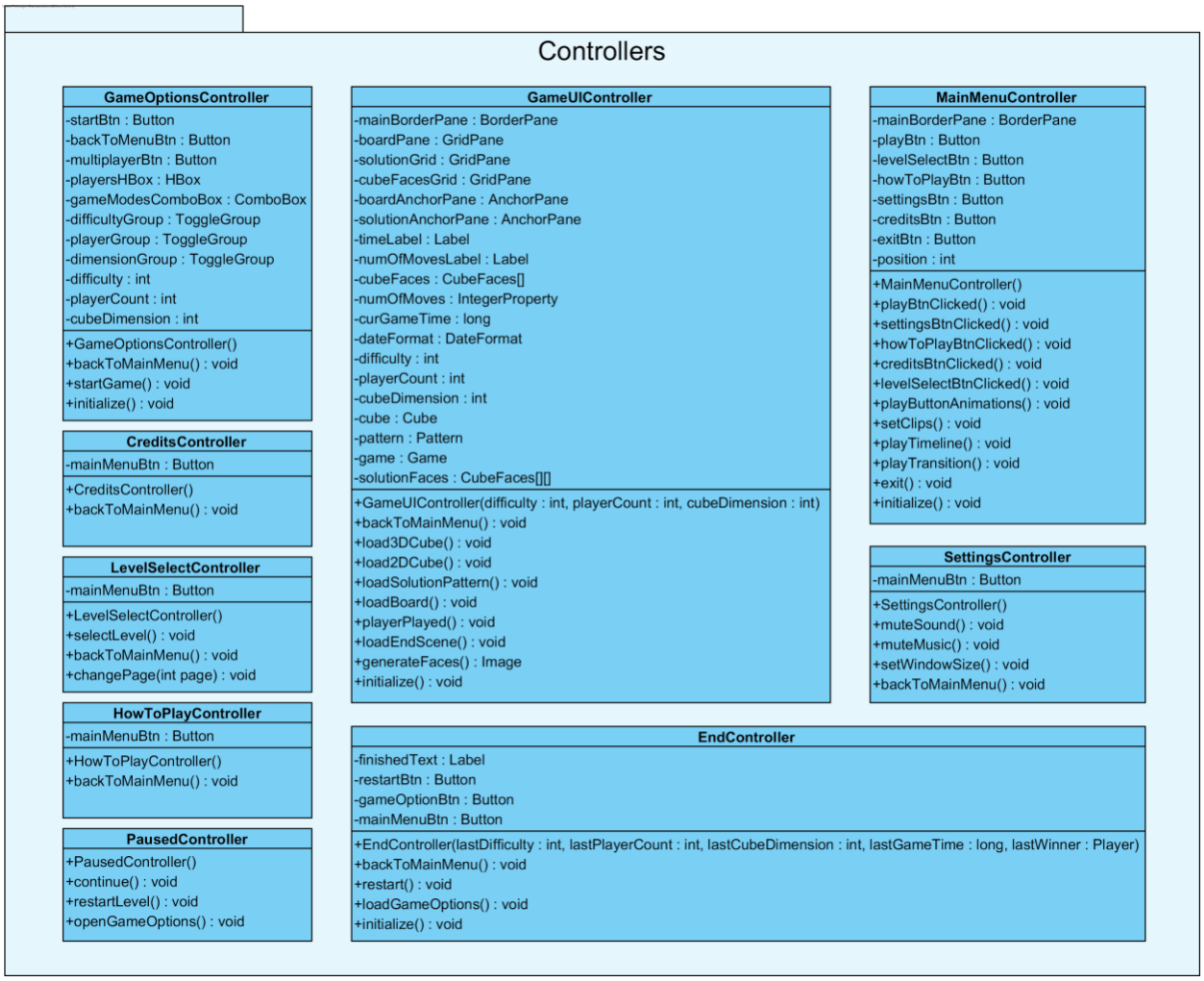

Figure 8

In the following controller classes will be explained.

3.2.1 GameUIController

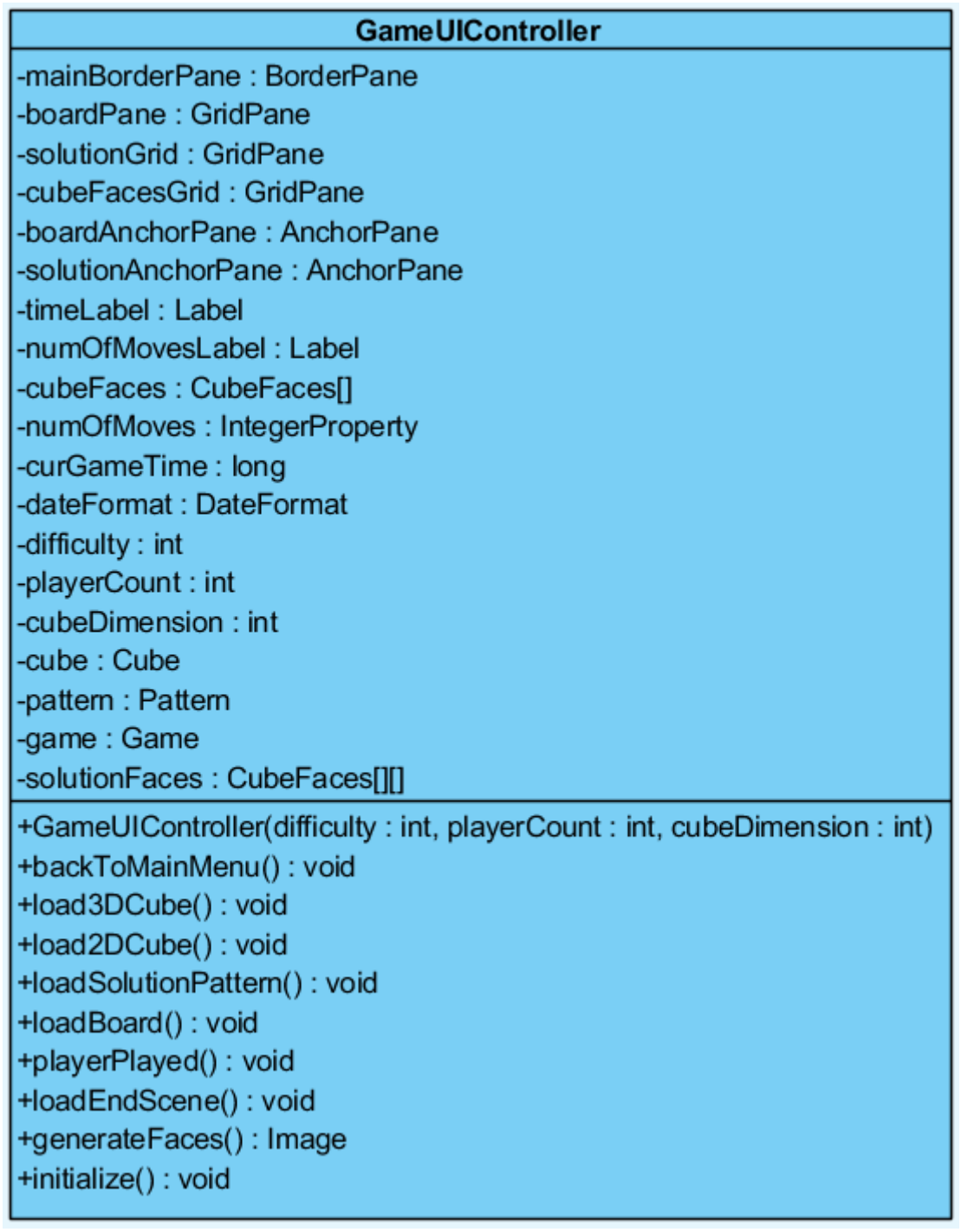

Figure 9

A class used during a round of the game to communicate with the view and the model. It displays the current model and updates the model according to the user input from the view. Then, when the round is over it switches the control to another class.

Attributes:

- BorderPane mainBoardPane: This is the parent that contains all of the view objects and lays them out in a conventional border pane manner (left, right, center, top, bottom).
- GridPane boardPane: This pane contains the children related to the game board that player places the cube faces on. Using a grid style pane makes sense here since the game board is either of size 3x3, 4x4 or 5x5.
- GridPane solutionPane: Similar to boardPane, this pane containts the children related to the solution board that shows the player which pattern he/she is supposed to put together.
- GridPane cubeFacesGrid: This pane containts the available cube faces to the player if the game mode is chosen to be 2D.
- Cube cube: The cube class allows us to match the cube face the player chooses and places on the board with the correct cube face.
- Pattern pattern: The pattern class is used to generate a random pattern at the start of each game.
- Game game: The game class is responsible for starting the game and keeping track of each move of every player that is included in the game. It is responsible for determining the winner as well.

The rest of the attributes are left out for simplicity because their intentions are clear by their names. For instance, timeLabel displays the time, numOfMoves keeps track of the number of moves, playerCount stores the number of players in the game and so on.

Constructor:

- GameUIController(int difficulty, int playerCount, int cubeDimension): This constructor takes in the parameters difficulty (3x3, 4x4 or 5x5), playerCount (1, 2, 3 or 4) and cubeDimension (2 or 3) and stores them. It also initializes all the attribute objects.

Methods:

- void initialize(): This method is available in all of the controller classes by default in the JavaFX library. It is called automatically when the class is constructed and is used to do any necessary work before the view is displayed. As an example, for GameUIController, initialize method calls all of the methods prefixed with “load” below and also starts the game using the Game class.
- void load3DCube(): Creates the 3D Cube, makes all of its rotation bindings and drag bindings, creates a perspective camera for the 3D effect and adds them to the center of mainBorderPane. Called only when the game mode is 3D.

- void load2DCube(): Loads the 2D cube faces and adds them as the children of cubeFacesGrid. Called only when the game mode is 2D.
- void loadBoard(): Loads the game board and makes it ready for all drag bindings.
- void playerPlayed(): Called whenever a player makes a move. This method calls game.playerMove(...) method and calls loadEndScene() if there are any winners.
- void loadEndScene(): Loads the end scene onto the current stage. Called when there are any winners.
- Image generateFaces(): Generates the faces for the 3D cube and returns it as a single image.

3.2.2 MainMenuController

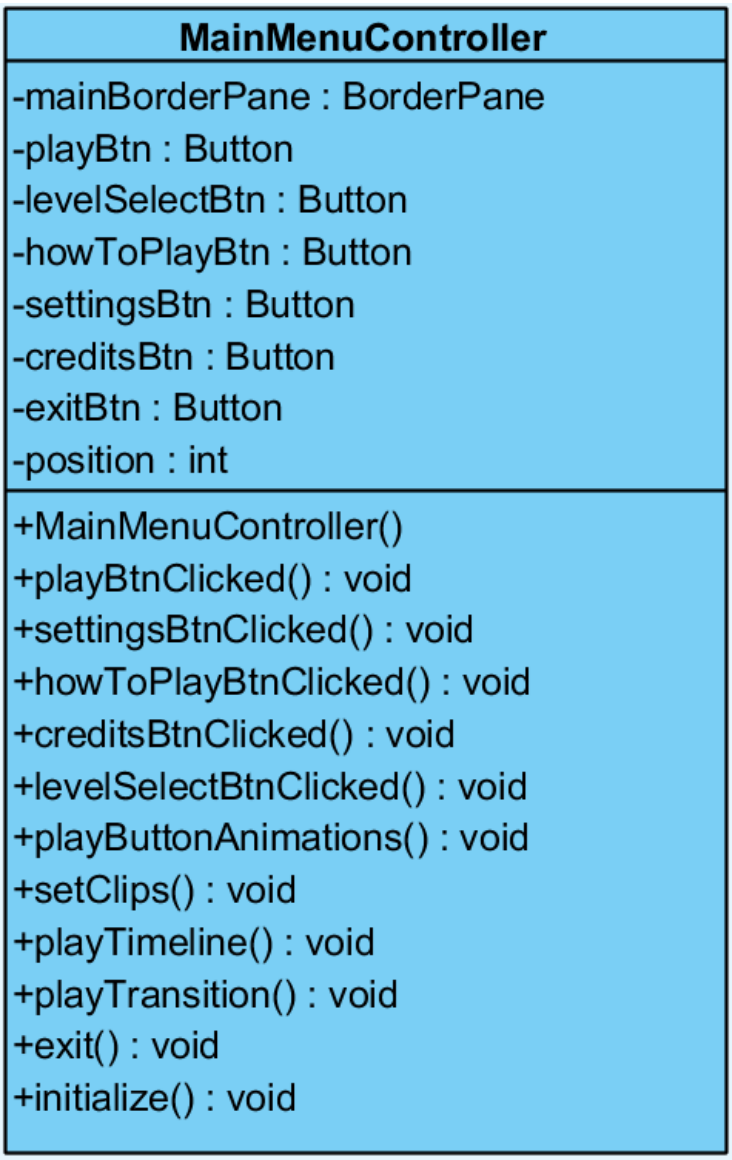

Figure 10

A class used to act as a bridge between different views. Some of the views can only be accessed through the main menu. Others eventually return to the main menu. MainMenuController is used to control the state changes and the main menu view itself.

Attributes:

- The attributes are simply references to the buttons on the screen. Each attribute has the name of its corresponding button.

Methods:

- void initialize(): The initialize method calls the playButtonAnimations method, which plays a slider animation for each button. setClips, playTimeline and playTransition methods are all related to playing the button animations.
- void playBtnClicked(): This method loads the game options scene when the play button is clicked.
- The remaining methods postfixed with “Clicked” load the corresponding scenes onto the stage when their buttons are clicked. For instance, settingsBtnClicked method loads the settings scene when the settings button is clicked.

3.2.3 GameOptionsController

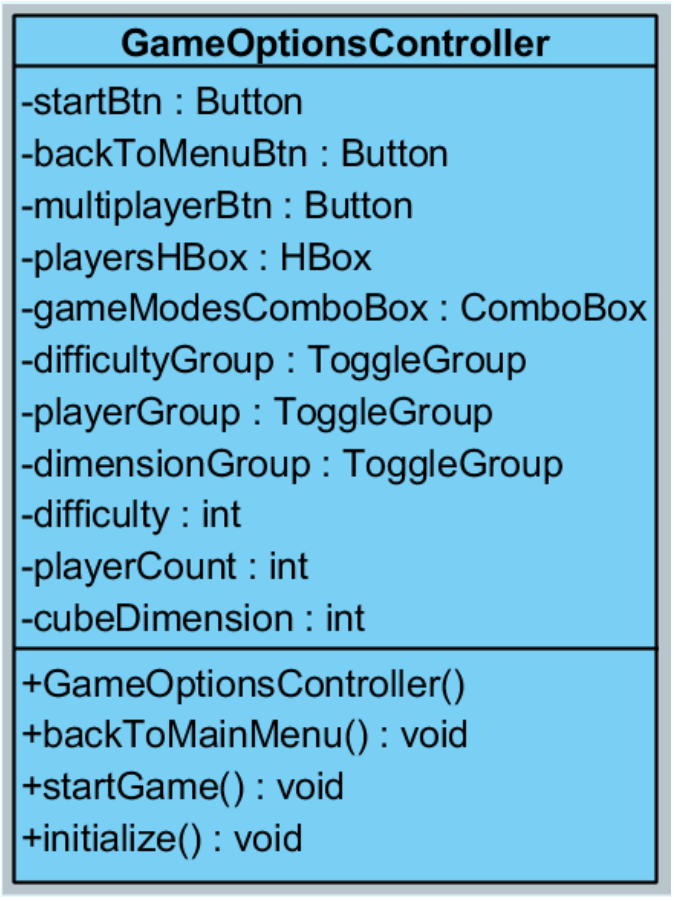

Figure 11

A class used to display and choose from options such as player count and difficulty for a round of the game. Then, either the player starts the game or returns to the main menu.
Attributes:

- HBox playersHBox: An HBox is a layout that lays its children horizontally. This layout is used to display the game options.
- ComboBox gameModesComboBox: This combo box displays the available game modes in a drop down list manner.
- ToggleGroups: Toggle groups are used to have only one toggle button active in that current group. These groups make sense in a game options scene in places where only one option can be selected.
- int difficulty: Whether the game board is 3x3, 4x4 or 5x5, this variable takes the values 3, 4 or 5 respectively.
- int playerCount: Stores the player count chosen by the player.
- int cubeDimension: Whether the game is 2D or 3D, this variable takes the values 2 or 3 respectively.

Methods:

- void initialize(): This method is used to assign all the necessary listeners to the view objects. For instance, a change listener is assigned to the difficultyGroup so that whenever the value in there changes, the difficulty variable changes as well.
- void startGame(): Loads the game view whenever the player clicks start with the current difficulty, playerCount and cubeDimension settings.

3.2.4 CreditsController

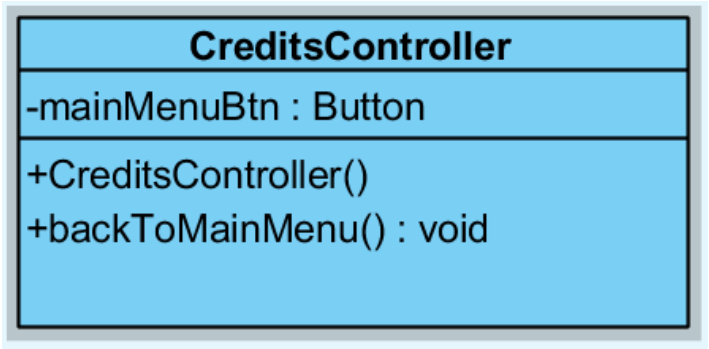

Figure 12

A class used to display the credits and then, when the user is done, return to the main menu.
Attributes:

- Button mainMenuBtn: The button that the player can click on to go back to the main menu.

Methods:

- void backToMainMenu(): When the mainMenuBtn is clicked, loads the main menu scene onto the stage.

3.2.5 SettingsController

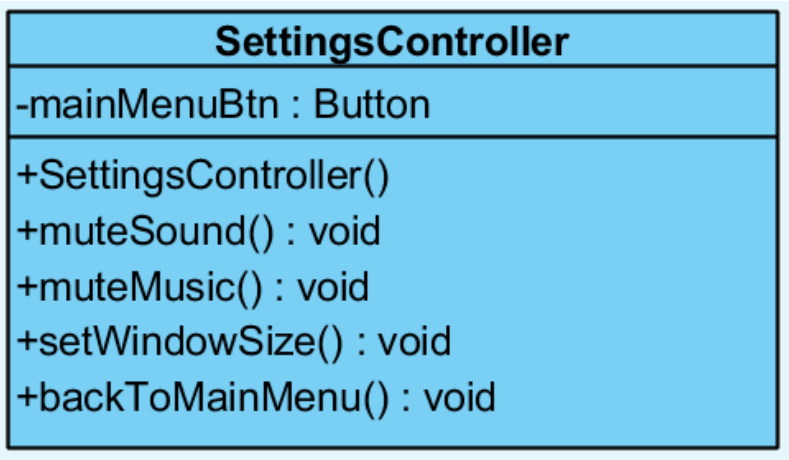

Figure 13

A class used to display the settings and through the user interface elements set the sound and music options. Then, when the user is done, returns to the main menu.

Attributes:

- Button mainMenuBtn: The button that the player can click on to go back to the main menu.

Methods:

- void muteSound(): Can be toggled to mute/unmute the sound.
- void muteMusic(): Can be toggled to mute/unmute the music.
- void setWindowSize(): Used to choose a window size to play on, like 800x600 or 1920x1080.

3.2.6 HowToPlayController

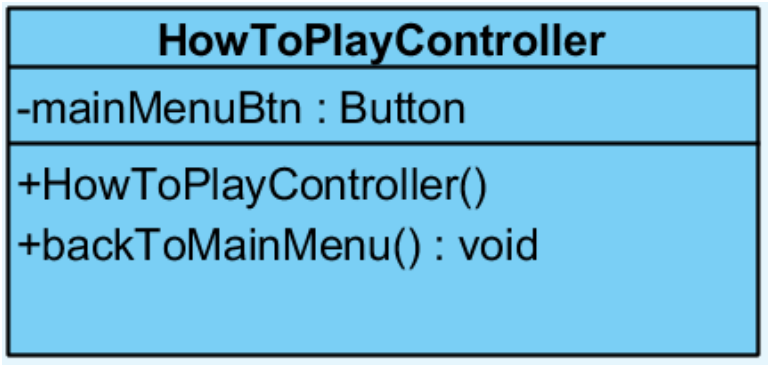

Figure 14

A class used to display the tutorial and then, when the user is done, return to the main menu.

Attributes:

- Button mainMenuBtn: The button that the player can click on to go back to the main menu.

Methods:

- void backToMainMenu(): When the mainMenuBtn is clicked, loads the main menu scene onto the stage.

3.2.7 LevelSelectController

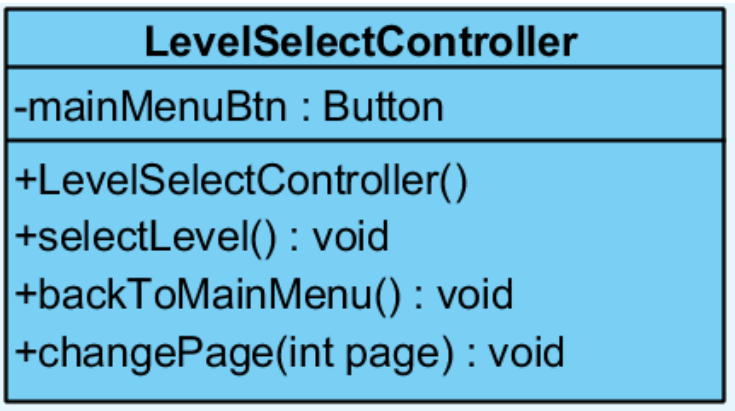

Figure 15

A class used to allow the player to play in puzzle mode. In the puzzle mode, player plays through premade levels and receives a performance rating for each level. This controller is used to select the level to play or return to the main menu.

Attributes:

- Button mainMenuBtn: The button that the player can click on to go back to the main menu.

Methods:

- void selectLevel(): Used to select a particular level and load it into the scene.
- void changePage(int page): Changes the current page to display other levels with the navigation arrows in the scene if all the levels don't fit the current scene.

3.2.8 EndController

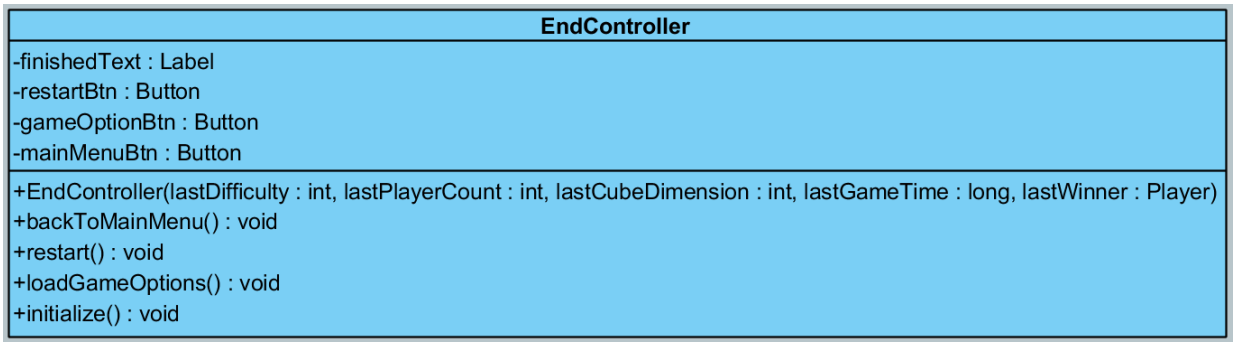

Figure 16

A class used to display end of the game information to the user and allow the user to replay the game or return to the main menu.

Attributes:

- Button mainMenuBtn: The button that the player can click on to go back to the main menu.
- Label finishedText: Displays an informative text to the player, like who won, in how many seconds did they finish and how many moves did they make.
- Button restartBtn: When clicked, restarts the last game with the same game options.

Constructor:

- EndController(int lastDifficulty, int lastPlayerCount, int lastCubeDimension, long lastGameTime, Player lastWinner): Takes parameters related to the last game that ended, like the game options of the last game or the winner.

Methods:

- void initialize(): Initialize method is used to customize the finishedText based on the information from the last game.
- void restart(): Restarts the last game with the same game options when the restartBtn is clicked.
- void loadGameOptions(): Loads the game options menu to quickly start a new game when the gameOptionsBtn is clicked.
- void backToMainMenu(): When the mainMenuBtn is clicked, loads the main menu scene onto the stage.

3.2.9 PausedController

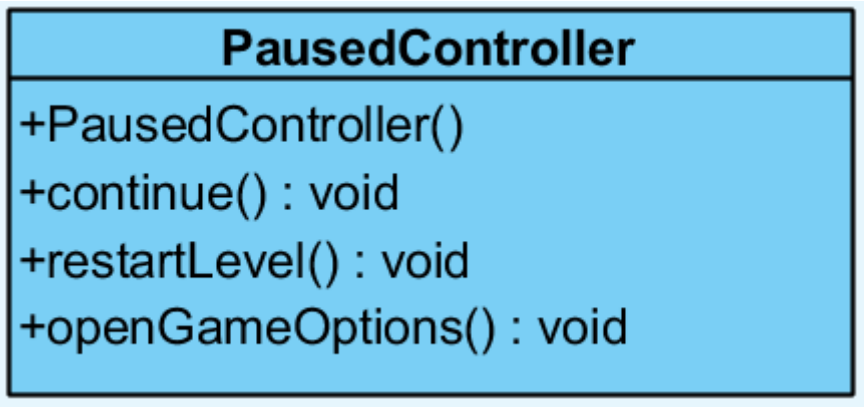

Figure 17

A class used to display a pause screen to the user if he/she is playing in the SinglePlayer mode.

Methods:

- void continue(): Continues from where the game was paused.
- void restartLevel(): Restarts the level with the same game options.
- void openGameOptions(): Returns to the game options menu to start a new game with different options or quit.

3.3 Model: Game Object Subsystem

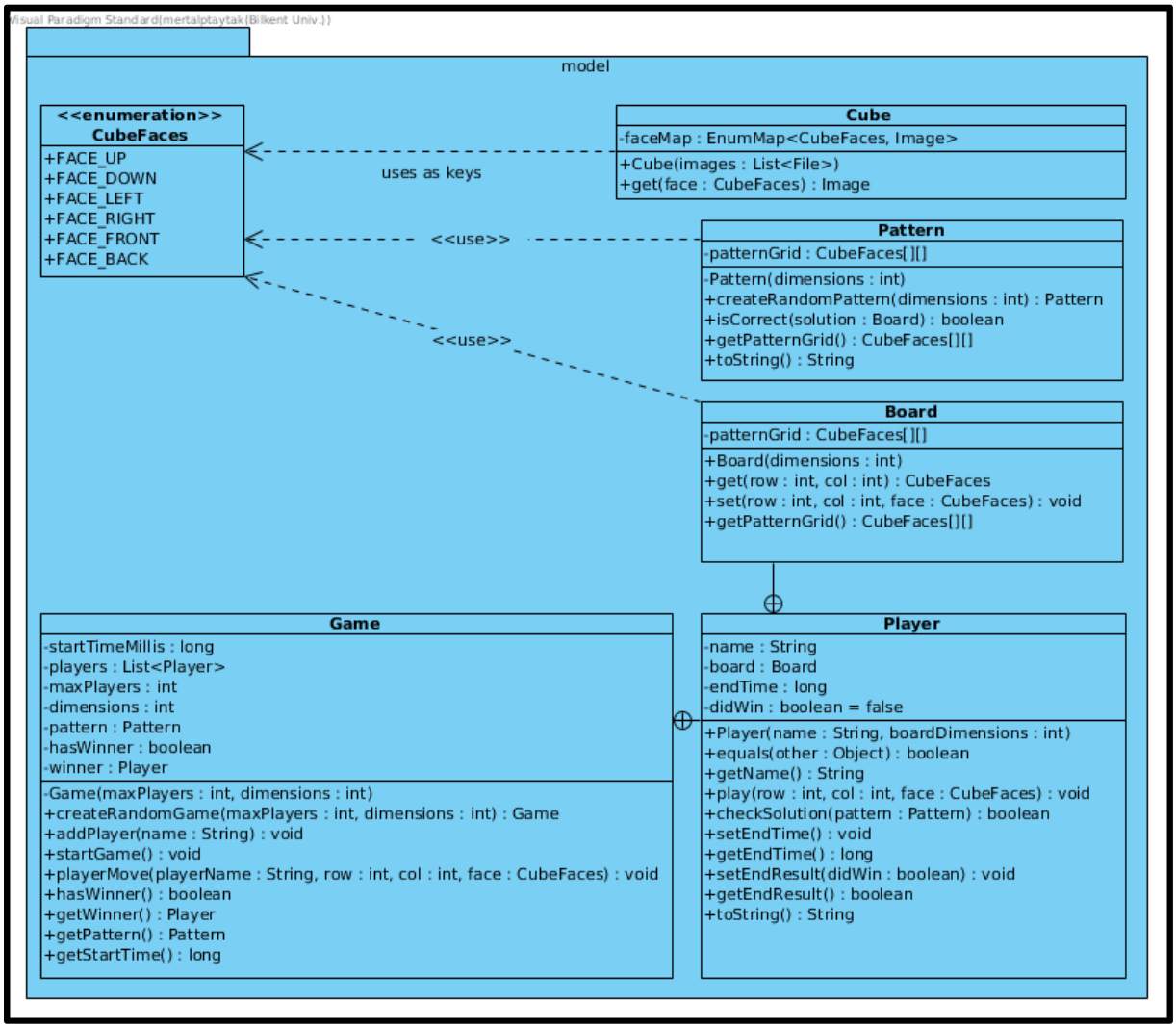

Figure 18

In the following model classes will be explained.

3.3.1 CubeFaces

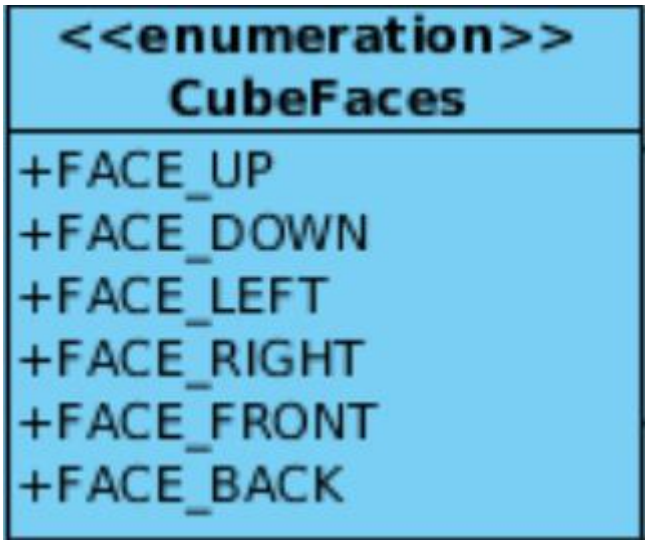

Figure 19

An enumeration used to keep label the faces of a cube. Each of these attributes FACE_UP, FACE_DOWN, FACE_LEFT, FACE_RIGHT, FACE_FRONT, FACE_BACK holds one side of the cube.

3.3.2 Cube

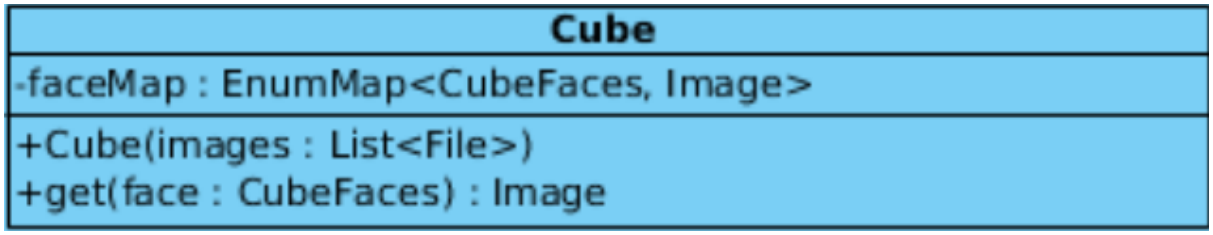

Figure 20

A class whose instances carry a map of cube faces to images. This mapping is used to render the faces of a cube instance.

Attributes:

- EnumMap<CubeFaces, Image> faceMap: It is a mapping of CubeFaces to Images. Which is used to turn sides of the cube into renderable images.

Constructor:

- Cube(images: List<File>): Takes a list of six image files and maps cube faces to them based on some preset naming rules.

Methods:

- get(face: CubeFaces): Image: Takes a CubeFace enumeration and returns the image associated with it using the internal faceMap.

3.3.3 Pattern

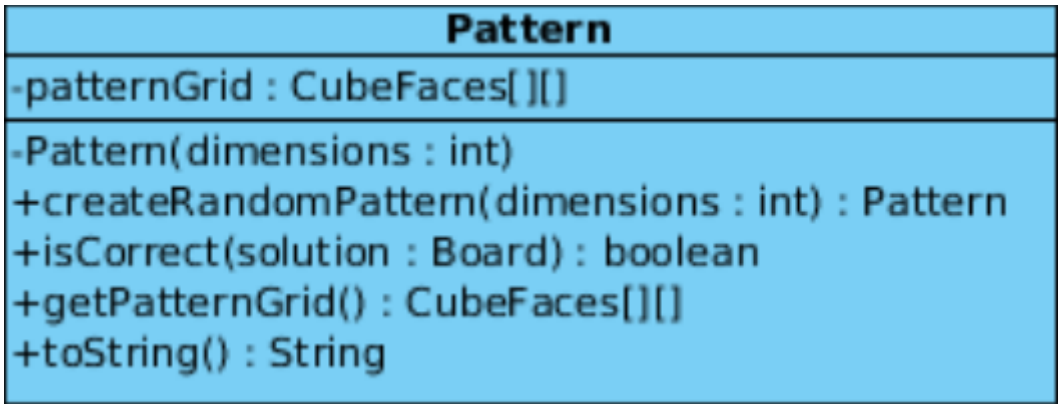

Figure 21

A class that contains a initialized double array of cube faces. Any Board can be compared with the internal grid to check if they are equal or in other words if the Board is correct. Currently the patterns are created randomly, however there will be premade patterns in the next implementation.

Attributes:

- patternGrid: CubeFaces[ ][ ]: A two dimensional array of CubeFaces that is used to store the correct pattern.

Constructor:

- Pattern(dimensions: int): Creates a Pattern with the two dimensional array of given size. It is set to private.

Methods:

- createRandomPattern(dimensions: int): Pattern: Creates a random Pattern of given dimensions. It is a static method.
- isCorrect(solution: Board): boolean: Compares the grid inside the given Board with the internal grid to check if the given Board is correct.

3.3.4 Board

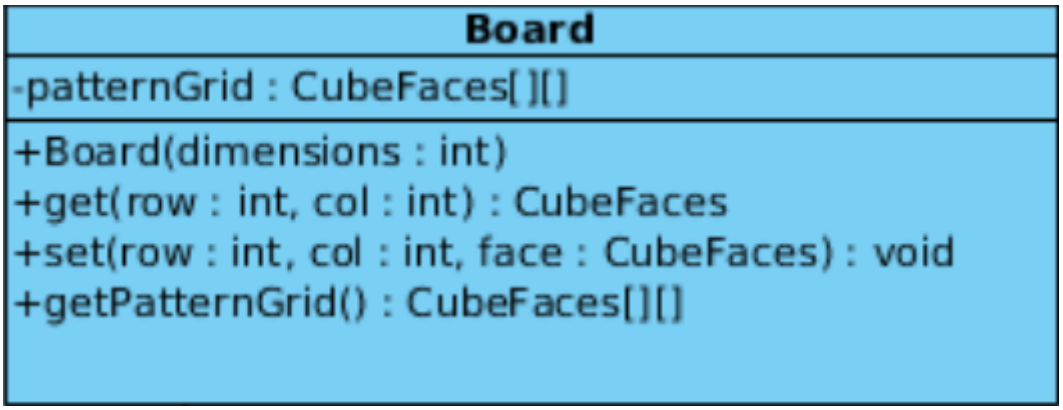

Figure 22

A class used to keep track of player moves. It contains a double array of cube faces. Each location on the board can be accessed through the use of set and get methods.

Attributes:

- patternGrid: CubeFaces[ ][ ]: A two dimensional array of CubeFaces to keep track of player boards.

Constructor:

- Board(dimensions: int): Creates a board with the two dimensional array of given size.

Methods:

- get(row: int, col: int): CubeFaces: Gets the CubeFaces stored in the patternGrid at the given indices.
- set(row: int, col: int, face: CubeFaces): Writes the given CubeFaces object into the patternGrid at the given indices.
- getPatternGrid(): CubeFaces[ ][ ]: Gets the internal two dimensional array of CubeFaces taken from the patternGrid.

3.3.5 Player

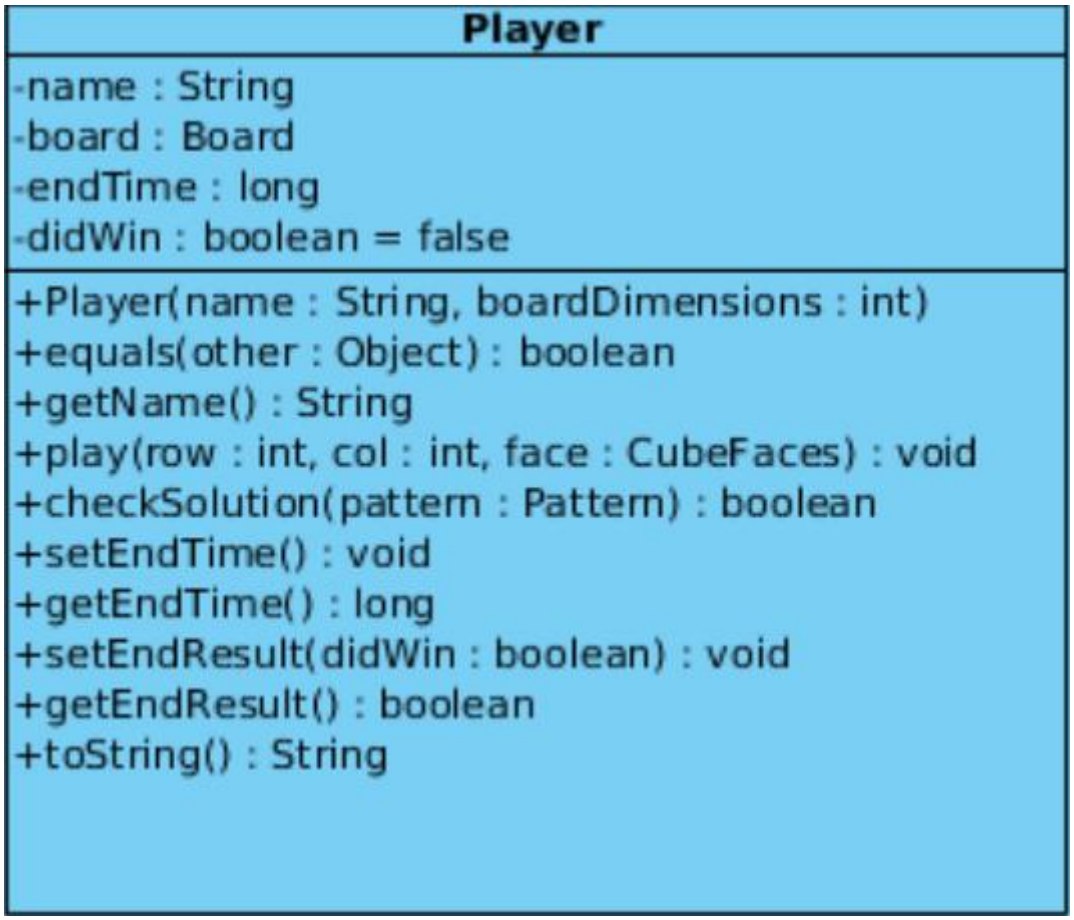

Figure 23

A class used to represent the players that play the game. Along with the identifying information and the board, there are also other variables to keep track of the player state in a round of the game.

Attributes:

- String name: Name of the player.
- Board board: Board on which this player plays.
- long endTime: In how long the player has finished the game.
- boolean didWin: Whether or not the player won the game.

Constructor:

- Player(String name, int boardDimensions): Constructs the player with a given name and board dimension (3x3, 4x4 or 5x5).

Methods:

- void play(int row, int col, CubeFaces face): Each player has his own game state array, which is a 2D array representing where on the board the player played and what he/she played. This play method will add the given cube face to a location on the array which is at the given row and column.

- boolean CheckSolution(Pattern pattern): Checks whether the player's solution matches the given pattern.

3.3.6 Game

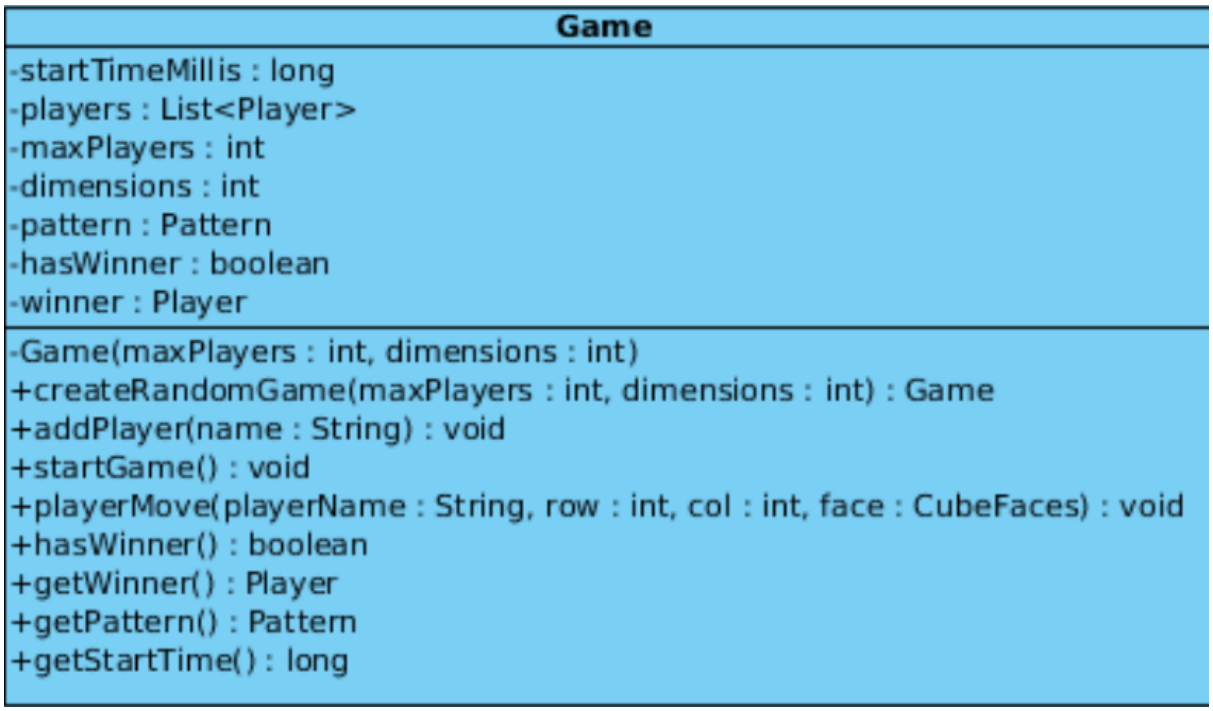

Figure 24

A class used to represent a single round of the game. It contains players, the goal pattern and other information that is relevant to construct the game and keep track of the game status. The game is initialized through the use of pattern constructors and the player adder method. Then, the game is started and each move made by the players is recorded through the playerMove() method. After every move the board of the player is checked against the correct pattern to find if the player has finished the game and player time is tracked. Depending on the game mode, a winner is determined by different metrics.

Attributes:

- long StartTimeMillis: The starting time of the game based on the system time.
- List<Player> players: A list of players that are in this game.
- int maxPlayers: Maximum number of players this game allows.
- int dimensions: The dimensions of the board this game is being played on.
- Pattern pattern: The correct pattern each player has to match for this game.
- boolean hasWinner: Whether or not the game has a winner.
- Player winner: If the game has a winner, this variable stores the player that won, otherwise it is null.

Constructor:

- Game(int maxPlayers, int dimensions): Constructs the game object that accepts maximum number of players equal to maxPlayers parameter and has the board size equal to dimensions parameter.

Methods:

- createRandomGame(maxPlayers: int, dimensions: int): Game: Returns a randomly created Game object with the given maximum number of players and pattern dimensions.
- addPlayer(name: String): Creates and adds a Player with the given name into the game if there is room for a new player.
- startGame( ): Records the current time as the Game start time.
- playerMove(playerName: String, row: int, col: int, face: CubeFaces): Finds the Player with the given name and plays the given move on their model.
- hasWinner( ): boolean: Returns if the Game has a winner yet.
- getWinner( ): Player: Returns the winner of the Game if it exists.
- getPattern( ): Pattern: Gets the Pattern object used in the Game.
- getStartTime( ): long: Gets the starting time of the Game.

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.

Rapid Development vs Functionality:

Since we develop desktop version of the game Q-bitz from the very beginning, firstly we aim to finish main parts of the game within the scope of our design. Then, we plan to increment the number of functions to make the game more interesting and increase the variance however, this features will be added after we are done with the basics of the game. Extra features are considered later and implemented without affecting our pace when we implement the game.

4.2 Final Object Design

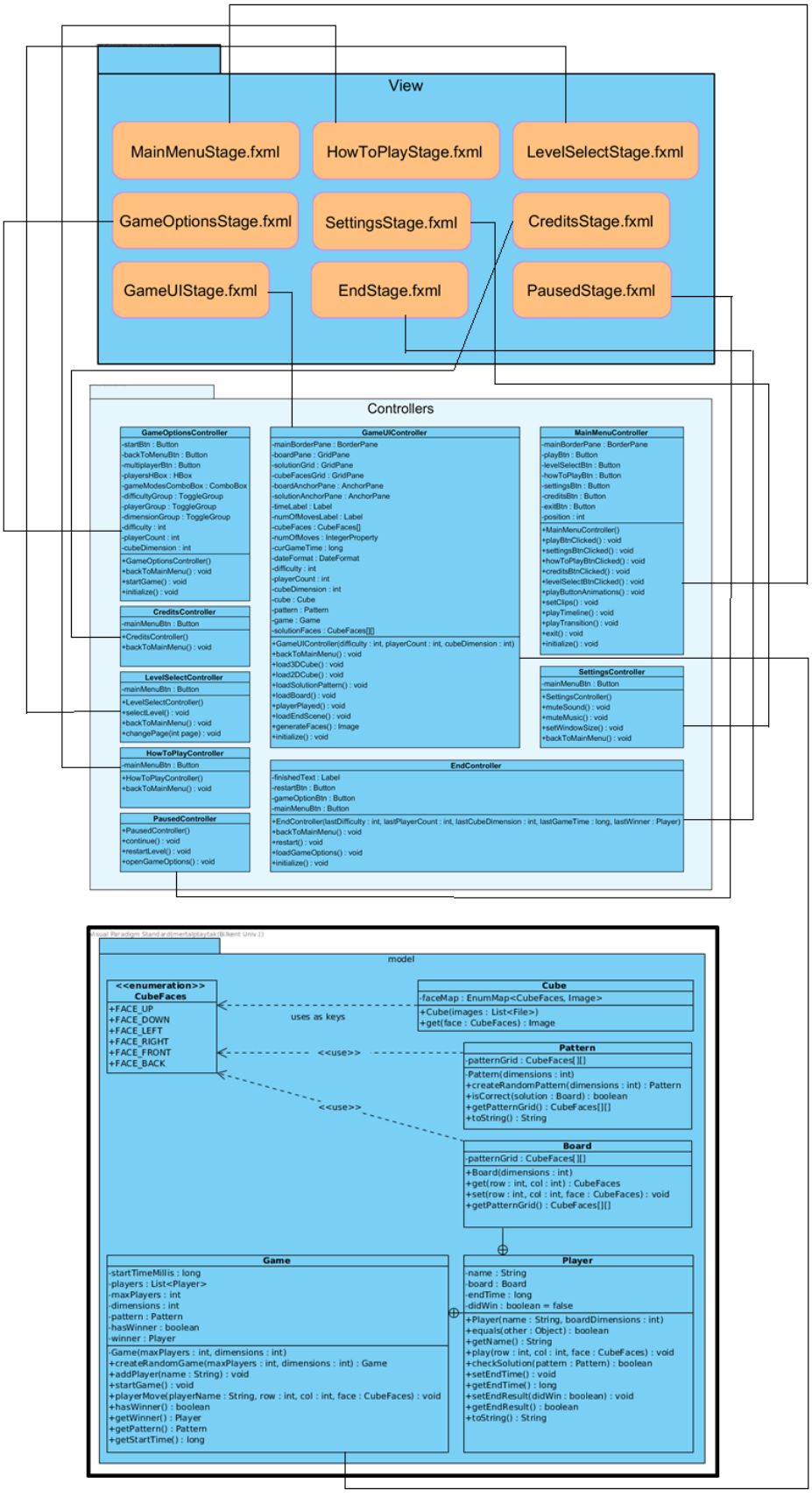

Figure 25

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 scene graph 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.

5. Improvement Summary

We have made some rational changes in our project since iteration one. From the beginning of our implementation phase, lots of classes clearer in our minds so we have added and removed some classes or some methods from our design. It was much easier to understand what was really necessary and what was not. Thus, this was is definitely an improvement for our project. Changes are as follows:

- Section 1.3 Acronyms and Abbreviations has been explained.
- Section 2.2 Hardware and software mapping has been written in detail and supported by a deployment diagram and one node added to represent general relationship between machines and hardware component.
- Section 2.3 Persistent Data Management section has been fixed and extended for better comprehension.
- Section 2.4 has been supported by an access matrix.
- Section 4.1 portability vs efficiency trade-off has been removed since the feedback says it is not good, instead rapid development vs functionality part has been added.
- In section 3, Model View and Controller subsystems are explained much detailly, and some classes of them have been changed based on our new design and the feedback. In this way we clarify our design.
- In final object design fxml related representation problems are fixed (controllers are placed in controller subsystem as section 3). Therefore, view and controllers are separated properly as feedback suggested.
- Class Interfaces part from iteration 1 has been removed since it is no longer necessary and our design has been altered as we move.

We have learned from the feedback from the first iteration and add more explanation to every class we have. This brings clarification for anyone that reads this design document and construct the project in their minds.

6. Glossary & References

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

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

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

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

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

[6] https://docs.oracle.com/javase/8/javafx/api/